

Problema del Viajante: Búsqueda Exhaustiva, Heurística y Aplicación de Algoritmos Genéticos

15/10/2019

UTN FRRO | Profesores: Díaz – Lombardo

ALGORITMOS GENÉTICOS 2019 – Trabajo Práctico 3

ANTONELLI (44852) – RECALDE (44704) – ROHN (41355)

Índice

[Introducción 2](#_Toc22775266)

[Código 3](#_Toc22775267)

[BACKEND 3](#_Toc22775268)

[FRONTEND 19](#_Toc22775269)

[Explicacion de Funcionamiento 27](#_Toc22775270)

[Salida de Pantalla 30](#_Toc22775271)

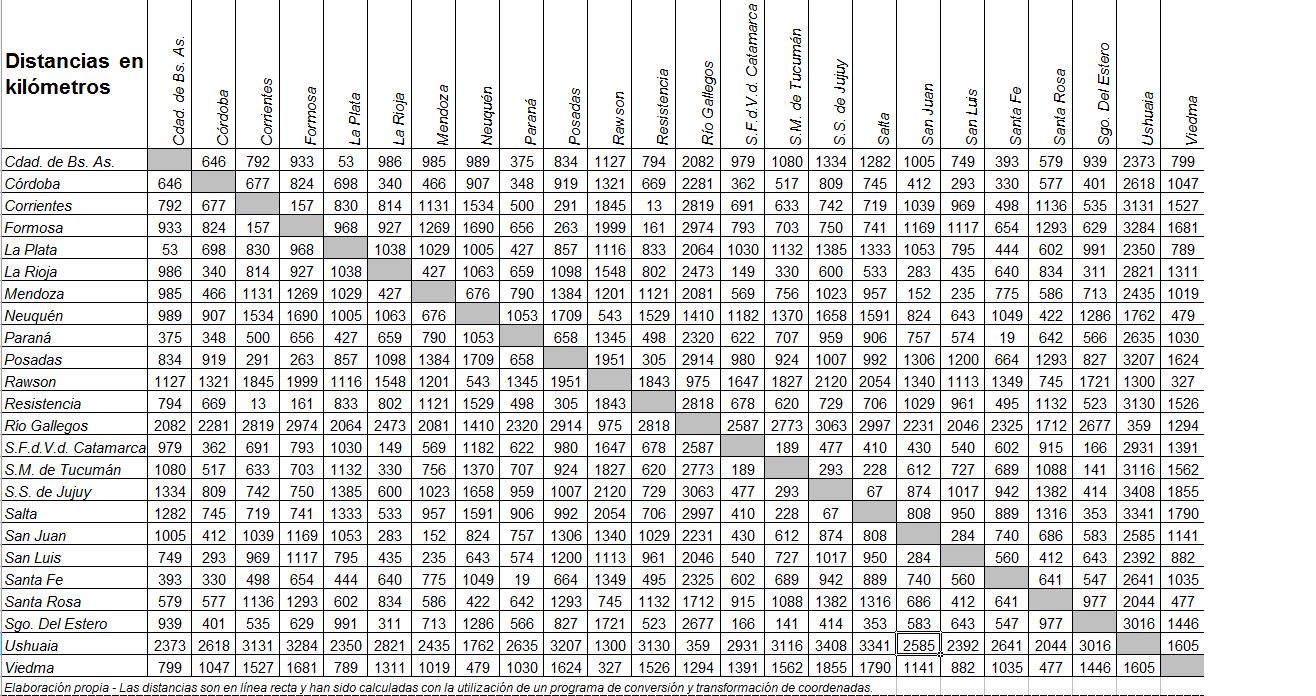
[Conclusiones 34](#_Toc22775272)

# Introducción

**El Problema del Viajante**

Consiste en encontrar una ruta que, comenzando y terminando en una ciudad concreta, pase una sola vez por cada una de las ciudades y minimice la distancia recorrida por el viajante.

Siendo N ciudades de un territorio, la distancia entre cada ciudad viene dada por la matriz D: NxN, donde d[x,y] representa la distancia que hay entre la ciudad X y la ciudad Y.



**Ejercicio 1**

Resolver el Problema del viajante utilizando una Búsqueda Exhaustiva.

**Ejercicio 2**

1. Resolver el ejercicio anterior permitiendo ingresar el inicio del recorrido y usando la heuristica: “Desde cada ciudad ir a la ciudad mas cercana no visitada”. Ademas presentar un mapa de la Republica con el recorrido realizado indicando la partida, el recorrido completo y la longitud del trayecto.
2. Buscar el minimo recorrido para visitar todas las capitales de la Republica Argentina siguiendo la heuristica mencionada en el punto A. Motrar el recorrudo y la longitud del trayecto.
3. Hallar la ruta de distancia minima que logre unir todas las capitales de provincias de la Republica Argentina, utilizando un algoritmo genetico.

# Código

***Código perteneciente a la resolución de todos los ejercicios***

## BACKEND

**app.py**

1. #!/usr/bin/env python
2. # -\*- coding: utf-8 -\*-
4. """
5. TP3: TRAVELING SALESMAN PROBLEM - ENUNCIADO:
6. 1. Hallar la ruta de distancia mínima que logre unir todas las capitales de provincias de la República Argentina,
7. utilizando un método exhaustivo. ¿Puede resolver el problema? Justificar de manera teórica.
8. 2. Realizar un programa que cuente con un menú con las siguientes opciones:
9. a) Permitir ingresar una provincia y hallar la ruta de distancia mínima que logre unir todas las capitales de
10. provincias de la República Argentina partiendo de dicha capital utilizando la siguiente heurística:
11. "Desde cada ciudad ir a la ciudad más cercana no visitada". Recordar regresar siempre a la ciudad de partida.
12. Presentar un mapa de la República con el recorrido indicado. Además indicar la ciudad de partida, el recorrido
13. completo y la longitud del trayecto. El programa deberá permitir seleccionar la capital que el usuario desee
14. ingresar como inicio del recorrido.
15. b) Encontrar el recorrido mínimo para visitar todas las capitales de las provincias de la República Argentina
16. siguiendo la heurística mencionada en el punto a. Deberá mostrar como salida recorrido y longitud del trayecto.
17. c) Hallar la ruta de distancia mínima que logre unir todas las capitales de provincias de la República Argentina,
18. utilizando un algoritmo genético.
19. FECHA DE ENTREGA: 30/09/2019
20. --> Genetic-Algorithm TP3 --- V1.0 ---  Created on 3 sep. 2019
21. Antonelli, Nicolás - Recalde, Alejando - Rohn, Alex
22. """
24. **from** flask **import** Flask
25. **from** flask **import** render\_template
26. **from** citiesManager **import** CitiesManager
28. # App Initialization
29. app = Flask(\_\_name\_\_)
31. # Cities Initialization
32. cities = CitiesManager()

35. # Index
36. @app.route('/')
37. **def** hello\_world():
38. **return** render\_template("index.html")

41. # Best Track starting from a Selected City
42. @app.route('/Map/<root\_city\_name>')
43. **def** show\_track\_from(root\_city\_name):
44. track = cities.get\_best\_track\_from(root\_city\_name)
46. **print**(track)
48. **return** render\_template("showMap.html", track=track)

51. # Best Track of all Tracks
52. @app.route('/Map')
53. **def** show\_best\_track():
54. best\_track = cities.get\_best\_track()
56. **return** render\_template("showMap.html", track=best\_track)

59. # Best Track Using Genetic Algorithm
60. @app.route('/Map/Ag')
61. **def** show\_track\_with\_ag():
62. track\_ag = cities.get\_track\_with\_ag()
64. **return** render\_template("showMap.html", track=track\_ag)

67. # Main Function
68. **if** \_\_name\_\_ == '\_\_main\_\_':
69. app.run()

**citiesManager.py**

1. #!/usr/bin/env python
2. # -\*- coding: utf-8 -\*-
4. **from** neighbourCitiesRepo **import** NeighbourCitiesRepo
5. **from** city **import** City
6. **from** cityToVisit **import** CityToVisit
7. **from** population **import** Population
8. **from** chromosome **import** Chromosome
10. # Initialization of All Cities
11. **class** CitiesManager(object):
12. **def** \_\_init\_\_(self):
13. self.citiesRepo = NeighbourCitiesRepo()
15. buenos\_aires = City(name="Buenos Aires", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("Buenos Aires")
16. , lat=-34.6131516, long=-58.3772316,)
18. cordoba = City(name="Cordoba", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("Cordoba")
19. , lat=-31.4134998, long=-64.1810532)
21. corrientes = City(name="Corrientes", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("Corrientes")
22. , lat=-27.4806004, long=-58.8340988)
24. formosa = City(name="Formosa", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("Formosa")
25. , lat=-26.1775303, long=-58.1781387)
27. la\_plata = City(name="La Plata", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("La Plata")
28. , lat=-34.9214516, long=-57.9545288)
30. la\_rioja = City(name="La Rioja", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("La Rioja")
31. , lat=-29.4110508, long=-66.8506699)
33. mendoza = City(name="Mendoza", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("Mendoza")
34. , lat=-32.8908386, long=-68.8271713)
36. neuquen = City(name="Neuquen", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("Neuquen")
37. , lat=-38.9516106, long=-68.0590973)
39. parana = City(name="Parana", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("Parana")
40. , lat=-31.7319698, long=-60.5237999)
42. posadas = City(name="Posadas", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("Posadas")
43. , lat=-27.3670807, long=-55.89608)
45. rawson = City(name="Rawson", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("Rawson")
46. , lat=-43.3001595, long=-65.1022797)
48. resistencia = City(name="Resistencia", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("Resistencia")
49. , lat=-27.4605598, long=-58.9838905)
51. rio\_gallegos = City(name="Rio Gallegos", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("Rio Gallegos")
52. , lat=-51.6226082, long=-69.218132)
54. sfdvd\_catamarca = City(name="S.F.d.V.d. Catamarca", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("S.F.d.V.d. Catamarca")
55. , lat=-28.4695702, long=-65.7852402)
57. sm\_de\_tucuman = City(name="S.M. de Tucuman", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("S.M. de Tucuman")
58. , lat=-26.8241405, long=-65.2226028)
60. ss\_de\_jujuy = City(name="S.S. de Jujuy", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("S.S. de Jujuy")
61. , lat=-24.1945705, long=-65.2971191)
63. salta = City(name="Salta", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("Salta")
64. , lat=-24.7859001, long=-65.4116592)
66. san\_juan = City(name="San Juan", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("San Juan")
67. , lat=-31.5375004, long=-68.5363922)
69. san\_luis = City(name="San Luis", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("San Luis")
70. , lat=-33.2950096, long=-66.3356323)
72. santa\_fe = City(name="Santa Fe", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("Santa Fe")
73. , lat=-31.6333294, long=-60.7000008)
75. santa\_rosa = City(name="Santa Rosa", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("Santa Rosa")
76. , lat=-36.6166687, long=-64.2833328)
78. sgo\_del\_estero = City(name="Sgo. del Estero", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("Sgo. del Estero")
79. , lat=-27.7951107, long=-64.2614899)
81. ushuaia = City(name="Ushuaia", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("Ushuaia")
82. , lat=-54.7999992, long=-68.3000031)
84. viedma = City(name="Viedma", cities\_neig=self.citiesRepo.get\_near\_cities\_dict("Viedma")
85. , lat=-40.8134499, long=-62.9966812)
87. # Dictionary with all City Objects
88. self.cities = {
89. "Buenos Aires": buenos\_aires,
90. "Cordoba": cordoba,
91. "Corrientes": corrientes,
92. "Formosa": formosa,
93. "La Plata": la\_plata,
94. "La Rioja": la\_rioja,
95. "Mendoza": mendoza,
96. "Neuquen": neuquen,
97. "Parana": parana,
98. "Posadas": posadas,
99. "Rawson": rawson,
100. "Resistencia": resistencia,
101. "Rio Gallegos": rio\_gallegos,
102. "S.F.d.V.d. Catamarca": sfdvd\_catamarca,
103. "S.M. de Tucuman": sm\_de\_tucuman,
104. "S.S. de Jujuy": ss\_de\_jujuy,
105. "Salta": salta,
106. "San Juan": san\_juan,
107. "San Luis": san\_luis,
108. "Santa Fe": santa\_fe,
109. "Santa Rosa": santa\_rosa,
110. "Sgo. del Estero": sgo\_del\_estero,
111. "Ushuaia": ushuaia,
112. "Viedma": viedma,
113. }
114. **print**("Correct Initialization")
116. # Best Track starting with a Selected One
117. **def** get\_best\_track\_from(self, root\_city\_name):
118. **print**("Llego a get\_best\_track\_from, parametro: " + root\_city\_name)
119. **print**(self.cities[root\_city\_name].get\_lat())
120. # Create a "CityToVisit" object for Every city on the travel
121. root\_city = CityToVisit(
122. name=self.cities[root\_city\_name].get\_name(),
123. lat=self.cities[root\_city\_name].get\_lat(),
124. long=self.cities[root\_city\_name].get\_lng())
126. # All CityToVisit List and Total Travel Distance
127. visited\_cities = [root\_city]
128. accumulated\_distance = 0
130. # The List of Visited Cities must have the same length of all "Cities" list
131. **while** len(visited\_cities) <= len(self.cities):
132. last\_city = self.cities[visited\_cities[-1].get\_name()]
134. # On the last Loop: Go back to the Start City
135. **if** len(visited\_cities) == len(self.cities):
136. accumulated\_distance = accumulated\_distance + last\_city.get\_distance\_to(root\_city)
137. visited\_cities.append(root\_city)
138. **else**:
139. # Check that the next nearest city is not in the List of Visited Cities
140. nearest\_neig = last\_city.get\_nearest\_neig([c.name **for** c **in** visited\_cities])
141. # Add the new City on the Visited Cities List
142. visited\_cities.append(CityToVisit(
143. name=self.cities[nearest\_neig["name"]].get\_name(),
144. lat=self.cities[nearest\_neig["name"]].get\_lat(),
145. long=self.cities[nearest\_neig["name"]].get\_lng())
146. )
147. # Update Total Distance
148. accumulated\_distance = accumulated\_distance + nearest\_neig["distance"]
150. # https://stackoverflow.com/questions/21411497/flask-jsonify-a-list-of-objects
151. # Returns all cities on Visited Cities List (Serialized) and Total Distance
152. **return** {
153. "cities\_to\_visit": [c.serialize() **for** c **in** visited\_cities],
154. "accumulated\_distance": accumulated\_distance
155. }
157. # Best Track of all Tracks
158. **def** get\_best\_track(self):
159. # Get Best Track for the First existent City
160. best\_track = self.get\_best\_track\_from(list(self.cities)[0])
162. # Try every posible Track and Compare with the First One
163. **for** c **in** self.cities:
164. track = self.get\_best\_track\_from(c)
165. **if** track["accumulated\_distance"] < best\_track["accumulated\_distance"]:
166. best\_track = track
168. **return** best\_track
170. # Best Track Using Genetic Algorithm
171. **def** get\_track\_with\_ag(self):
172. # ImportantValues
173. iterationLimit = 1500  # Population Iterations
174. populationSize = 200  # Initial Population Size
175. crossoverProb = 0.75  # Probability of CrossOver
176. mutationProb = 0.4  # Probability of Mutation
178. track\_ag = None  # Initialization of Best Track
179. km\_ag = 0  # Initialization of Total Distance
181. # Initialize Chromosome
182. Chromosome.setCitiesDict(self.cities)
184. # First Population
185. pob = Population(populationSize, list(self.cities.keys()), crossoverProb, mutationProb)
187. # Iterations
188. **for** iterationCount **in** range(iterationLimit):
189. values = pob.showPopulation(iterationCount)
191. # In the last iteration, the chromosomes population mustn't reproduce
192. **if** iterationCount < iterationLimit - 1:
193. pob.reproduce(values["ElitChrom"], values["SecondElitChrom"])  # Reproduction of Actual Generation
194. **print**("------------")
196. # Last Reproduction Message
197. **print**("Last Generation Reached Correctly")
198. **print**("------------")
199. **print**()
200. **print**()
202. # Get the Best Results after last generation
203. track\_ag, km\_ag = pob.getBestTrackAg()
205. # Convert track\_ag in a citiesToVisit List
206. track\_ag\_cities = []
207. **for** city\_name **in** track\_ag.getRoute():
208. track\_ag\_cities.append(CityToVisit(name=city\_name, lat=self.cities[city\_name].get\_lat(),
209. long=self.cities[city\_name].get\_lng()))
211. root\_city = track\_ag.getRoute()[0]
212. track\_ag\_cities.append(CityToVisit(name=root\_city, lat=self.cities[root\_city].get\_lat(),
213. long=self.cities[root\_city].get\_lng()))
215. # Show Final Results
216. **print**("Best Track:", end=" ")
217. **for** i **in** range(len(track\_ag\_cities)):
218. **if** i < len(track\_ag\_cities) - 1:
219. **print**(track\_ag\_cities[i].get\_name(), end=", ")
220. **else**:
221. **print**(track\_ag\_cities[len(track\_ag\_cities)-1].get\_name(), "(Vuelta al Inicio)")
222. **print**("Total Distance:", km\_ag, "km, after", iterationLimit, "generations with", populationSize, "Chromosomes")
224. **return** {
225. "cities\_to\_visit": [c.serialize() **for** c **in** track\_ag\_cities],
226. "accumulated\_distance": km\_ag
227. }

**city.py**

1. #!/usr/bin/env python
2. # -\*- coding: utf-8 -\*-

5. # Every City for the Cities Manager's Cities List
6. **class** City(object):
7. **def** \_\_init\_\_(self, name, cities\_neig, lat, long):
8. self.name = name
9. self.neig = cities\_neig
10. self.lat = lat
11. self.lng = long
13. # Shows the Nearest City among all
14. **def** get\_nearest\_neig(self, visited\_cities):
15. # List Comprehension (and Lambda Expression) for Nearest City
16. nearest\_city = min({x **for** x **in** self.neig.items() **if** x[0] **not** **in** visited\_cities},
17. key=**lambda** x: x[1])
19. **return** {
20. "name": nearest\_city[0],
21. "distance": nearest\_city[1]
22. }
24. # Get a Distance between the actual City and another one
25. **def** get\_distance\_to(self, city):
26. **return** self.neig[city.get\_name()]
28. **def** get\_name(self):
29. **return** self.name
31. **def** get\_lat(self):
32. **return** self.lat
34. **def** get\_lng(self):
35. **return** self.lng

**cityToVisit.py**

1. #!/usr/bin/env python
2. # -\*- coding: utf-8 -\*-

5. # Every City of the Visited List
6. **class** CityToVisit(object):
7. **def** \_\_init\_\_(self, name, lat, long):
8. self.name = name
9. self.lat = lat
10. self.lng = long
12. **def** get\_name(self):
13. **return** self.name
15. **def** get\_lat(self):
16. **return** self.lat
18. **def** get\_lng(self):
19. **return** self.lng
21. **def** serialize(self):
22. **return** {
23. "name": self.name,
24. "lat": self.lat,
25. "lng": self.lng
26. }

**neighbourCitiesRepo.py**

1. #!/usr/bin/env python
2. # -\*- coding: utf-8 -\*-

5. # Dictionary with all the Cities and Distances between all others
6. **class** NeighbourCitiesRepo(object):
7. **def** \_\_init\_\_(self):
8. # Completar...
9. self.cities = {
10. "Buenos Aires": {"Cordoba": 646, "Corrientes": 792, "Formosa": 933, "La Plata": 53, "La Rioja": 986,
11. "Mendoza": 985, "Neuquen": 989, "Parana": 375, "Posadas": 834, "Rawson": 1127,
12. "Resistencia": 794, "Rio Gallegos": 2082, "S.F.d.V.d. Catamarca": 979,
13. "S.M. de Tucuman": 1080, "S.S. de Jujuy": 1134, "Salta": 1282, "San Juan": 1005,
14. "San Luis": 749, "Santa Fe": 393, "Santa Rosa": 579, "Sgo. del Estero": 939,
15. "Ushuaia": 2373, "Viedma": 799},
17. "Cordoba": {"Buenos Aires": 646, "Corrientes": 677, "Formosa": 824, "La Plata": 698, "La Rioja": 340,
18. "Mendoza": 466, "Neuquen": 907, "Parana": 348, "Posadas": 919, "Rawson": 1321,
19. "Resistencia": 669, "Rio Gallegos": 2281, "S.F.d.V.d. Catamarca": 362,
20. "S.M. de Tucuman": 517, "S.S. de Jujuy": 809, "Salta": 745, "San Juan": 412,
21. "San Luis": 293, "Santa Fe": 330, "Santa Rosa": 577, "Sgo. del Estero": 401,
22. "Ushuaia": 2618, "Viedma": 1047},
24. "Corrientes": {"Buenos Aires": 792, "Cordoba": 677, "Formosa": 157, "La Plata": 830, "La Rioja": 814,
25. "Mendoza": 1131, "Neuquen": 1534, "Parana": 500, "Posadas": 291, "Rawson": 1845,
26. "Resistencia": 13, "Rio Gallegos": 2819, "S.F.d.V.d. Catamarca": 691,
27. "S.M. de Tucuman": 633, "S.S. de Jujuy": 742, "Salta": 719, "San Juan": 1039,
28. "San Luis": 969, "Santa Fe": 498, "Santa Rosa": 1136, "Sgo. del Estero": 535,
29. "Ushuaia": 3131, "Viedma": 1527},
31. "Formosa": {"Buenos Aires": 933, "Cordoba": 824, "Corrientes": 157, "La Plata": 968, "La Rioja": 927,
32. "Mendoza": 1269, "Neuquen": 1690, "Parana": 656, "Posadas": 263, "Rawson": 1999,
33. "Resistencia": 161, "Rio Gallegos": 2974, "S.F.d.V.d. Catamarca": 793,
34. "S.M. de Tucuman": 703, "S.S. de Jujuy": 750, "Salta": 741, "San Juan": 1169,
35. "San Luis": 1117, "Santa Fe": 654, "Santa Rosa": 1293, "Sgo. del Estero": 629,
36. "Ushuaia": 3284, "Viedma": 1681},
38. "La Plata": {"Buenos Aires": 53, "Cordoba": 698, "Corrientes": 830, "Formosa": 968, "La Rioja": 1038,
39. "Mendoza": 1029, "Neuquen": 1005, "Parana": 427, "Posadas": 857, "Rawson": 1116,
40. "Resistencia": 883, "Rio Gallegos": 2064, "S.F.d.V.d. Catamarca": 1030,
41. "S.M. de Tucuman": 1132, "S.S. de Jujuy": 1385, "Salta": 1333, "San Juan": 1053,
42. "San Luis": 795, "Santa Fe": 444, "Santa Rosa": 602, "Sgo. del Estero": 991,
43. "Ushuaia": 2350, "Viedma": 789},
45. "La Rioja": {"Buenos Aires": 986, "Cordoba": 340, "Corrientes": 814, "Formosa": 927, "La Plata": 1038,
46. "Mendoza": 427, "Neuquen": 1063, "Parana": 659, "Posadas": 1098, "Rawson": 1548,
47. "Resistencia": 802, "Rio Gallegos": 2473, "S.F.d.V.d. Catamarca": 149,
48. "S.M. de Tucuman": 330, "S.S. de Jujuy": 600, "Salta": 533, "San Juan": 283,
49. "San Luis": 435, "Santa Fe": 640, "Santa Rosa": 834, "Sgo. del Estero": 311,
50. "Ushuaia": 2821, "Viedma": 1311},
52. "Mendoza": {"Buenos Aires": 985, "Cordoba": 466, "Corrientes": 1131, "Formosa": 1269, "La Plata": 1029,
53. "La Rioja": 427, "Neuquen": 676, "Parana": 790, "Posadas": 1384, "Rawson": 1201,
54. "Resistencia": 1121, "Rio Gallegos": 2081, "S.F.d.V.d. Catamarca": 569,
55. "S.M. de Tucuman": 756, "S.S. de Jujuy": 1023, "Salta": 957, "San Juan": 152,
56. "San Luis": 235, "Santa Fe": 775, "Santa Rosa": 586, "Sgo. del Estero": 713,
57. "Ushuaia": 2435, "Viedma": 1019},
59. "Neuquen": {"Buenos Aires": 989, "Cordoba": 907, "Corrientes": 1534, "Formosa": 1690, "La Plata": 1005,
60. "La Rioja": 1063, "Mendoza": 676, "Parana": 1053, "Posadas": 1709, "Rawson": 543,
61. "Resistencia": 1529, "Rio Gallegos": 1410, "S.F.d.V.d. Catamarca": 1182,
62. "S.M. de Tucuman": 1370, "S.S. de Jujuy": 1658, "Salta": 1591, "San Juan": 824,
63. "San Luis": 643, "Santa Fe": 1049, "Santa Rosa": 422, "Sgo. del Estero": 1286,
64. "Ushuaia": 1762, "Viedma": 479},
66. "Parana": {"Buenos Aires": 375, "Cordoba": 348, "Corrientes": 500, "Formosa": 656, "La Plata": 427,
67. "La Rioja": 659, "Mendoza": 790, "Neuquen": 1053, "Posadas": 658, "Rawson": 1345,
68. "Resistencia": 498, "Rio Gallegos": 2230, "S.F.d.V.d. Catamarca": 622,
69. "S.M. de Tucuman": 707, "S.S. de Jujuy": 659, "Salta": 906, "San Juan": 757,
70. "San Luis": 574, "Santa Fe": 19, "Santa Rosa": 642, "Sgo. del Estero": 566,
71. "Ushuaia": 2635, "Viedma": 1030},
73. "Posadas": {"Buenos Aires": 834, "Cordoba": 919, "Corrientes": 291, "Formosa": 263, "La Plata": 857,
74. "La Rioja": 1098, "Mendoza": 1384, "Neuquen": 1709, "Parana": 658, "Rawson": 1951,
75. "Resistencia": 305, "Rio Gallegos": 2914, "S.F.d.V.d. Catamarca": 980,
76. "S.M. de Tucuman": 924, "S.S. de Jujuy": 1007, "Salta": 992, "San Juan": 1306,
77. "San Luis": 1200, "Santa Fe": 664, "Santa Rosa": 1293, "Sgo. del Estero": 827,
78. "Ushuaia": 3207, "Viedma": 1624},
80. "Rawson": {"Buenos Aires": 1127, "Cordoba": 1321, "Corrientes": 1845, "Formosa": 1999, "La Plata": 1116,
81. "La Rioja": 1548, "Mendoza": 1201, "Neuquen": 543, "Parana": 1345, "Posadas": 1951,
82. "Resistencia": 1843, "Rio Gallegos": 975, "S.F.d.V.d. Catamarca": 1647,
83. "S.M. de Tucuman": 1827, "S.S. de Jujuy": 2120, "Salta": 2054, "San Juan": 1340,
84. "San Luis": 1113, "Santa Fe": 1349, "Santa Rosa": 745, "Sgo. del Estero": 1721,
85. "Ushuaia": 1300, "Viedma": 327},
87. "Resistencia": {"Buenos Aires": 794, "Cordoba": 669, "Corrientes": 13, "Formosa": 161, "La Plata": 833,
88. "La Rioja": 802,
89. "Mendoza": 1121, "Neuquen": 1529, "Parana": 498, "Posadas": 305, "Rawson": 1843,
90. "Rio Gallegos": 2818, "S.F.d.V.d. Catamarca": 678,
91. "S.M. de Tucuman": 620, "S.S. de Jujuy": 729, "Salta": 706, "San Juan": 1029,
92. "San Luis": 961, "Santa Fe": 495, "Santa Rosa": 1132, "Sgo. del Estero": 523,
93. "Ushuaia": 3130, "Viedma": 1526},
95. "Rio Gallegos": {"Buenos Aires": 2082, "Cordoba": 2281, "Corrientes": 2819, "Formosa": 2974,
96. "La Plata": 2064, "La Rioja": 2473, "Mendoza": 2081, "Neuquen": 1410, "Parana": 2320,
97. "Posadas": 2914, "Rawson": 975, "Resistencia": 2818, "S.F.d.V.d. Catamarca": 2587,
98. "S.M. de Tucuman": 2773, "S.S. de Jujuy": 3063, "Salta": 2997, "San Juan": 2231,
99. "San Luis": 2046, "Santa Fe": 2325, "Santa Rosa": 1712, "Sgo. del Estero": 2677,
100. "Ushuaia": 359, "Viedma": 1294},
102. "S.F.d.V.d. Catamarca": {"Buenos Aires": 979, "Cordoba": 362, "Corrientes": 691, "Formosa": 793,
103. "La Plata": 1030, "La Rioja": 149, "Mendoza": 569, "Neuquen": 1182, "Parana": 622,
104. "Posadas": 980, "Rawson": 1647, "Resistencia": 678, "Rio Gallegos": 2587,
105. "S.M. de Tucuman": 189, "S.S. de Jujuy": 477, "Salta": 410, "San Juan": 430,
106. "San Luis": 540, "Santa Fe": 689, "Santa Rosa": 1088, "Sgo. del Estero": 166,
107. "Ushuaia": 3116, "Viedma": 1562},
109. "S.M. de Tucuman": {"Buenos Aires": 1080, "Cordoba": 517, "Corrientes": 633, "Formosa": 703,
110. "La Plata": 1132, "La Rioja": 330, "Mendoza": 756, "Neuquen": 1370, "Parana": 707,
111. "Posadas": 924, "Rawson": 1827, "Resistencia": 620, "Rio Gallegos": 2773,
112. "S.F.d.V.d. Catamarca": 189, "S.S. de Jujuy": 293, "Salta": 228, "San Juan": 612,
113. "San Luis": 727, "Santa Fe": 689, "Santa Rosa": 1088, "Sgo. del Estero": 141,
114. "Ushuaia": 3116, "Viedma": 1562},
116. "S.S. de Jujuy": {"Buenos Aires": 1334, "Cordoba": 809, "Corrientes": 742, "Formosa": 750,
117. "La Plata": 1385, "La Rioja": 600, "Mendoza": 1023, "Neuquen": 1658, "Parana": 959,
118. "Posadas": 1007, "Rawson": 2120, "Resistencia": 729, "Rio Gallegos": 3063,
119. "S.F.d.V.d. Catamarca": 477, "S.M. de Tucuman": 293, "Salta": 67, "San Juan": 874,
120. "San Luis": 1017, "Santa Fe": 942, "Santa Rosa": 1382, "Sgo. del Estero": 414,
121. "Ushuaia": 3408, "Viedma": 1855},
123. "Salta": {"Buenos Aires": 1282, "Cordoba": 745, "Corrientes": 719, "Formosa": 741, "La Plata": 1333,
124. "La Rioja": 233, "Mendoza": 957, "Neuquen": 1591, "Parana": 906, "Posadas": 992, "Rawson": 2054,
125. "Resistencia": 706, "Rio Gallegos": 2997, "S.F.d.V.d. Catamarca": 410,
126. "S.M. de Tucuman": 228, "S.S. de Jujuy": 67, "San Juan": 808,
127. "San Luis": 950, "Santa Fe": 889, "Santa Rosa": 1316, "Sgo. del Estero": 353,
128. "Ushuaia": 3341, "Viedma": 1790},
130. "San Juan": {"Buenos Aires": 1005, "Cordoba": 412, "Corrientes": 1039, "Formosa": 1169, "La Plata": 1053,
131. "La Rioja": 283, "Mendoza": 152, "Neuquen": 824, "Parana": 757, "Posadas": 1306, "Rawson": 1340,
132. "Resistencia": 1029, "Rio Gallegos": 2231, "S.F.d.V.d. Catamarca": 430,
133. "S.M. de Tucuman": 612, "S.S. de Jujuy": 874, "Salta": 808,
134. "San Luis": 284, "Santa Fe": 740, "Santa Rosa": 686, "Sgo. del Estero": 583,
135. "Ushuaia": 2585, "Viedma": 1141},
137. "San Luis": {"Buenos Aires": 749, "Cordoba": 293, "Corrientes": 969, "Formosa": 1117, "La Plata": 795,
138. "La Rioja": 435, "Mendoza": 235, "Neuquen": 643, "Parana": 574, "Posadas": 1200, "Rawson": 1113,
139. "Resistencia": 961, "Rio Gallegos": 2046, "S.F.d.V.d. Catamarca": 540,
140. "S.M. de Tucuman": 727, "S.S. de Jujuy": 1017, "Salta": 950, "San Juan": 284,
141. "Santa Fe": 560, "Santa Rosa": 412, "Sgo. del Estero": 643, "Ushuaia": 2392, "Viedma": 882},
143. "Santa Fe": {"Buenos Aires": 393, "Cordoba": 330, "Corrientes": 498, "Formosa": 654, "La Plata": 444,
144. "La Rioja": 640,
145. "Mendoza": 775, "Neuquen": 1049, "Parana": 19, "Posadas": 664, "Rawson": 1349,
146. "Resistencia": 495, "Rio Gallegos": 2325, "S.F.d.V.d. Catamarca": 602,
147. "S.M. de Tucuman": 689, "S.S. de Jujuy": 942, "Salta": 889, "San Juan": 740,
148. "San Luis": 560, "Santa Rosa": 641, "Sgo. del Estero": 547,
149. "Ushuaia": 2641, "Viedma": 1035},
151. "Santa Rosa": {"Buenos Aires": 579, "Cordoba": 577, "Corrientes": 1136, "Formosa": 1293, "La Plata": 602,
152. "La Rioja": 834,
153. "Mendoza": 586, "Neuquen": 422, "Parana": 642, "Posadas": 1293, "Rawson": 745,
154. "Resistencia": 1132, "Rio Gallegos": 1712, "S.F.d.V.d. Catamarca": 915,
155. "S.M. de Tucuman": 1088, "S.S. de Jujuy": 1382, "Salta": 1316, "San Juan": 686,
156. "San Luis": 412, "Santa Fe": 641, "Sgo. del Estero": 977,
157. "Ushuaia": 2044, "Viedma": 477},
159. "Sgo. del Estero": {"Buenos Aires": 979, "Cordoba": 401, "Corrientes": 535, "Formosa": 629, "La Plata": 991,
160. "La Rioja": 311,
161. "Mendoza": 713, "Neuquen": 1286, "Parana": 566, "Posadas": 827, "Rawson": 1721,
162. "Resistencia": 523, "Rio Gallegos": 2677, "S.F.d.V.d. Catamarca": 166,
163. "S.M. de Tucuman": 141, "S.S. de Jujuy": 414, "Salta": 353, "San Juan": 583,
164. "San Luis": 643, "Santa Fe": 547, "Santa Rosa": 977,
165. "Ushuaia": 3016, "Viedma": 1446},
167. "Ushuaia": {"Buenos Aires": 2373, "Cordoba": 2618, "Corrientes": 3131, "Formosa": 3284, "La Plata": 2350,
168. "La Rioja": 2821, "Mendoza": 2435, "Neuquen": 1762, "Parana": 2635, "Posadas": 3207,
169. "Rawson": 1300, "Resistencia": 3130, "Rio Gallegos": 359, "S.F.d.V.d. Catamarca": 2931,
170. "S.M. de Tucuman": 3116, "S.S. de Jujuy": 3408, "Salta": 3341, "San Juan": 2585,
171. "San Luis": 2392, "Santa Fe": 2641, "Santa Rosa": 2044, "Sgo. del Estero": 3016,
172. "Viedma": 1605},
174. "Viedma": {"Buenos Aires": 799, "Cordoba": 1047, "Corrientes": 1527, "Formosa": 1681, "La Plata": 789,
175. "La Rioja": 1311, "Mendoza": 1019, "Neuquen": 479, "Parana": 1030, "Posadas": 1624, "Rawson": 327,
176. "Resistencia": 1526, "Rio Gallegos": 1294, "S.F.d.V.d. Catamarca": 1391,
177. "S.M. de Tucuman": 1562, "S.S. de Jujuy": 1855, "Salta": 1790, "San Juan": 1141,
178. "San Luis": 882, "Santa Fe": 1035, "Santa Rosa": 477, "Sgo. del Estero": 1446,
179. "Ushuaia": 1605},
181. # More 16 Cities like "Santa Fe": {"Buenos Aires": 4, "La Plata": 5, "Cordoba": 13},
182. }
184. # Get one City element of the Dictionary
185. **def** get\_near\_cities\_dict(self, city):
186. **return** self.cities[city]

**chormosome.py**

1. #!/usr/bin/env python
2. # -\*- coding: utf-8 -\*-
4. **import** random

7. **class** Chromosome(object):
9. # Class Attribute
10. CitiesDict = {}
12. # Constructor / Instance Attributes
13. **def** \_\_init\_\_(self, large, cities, newRoute):
14. # Chromosome's Genes
15. **if** newRoute **is** None:
16. random.shuffle(cities)
17. self.route = []
18. **for** i **in** range(len(cities)):
19. self.route.append(cities[i])  # Shuffle and then Assignation
20. **print**(self.route)
21. **else**:
22. self.route = newRoute
23. self.large = large
24. self.objectivePunctuation = 0
25. self.fitness = 0
27. # Initialize Objective Function Punctuation
28. self.setObjectivePunctuation()

31. @classmethod
32. **def** getCitiesDict(cls):
33. **return** cls.CitiesDict
35. @classmethod
36. **def** setCitiesDict(cls, citiesDictionary):
37. cls.CitiesDict = citiesDictionary
39. # Show All Genes of the Chromosome
40. **def** getRoute(self):
41. **return** self.route
43. **def** getAccumulatedDistance(self):
44. accumulated\_distance = 0
45. cd = self.getCitiesDict()
46. city\_last = self.route[0]
47. # Calculate Distance Step by Step comparing the Chromosome's Route with Neighbours on Cities Dictionary
48. **for** i **in** range(1, len(self.route) - 2):
49. city\_step = self.route[i]
50. accumulated\_distance += cd[city\_last].get\_distance\_to(cd[city\_step])
51. city\_last = city\_step
52. # At the End, go back to the First City
53. accumulated\_distance += cd[self.route[len(self.route) - 1]].get\_distance\_to(cd[self.route[0]])
54. **return** accumulated\_distance
56. **def** calcObjPunc(self):
57. **return** 1 / self.getAccumulatedDistance()
59. **def** calcFitness(self, totalObj):  # Reverse Score (Little Distances gets better Fitness)
60. # self.fitness = 1000 / self.getObjectivePunctuation()
61. self.fitness = (self.getObjectivePunctuation() / totalObj)  # Update Fitness
62. **return** self.fitness
64. # Getters and Setters
65. **def** getLarge(self):
66. **return** self.large
68. **def** setLarge(self, large):
69. self.large = large
71. **def** getObjectivePunctuation(self):
72. **return** self.objectivePunctuation
74. **def** setObjectivePunctuation(self):
75. self.objectivePunctuation = self.calcObjPunc()
77. **def** getFitness(self):
78. **return** self.fitness
80. **def** setFintess(self, fitness):
81. self.fitness = fitness
83. """
84. def copy(self, another\_crom, start, end):
85. for i in range(start, end):
86. self.route[i] = (another\_crom.route[i])
87. """
89. **def** mutate(self):
90. swapPos1 = random.randint(1, self.getLarge() - 1)
91. swapPos2 = random.randint(1, self.getLarge() - 1)
92. **while** swapPos1 == swapPos2:
93. swapPos2 = random.randint(1, self.getLarge() -1)
94. newRoute = []
95. **for** i **in** range(self.getLarge()):
96. **if** i != swapPos1 **and** i != swapPos2:
97. newRoute.append(self.route[i])
98. **elif** i == swapPos1:
99. newRoute.append(self.route[swapPos2])
100. **elif** i == swapPos2:
101. newRoute.append(self.route[swapPos1])
102. **print**("Mutated Chrom in positions:", swapPos1, "and", swapPos2, ": ", self.route)

**population.py**

1. #!/usr/bin/env python
2. # -\*- coding: utf-8 -\*-
4. **from** chromosome **import** Chromosome
5. **import** random

8. **class** Population(object):
9. # Constructor / Instance Attributes
10. **def** \_\_init\_\_(self, numChroms, cities, crossProb, mutProb):
11. self.population = []  # Initial Population (Array of Chromosomes)
12. self.totalObjPunc = 0  # The Sum of All Objective Functions Punctuation
13. self.totalFitness = 0  # The Sum of All Objective Values
14. self.numChroms = numChroms
15. self.chromSize = len(cities)
16. self.crossProb = crossProb
17. self.mutProb = mutProb
18. self.bestTrack = None
19. self.bestTrackDistance = 1000000
21. **print**("Start Algorithm")
22. **for** i **in** range(numChroms):
23. **print**("Chrom ", i, end=": ")
24. # Initialization of Chromosomes
25. self.population.append(Chromosome(self.chromSize, cities, None))  # Add to Population
27. # Show Actual Population and Stats
28. **def** showPopulation(self, numIter):
29. self.setTotalFitness(0)
30. self.setTotalObjPunc(self.calcTotalObjPunc())
31. large = self.getChromSize()
32. averageObjPunc = self.getTotalObjPunc() / len(self.population)
33. fitness = 0
34. maxVal = 0
35. secondMaxVal = 0
36. minVal = 0
37. bestRoutePos = 0
38. secondBestRoutePos = 0
39. worstRoutePos = 0
40. **print**()
41. **print**()
42. **print**("Population ", (numIter + 1), ":")
43. **for** i **in** range(len(self.population)):
44. fitness = self.population[i].calcFitness(self.getTotalObjPunc())
45. self.updateTotalFitness(fitness)
46. **if** i == 0:
47. maxVal = fitness
48. minVal = fitness
49. **elif** fitness > maxVal:
50. maxVal = fitness
51. bestRoutePos = i
52. **elif** fitness < minVal:
53. minVal = fitness
54. worstRoutePos = i
55. **elif** (fitness > secondMaxVal) **and** (fitness < maxVal):
56. secondMaxVal = fitness
57. secondBestRoutePos = i
58. # for j in range(large):
59. # print(self.population[i].getRoute()[j], end=', ')
60. **print**(self.population[i].getRoute())
61. fitness = self.getTotalFitnessAverage()
63. # Setting Current Generation's Best Values (If Improves Actual Best)
64. **if** self.population[bestRoutePos].getAccumulatedDistance() < self.getBestTrackDistance():
65. self.setBestTrack(self.population[bestRoutePos])
66. self.setBestTrackDistance(self.population[bestRoutePos].getAccumulatedDistance())
68. **print**()
69. **print**("Generation", numIter+1, "Final Status:")
70. **print**("Worst Value: Route Nº", worstRoutePos, "with:",
71. self.population[worstRoutePos].getObjectivePunctuation(), "OP,", round(minVal, 4), "Fit")
72. **print**("Best Value: Route Nº", bestRoutePos, "with:",
73. self.population[bestRoutePos].getObjectivePunctuation(), "OP,", round(maxVal, 4), "Fit")
74. **print**("Total Distance of Best Route:", self.getBestTrackDistance(), "km")
75. **print**("Average OP:", averageObjPunc, "--- Average Fitness:", round(self.getTotalFitnessAverage(), 6))
76. **print**("---")
77. **print**()
79. elitChrom = self.population[bestRoutePos]
80. secondElitChrom = self.population[secondBestRoutePos]
82. **return** {
83. "AverageOP": averageObjPunc,
84. "MinOP": self.population[worstRoutePos].getObjectivePunctuation(),
85. "MaxOP": self.population[bestRoutePos].getObjectivePunctuation(),
86. "ElitChrom": elitChrom,
87. "SecondElitChrom": secondElitChrom
88. }
90. # Get the Best Values
91. **def** getBestTrackAg(self):
92. **return** self.getBestTrack(), self.getBestTrackDistance()
94. # Calculate Total of Objective Functions Punctuation in the actual Generation
95. **def** calcTotalObjPunc(self):
96. acumObjPunc = 0
97. **for** chromosome **in** self.population:
98. acumObjPunc += chromosome.getObjectivePunctuation()  # Add Every Objective Function Punctuation
99. **return** acumObjPunc
101. # Update Total Fitness
102. **def** updateTotalFitness(self, fitness):
103. self.totalFitness += fitness
105. **def** getBestTrack(self):
106. **return** self.bestTrack
108. **def** setBestTrack(self, track):
109. self.bestTrack = track
111. **def** getBestTrackDistance(self):
112. **return** self.bestTrackDistance
114. **def** setBestTrackDistance(self, trackDistance):
115. self.bestTrackDistance = trackDistance
117. # Add to Population
118. **def** addChrom(self, Chrom):
119. self.population.append(Chrom)
121. # Reproduction
122. **def** reproduce(self, elitChrom, secondElitChrom):
123. parents = []  # List of Potential Parents
124. newGeneration = []  # List of Children
126. # Elitism
127. parents.append(elitChrom)
128. parents.append(secondElitChrom)
130. **print**("Roulette Results: ", end='')
131. **for** \_ **in** range(len(self.population)):
132. indexSelectedParent = self.roulette()  # Parents Selected by Roulette
133. **if** indexSelectedParent **is** **not** None:
134. parents.append(self.population[indexSelectedParent])
135. **else**:
136. parents.append(self.population[0])  # Elite Chromosome
137. **print**(0, end=', ')
138. **print**()
139. **for** i **in** range(2, len(parents), 2):
140. father1 = parents[i]
141. father2 = parents[i + 1]
142. **if** self.crossPosibility():  # CrossOver Probability Evaluation
143. son1, son2 = self.cross(father1, father2)  # CrossOver
144. **print**("Successful CrossOver in reproduction:", (i + 2) / 2)  # Only Print
145. **else**:
146. son1, son2 = self.copy(father1, father2)  # Direct Assignation (Without CrossOver)
147. **print**("CrossOver didn't happen in reproduction:", (i + 2) / 2)  # Only Print
148. # Individual Mutation Probability Evaluation
149. **if** self.mutationPosibility():
150. son1.mutate()
151. **if** self.mutationPosibility():
152. son2.mutate()
154. son1.setObjectivePunctuation()
155. son2.setObjectivePunctuation()
157. self.addChildren(son1, son2, newGeneration)
158. self.replacePopulation(newGeneration)
159. self.setTotalFitness(0)
161. # Genetic Operator (Roulette Method)
162. **def** roulette(self):
163. # Generator of a Bidimensional List (Fitness Range of Chromosomes)
164. newRoulette = [[0] \* 2 **for** \_ **in** range(len(self.population))]
165. acum = 0  # Acumulator of Relative Fitness from 0 to 1 (Fills Roulette)
166. **for** i **in** range(len(self.population)):
167. newRoulette[i][0] = acum  # Range Min: Last Acum Value
168. acum += self.population[i].getFitness()  # Acum's Value From Zero
169. newRoulette[i][1] = acum  # Range Max: New Acum Value
170. ranNum = random.uniform(0, 1)  # Random Number from 0.000000 to 0.999999
171. **for** i **in** range(len(newRoulette)):
172. **if** newRoulette[i][0] <= ranNum <= newRoulette[i][1]:
173. # Return Selected Chromosome if the Random Number Exists in its Range
174. **print**(i, end=', ')
175. **return** i
176. **return** None  # Error Return
178. **def** crossPosibility(self):  # CrossOver posibility evaluation
179. **if** self.getCrossProb()\*100 >= random.randint(1, 100):
180. **return** True
181. **else**:
182. **return** False
184. # Cyclic Crossover
185. **def** cross(self, parent1, parent2):
186. crom\_size = parent1.getLarge()
188. newRoute1 = ['']\*crom\_size
189. newRoute2 = ['']\*crom\_size
191. newRoute1[0] = parent1.route[0]
192. aux = parent2.route[0]
193. pos = None
194. **while** aux != parent1.route[0]:
195. **for** i **in** range(crom\_size):
196. **if** parent1.route[i] == aux:
197. pos = i
198. **break**
199. newRoute1[pos] = aux
200. aux = parent2.route[pos]
202. **for** i **in** range(crom\_size):
203. **if** newRoute1[i] == '':
204. newRoute1[i] = parent2.route[i]
206. newRoute2[0] = parent2.route[0]
207. aux = parent1.route[0]
208. pos = None
209. **while** aux != parent2.route[0]:
210. **for** i **in** range(crom\_size):
211. **if** parent2.route[i] == aux:
212. pos = i
213. **break**
214. newRoute2[pos] = aux
215. aux = parent1.route[pos]
217. **for** i **in** range(crom\_size):
218. **if** newRoute2[i] == '':
219. newRoute2[i] = parent1.route[i]
221. son1 = Chromosome(crom\_size, None, newRoute1)
222. son2 = Chromosome(crom\_size, None, newRoute2)
224. **print**()
225. **print**("New Sons after CrossOver: ")
226. **print**(son1.getRoute())
227. **print**(son2.getRoute())
229. **return** son1, son2
231. **def** copy(self, chrom1, chrom2):
232. crom\_size = chrom1.getLarge()
233. newRoute1 = chrom1.getRoute()
234. newRoute2 = chrom2.getRoute()
235. son1 = Chromosome(crom\_size, None, newRoute1)
236. son2 = Chromosome(crom\_size, None, newRoute2)
238. **print**()
239. **print**("New Sons without CrossOver (Indentical): ")
240. **print**(son1.getRoute())
241. **print**(son2.getRoute())
243. **return** son1, son2
245. **def** mutationPosibility(self):  # Mutation posibility evaluation
246. **if** self.getMutProb() \* 100 >= random.randint(1, 100):
247. **return** True
248. **else**:
249. **return** False
251. **def** addChildren(self, son1, son2, newGeneration):
252. newGeneration.append(son1)
253. newGeneration.append(son2)
255. **def** replacePopulation(self, newGeneration):  # Replace All Population in every Iteration
256. self.population = []
257. **for** i **in** range(len(newGeneration)):
258. self.population.append(newGeneration[i])
260. **def** getTotalObjPunc(self):
261. **return** self.totalObjPunc
263. **def** setTotalObjPunc(self, total):
264. self.totalObjPunc = total
266. **def** getTotalFitness(self):
267. **return** self.totalFitness
269. **def** setTotalFitness(self, total):
270. self.totalFitness = total
272. **def** getTotalFitnessAverage(self):
273. **return** self.totalFitness / len(self.population)
275. **def** getNumChroms(self):
276. **return** self.numChroms
278. **def** setNumChroms(self, numChroms):
279. self.numChroms = numChroms
281. **def** getChromSize(self):
282. **return** self.chromSize
284. **def** setChromSize(self, chromSize):
285. self.chromSize = chromSize
287. **def** getCrossProb(self):
288. **return** self.crossProb
290. **def** setCrossProb(self, crossProb):
291. self.crossProb = crossProb
293. **def** getMutProb(self):
294. **return** self.mutProb
296. **def** setMutProb(self, mutProb):
297. self.mutProb = mutProb

## FRONTEND

**index.html**

1. <!DOCTYPE html**>**
2. **<html** lang="en"**>**
3. **<head>**
4. **<meta** charset="UTF-8"**>**
5. **<meta** name="viewport" content="width=device-width, initial-scale=1.0"**>**
6. **<meta** http-equiv="X-UA-Compatible" content="ie=edge"**>**
7. **<link** rel="stylesheet" type="text/css" href="../static/css/styles.css"**>**
8. **<title>**Traveling Salesman Problem**</title>**
9. **</head>**
10. **<body** style="background-image:url(../static/resources/artificial-intelligence.jpg)" onload='initialization()'**>**
11. **<header>**
12. <!-- <body style="background-image:url({{url\_for('static', filename='artificial-intelligence.jpg')}})"> -->
13. **<div** class="title"**><h1>**Genetic Algorithms TP3: Traveling Salesman Problem**</h1></div>**
14. **</header>**
15. **<main>**
16. **<ul** id="options"**>**
17. **<li><a** href="#" id="opt1" class="menuLink"**>**
18. EXERCISE 2-A
19. **<br/>**
20. Best Selected City's Route by Heuristic
21. **</a></li>**
22. **<li><a** href="http://127.0.0.1:5000/Map" id="opt2" class="menuLink"**>**
23. EXERCISE 2-B
24. **<br/>**
25. Best All Cities' Route by Heuristic
26. **</a></li>**
27. **<li><a** href="http://127.0.0.1:5000/Map/Ag" id="opt3" class="menuLink"**>**
28. EXERCISE 2-C
29. **<br/>**
30. Best All Cities' Route by Genetic Algorithm
31. **</a></li>**
32. **</ul>**
33. **</main>**
34. **<footer>**
35. **<p>**MADE BY: Antonelli, Recalde, Rohn**</p>**
36. **<br/>**
37. **<p>**Genetics Algorithms 2019 - UTN FRRo**</p>**
38. **</footer>**
39. **<div** id="overlayModal" class="modal"**>**
40. **<div** class="modal-content"**>**
41. **<div** class="modal-header"**>**
42. **<span** class="close"**>**×**</span>**
43. **<h2>**Select One City**</h2>**
44. **</div>**
45. **<div** class="modal-body"**>**
46. **<input** type="text" id="cityName" name="cityName" placeholder="City Name"**>**
47. **<button** id="citySend"**>**Run!**</button>**
48. **</div>**
49. **</div>**
50. **</div>**
51. **<div** id="agModal" class="modal"**>**
52. **<div** class="modal-content"**>**
53. **<div** class="modal-header"**>**
54. **<h2>**Genetic Algorithm**</h2>**
55. **</div>**
56. **<div** class="modal-body"**>**
57. **<p>**Training Genetic Algorithm Model, please wait...**</p>**
58. **</div>**
59. **</div>**
60. **</div>**
61. **<script** src="../static/js/functions.js" type="text/javascript"**></script>**
62. **</body>**
63. **</html>**

**showMap.html**

1. <!DOCTYPE html**>**
2. **<html** lang="en"**>**
3. **<head>**
4. **<meta** charset="UTF-8"**>**
5. **<meta** name="viewport" content="width=device-width, initial-scale=1.0"**>**
6. **<meta** http-equiv="X-UA-Compatible" content="ie=edge"**>**
7. **<title>**TSP: Map**</title>**
8. **<link** rel="stylesheet" href="../static/css/styles.css"**>**
9. **<script** src="https://maps.googleapis.com/maps/api/js?key=AIzaSyBGsPRU6-v05wBr\_dfzUj-AkQn3HvpCezk&callback=initMap"
10. defer**></script>**
11. **<script** src="../static/js/mapConfiguration.js" type="text/javascript"**></script>**
12. **<script** type="text/javascript"**>**
13. document.addEventListener('DOMContentLoaded', function () {
14. setFlightPath({{ track|tojson }});
15. setMarkers({{ track|tojson }});
16. });
17. **</script>**
18. **</head>**
19. **<body** onload='logTrack({{ track|tojson }})'**>**
20. **<div** id="map-container"**>**
21. **<div** id="map"**></div>**
22. **<div** id="sideStats"**></div>**
23. **</div>**
24. **<script** src="../static/js/functions.js" type="text/javascript"**></script>**
25. **</body>**
26. **</html>**

**styles.css**

1. /\* ############ PRINCIPAL STYLES ############ \*/
2. \* {
3. **margin**: 0;
4. **padding**: 0;
5. }
7. body {
8. **background-color**: darkred;
9. /\* background-image: url("route/img"); \*/
10. **background-repeat**: no-repeat;
11. background-**size**: cover;
12. }
14. body, html{
15. **height**: 100vh;
16. }
18. header { **height**: 15vh; }
20. main { **height**: 75vh; }
22. footer {
23. **height**: 10vh;
24. **color**: white;
25. **background-color**: black;
26. **padding**: 1rem;
27. **text-align**: center;
28. }
30. .title {
31. **margin**: 2rem;
32. **display**: flex;
33. flex-**direction**: row;
34. justify-**content**: center;
35. **color**: white;
36. **text-shadow**: -2px 0 black, 0 2px black, 2px 0 black, 0 -2px black;
37. }
39. .title>h1 {
40. **padding**: 1rem;
41. **background-color**: saddlebrown;
42. **border**: 0.25rem dashed cornsilk;
43. }
45. #options {
46. **list-style**: none;
47. **display**: flex;
48. flex-**direction**: column;
49. **text-align**: center;
50. align-items: center;
51. /\* justify-content: center; \*/
52. }
54. #options>li {
55. **margin**: 1rem;
56. **font-size**: 150%;
57. **display**: flex;
58. }
60. #options>li>a {
61. **padding**: 1.5rem 15vw 1.5rem 15vw;
62. **border**: 0.25rem dashed cornsilk;
63. **background-color**: slateblue;
64. }
66. /\* All Links \*/
67. a { **text-decoration**: none; }
69. /\* Unvisited Links \*/
70. .menuLink:link { **color**: black; }
71. .whiteLink:link { **color**: white; }
73. /\* Visited Links \*/
74. .menuLink:visited { **color**: black; }
75. .whiteLink:visited { **color**: white; }
77. /\* MouseOver Links \*/
78. .menuLink:hover { **color**: cornflowerblue; }
79. .whiteLink:hover { **color**: cornflowerblue; }
81. /\* Selected Links \*/
82. .menuLink:active { **color**: cornflowerblue; }
83. .whiteLink:active { **color**: cornflowerblue; }
85. /\* ############ SHOWMAP STYLES ############ \*/
87. #map-container {
88. **display**: flex;
89. flex-**direction**: row;
90. **width**: 100vw;
91. **height**: 100vh;
92. }
94. #map {
95. /\* flex: 1 1 auto; \*/
96. **width**: 65vw;
97. **padding-bottom**: 10px;
98. }
100. #sideStats {
101. /\* flex: 1 1 auto; \*/
102. **color**: white;
103. **background-color**: black;
104. **width**: 35vw;
105. **padding-top**: 10px;
106. **padding-bottom**: 10px;
107. **text-align**: center;
108. **overflow**: scroll;
109. }
111. /\* ############ MODAL STYLES ############ \*/
113. /\* Modal Background \*/
114. .modal {
115. **display**: none; /\* Hidden by default \*/
116. **position**: fixed;
117. **z-index**: 10;
118. left: 0;
119. top: 0;
120. **width**: 100%;
121. **height**: 100%;
122. **overflow**: auto;
123. **background-color**: rgb(0,0,0);
124. **background-color**: rgba(0,0,0,0.4); /\* Black w/opacity \*/
125. }
127. /\* Modal Box \*/
128. .modal-content {
129. **background-color**: #FEFEFE;
130. **margin**: 15% auto;
131. **padding**: 0;
132. **border**: 1px solid #888;
133. **width**: 50%;
134. }
136. /\* Modal Header \*/
137. .modal-header {
138. **padding**: 2px 16px;
139. **background-color**: darkslategrey;
140. **color**: white;
141. }
143. /\* Modal Body \*/
144. .modal-body { **padding**: 20px 16px; }
146. /\* Modal Footer \*/
147. /\*.modal-footer {
148. padding: 2px 16px;
149. background-color: #5cb85c;
150. color: white;
151. }\*/
153. /\* Closing Styles \*/
154. .close {
155. **color**: #aaa;
156. **float**: right;
157. **font-size**: 28px;
158. **font-weight**: bold;
159. }
161. .close:hover, .close:focus {
162. **color**: black;
163. **text-decoration**: none;
164. **cursor**: pointer;
165. }

**functions.js**

1. // Stats of the Track (Map Events)
2. **function** logTrack(track) {
3. console.log("Stats Loading...");
4. document.getElementById("sideStats").innerHTML = "<p>" + "BEST ROUTE STATS:" + "</p>" + "<br />";
5. document.getElementById("sideStats").innerHTML += "<p style='padding-bottom: 10px'>" +
6. "Total Distance: " +  track["accumulated\_distance"] + " km" + "</p>" + "<hr />" + "<br />";
7. track["cities\_to\_visit"].forEach(**function** (item) {
8. document.getElementById("sideStats").innerHTML += "<p>" + item.name + ": " +
9. item.lat + " | " + item.lng + "</p>" + "<br />";
10. console.log(item.name, item.lat, item.lng);
11. });
12. document.getElementById("sideStats").innerHTML += "<hr />";
13. document.getElementById("sideStats").innerHTML += "<div id='goBack' style='padding-top: 10px'>" +
14. "<a href='#' class='whiteLink'>" + "\u21a9 Back to Index" + "</a>" + "</div>" + "<br />";
15. console.log("Total Distance: " + track["accumulated\_distance"] + " km");
17. // "Go Back" Button
18. let backBtn = document.getElementById("goBack");
20. // URL Root
21. let urlRoot = "http://127.0.0.1:5000/";
23. // Back to Index (Doesn't Work, Bubbling problem!)
24. backBtn.addEventListener('click', **function** (e) {
25. // backBtn.onclick = function() { window.location.href = urlRoot; };
26. e.preventDefault();
27. window.location.href = urlRoot;
28. }, **true**);
30. console.log("Fully Loaded!");
31. }
33. // Index Events
34. **function** initialization() {
35. // Get the Modals
36. let overlayModal = document.getElementById("overlayModal");
37. let agModal = document.getElementById("agModal");
39. // Get the Button that opens the Modal in both Modals
40. let btn = document.getElementById("opt1");
41. let btnAg = document.getElementById("opt3");
43. // Get the <span> element that closes the Modal in OverlayModal
44. let closeSpan = document.getElementsByClassName("close")[0];
46. // When the user clicks on the Button, open the Modal
47. btn.onclick = **function**() {
48. overlayModal.style.display = "block";
49. };
51. btnAg.onclick = **function**() {
52. agModal.style.display = "block";
53. };
55. // When the user clicks on <span> (x), close the Modal
56. closeSpan.onclick = **function**() {
57. overlayModal.style.display = "none";
58. };
60. // When the user clicks anywhere outside of the Modal, close it
61. window.onclick = **function**(event) {
62. **if** (event.target === overlayModal) {
63. overlayModal.style.display = "none";
64. }
65. };
67. // Root URL
68. let urlRoot = "http://127.0.0.1:5000/";
70. // URI with Map Endpoint
71. let uriMap = urlRoot + "Map/";
73. // Run! inside-modal button
74. let runBtn = document.getElementById("citySend");
76. // Reload Page (Exercise 2-A)
77. runBtn.onclick = **function**() {
78. // City name
79. let cityName = document.getElementById("cityName").value;
80. window.location.href = uriMap + cityName;
81. };
82. }

**configurations.js**

1. **function** initMap() {
2. // Central reference location, UTN FRRo
3. baseLocation = {lat: -41.0000000, lng: -64.0000000};
4. // Map, hiding business and other unnecessary stuff
5. map = **new** google.maps.Map(
6. document.getElementById('map'), {
7. center: baseLocation
8. ,zoom: 4.25
9. ,styles: [{"featureType": "poi.business","stylers": [{ "visibility": "off" }]}]
10. ,mapTypeControl: **false**
11. ,fullscreenControl: **false**
12. ,streetViewControl: **false**
13. });
15. locationsToAdd = [];
17. // To draw the track
18. // https://stackoverflow.com/questions/31305497/how-to-draw-an-arrow-on-every-polyline-segment-on-google-maps-v3
19. flightPath = **new** google.maps.Polyline({
20. path: locationsToAdd
21. ,geodesic: **true**
22. ,strokeColor: '#FF000'
23. ,strokeOpacity: 0.6
24. ,strokeWeight: 4
25. ,icons: [{
26. icon: {path: google.maps.SymbolPath.FORWARD\_CLOSED\_ARROW}
27. ,offset: '100%'
28. ,repeat: '200px'
29. }]
30. });
32. flightPath.setMap(map);
34. // Center the map taking into account every position
35. bounds = **new** google.maps.LatLngBounds();
37. markers = [];
38. }
40. **function** setFlightPath(track) {
41. track["cities\_to\_visit"].forEach(**function** (location) {
42. let locationToAdd = {
43. lat: parseFloat(location.lat)
44. ,lng: parseFloat(location.lng)
45. };
47. console.log(locationToAdd);
49. locationsToAdd.push(locationToAdd);
50. });
52. locationsToAdd.forEach(**function** (location) {
53. console.log(location.lat, location.lng);
54. });
56. flightPath.setPath(locationsToAdd);
58. console.log(flightPath);
59. }
61. **function** setMarkers(track) {
62. let locations = track["cities\_to\_visit"];
64. let location = locations[locations.length - 1];
66. console.log(location);
68. let locationToAdd = {
69. lat: parseFloat(location.lat)
70. ,lng: parseFloat(location.lng)
71. };
73. newMarker(locationToAdd);
74. }
76. **function** newMarker(locationToAdd) {
77. let marker = **new** google.maps.Marker({
78. position: locationToAdd
79. ,map: map
80. });
82. markers.push(marker);
83. }

**LENGUAJES UTILIZADOS Y OTRAS TECNOLOGÍAS:**

* ***BACKEND: Python, Flask (Framework), API Google Maps***
* ***FRONTEND: HTML5, CSS3, JavaScript***

**REPOSITORIO GITHUB:**

[https://github.com/utn-frro-geneticos19-g16/](https://github.com/utn-frro-geneticos19-g16/Genetic-Algorithms-TP2)

# Explicacion de Funcionamiento

***Ejercicio 1: Búsqueda Exhaustiva***

Podemos decir que la búsqueda de la ruta de distancia mínima que logra unir a todas las capitales de la republica argentina utilizando el método exhaustivo es prácticamente imposible ya que llevaría miles de años su resolución por ser el espacio de soluciones tan grande (24^24).

***Ejercicio 2: Método Heurístico, e implementación de Algoritmo Genético***

Para resolver los ejercicios 2a y 2b, implementamos la heurística de “Visitar la ciudad siguiente no-visitada más cercana a la ciudad actual”, y luego volver al inicio. En el inciso 2a se pide que dada una ciudad, se obtenga el mejor recorrido posible siguiendo esta heurística, y en el 2b: realizar lo anterior a cada provincia, y quedarse con el mejor recorrido posible, que en este caso llegamos a la conclusión de que es: partir desde la ciudad de **Neuquén**.

Más abajo explicaremos donde se encuentra y que hacen los métodos que componen los archivos de nuestra solución, pero en síntesis tenemos objetos ciudades donde cada una tiene un atributo que es un diccionario con la distancia respecto a todas las otras ciudades. Lo que hacemos es crear una lista de “Ciudades a Visitar” que son también objetos, pero solo el nombre de la ciudad, su latitud y su longitud real (para utilizar en el Mapa de Google Maps luego). Vamos en cada momento agregando la ciudad actual a esta lista, y preguntando cuál es la siguiente más cercana (corroborando no repetir las que ya están en la lista). Por su puesto, acumulando la distancia parcial entre ciudad y ciudad. En el final, le enviamos la lista de ciudades a visitar con latitudes y longitudes, y la distancia total recorrida, a nuestro mapa para que grafique el recorrido.

Para el Algoritmo Genético que se pide en el 2c, lo que le enviamos al mapa es lo mismo: la mejor ruta obtenida con nombres para exhibir a un lado y coordenadas para graficar el recorrido, y la distancia total. Los parámetros iniciales (tamaño de población, iteraciones, probabilidad de mutación y crossover) se encuentran en “citiesManager.py”, y todos los elementos que componen al algoritmo genético los encontraremos en “population.py” y “chromosome.py” (mostrar población actual, cálculo de función objetivo, cálculo de Fitness, reproducción, ruleta, crossover, mutación, etc…). Abajo, hay más detalles sobre estos métodos.

***Explicación del Funcionamiento de los métodos y archivos de ejercicios 2a, 2b, 2c***

**app.py**

* Aquí se da la inicialización de nuestra aplicación principal junto con las ciudades.
* También tenemos la lógica del web server donde cada URL se le asigna una funcionalidad con una entrada del Query String y un Template renderizado.

**city.py**

* Contiene la clase **City** con sus respectivas funciones que devuelven los datos de la clase.
* La función **get\_nearest\_neig** recibe un listado de las ciudades actualmente visitadas y compara con el diccionario de ciudades vecinas para encontrar la capital más próxima descartando las que ya han sido visitadas.
* La función **get\_distance\_to** devuelve la distancia de la ciudad actual a otra.

**cityToVisit.py**

* Contiene a la clase **CityToVisit** donde tenemos solamente la información necesaria para mostrar.

**neighbourCitiesRepo.py**

* Contiene a la clase llamada **NeighbourCitiesRepo** la cual básicamente es un repositorio donde se encuentran definidas, en un diccionario, todas las capitales donde cada una contiene los nombres y distancias respectivas a todas las otras capitales de la República Argentina.
* Esta clase además tiene la función **get\_near\_cities\_dict** que al ser llamada devuelve la ciudad indicada.

**citiesManager.py**

* Contiene la clase **CitiesManager** donde se inicializan todas las ciudades instanciando cada una con el objeto City y agregándolas a un nuevo diccionario.
* La función **get\_best\_track\_from** busca y devuelve el mejor recorrido que une a todas las capitales a partir de una ciudad dada, para ello inicia el recorrido con la capital raíz dada y a medida que avanza hacia las ciudades vecinas va acumulando la distancia total recorrida además de ir agregando a un array cada ciudad visitada.
* La función **get\_best\_track** busca y devuelve el mínimo recorrido entre todas las capitales.
* La función **get\_track\_with\_ag** busca y devuelve el mejor recorrido que une a todas las capitales utilizando algoritmos genéticos.

**chromosome.py**

* Mostrar sus genes en diferentes formatos (cada gen es una ciudad)
* Contiene la clase Chromosome donde cada cromosoma es un track (recorrido)
* Calcular función objetivo a partir de hacer el inverso a la sumatoria de todas las distancias entre ciudades a medida que avanza el recorrido
* Calcular su Fitness a partir de dividir la valuación de la función objetivo por la valuación de la función objetivo acumulada por todos los cromosomas

**population.py**

* Mostrar los cromosomas que la conforman
* Inicializarse con el número de cromosomas, la probabilidad de crossover y la probabilidad de mutación indicados
* Reproducirse, lo que incluiría la selección y crossover
* Seleccionar a los cromosomas dentro de una ruleta de manera probabilística según su Fitness
* Crear una ruleta que permita a los cromosomas más aptos tener mayores posibilidades de ser seleccionados, pero utilizando un rango de números reales para nuestra ruleta (Fitness Acumulativo)
* Calcular la puntuación objetivo acumulada por todos los cromosomas que la conforman
* Calcular el mínimo, máximo y promedio en cada generación (al inicializarse y después de cada reproducción)
* El crossover implementado es de tipo cíclico, pues se mantiene la estructura de genes todos diferentes intercambiando posiciones pero no se duplica ninguno, cosa indispensable en el TSP.
* La mutación que seleccionamos fue Swap Mutation, que en definitiva cambia N genes (en nuestro caso N=1) de posición entre sí, y tampoco hay repetición de cromosomas. Otra alternativa a Swap Mutation que también sirve para el TSP, entre otras, pudo haber sido Shift Mutation.
* Aplicación de Elitismo, cuya idea es hacer que los mejores cromosomas de una generación se mantengan en la generación siguiente. Nosotros aplicamos elitismo siempre a los 2 mejores cromosomas de la generación actual
* **Para aplicar el Elitismo hicimos lo siguiente**: calculamos las funciones objetivo de todos los cromosomas de la generación actual, y guardamos cuáles son los 2 con puntajes más altos; luego cuando llamamos al método que comienza con el proceso de selección y reproducción (lo primero que sucede en este método es instancias 2 listas vacías, una para los padres candidatos, y otra que tendrá los hijos de cada reproducción que se van a ir agregando a ésta para en el final reemplazar la población actual), y les pasamos los 2 cromosomas élites directamente a la lista de hijos (llamada NewGeneration) de forma que no haya manera de que se pierdan. Luego, como cualquier otro cromosoma, participarán de forma independiente de la ruleta, crossover y mutación. La siguiente generación tendrá entonces en su población a ambos cromosomas élites.

**Archivos HTML5 y CSS3 (Frontend)**

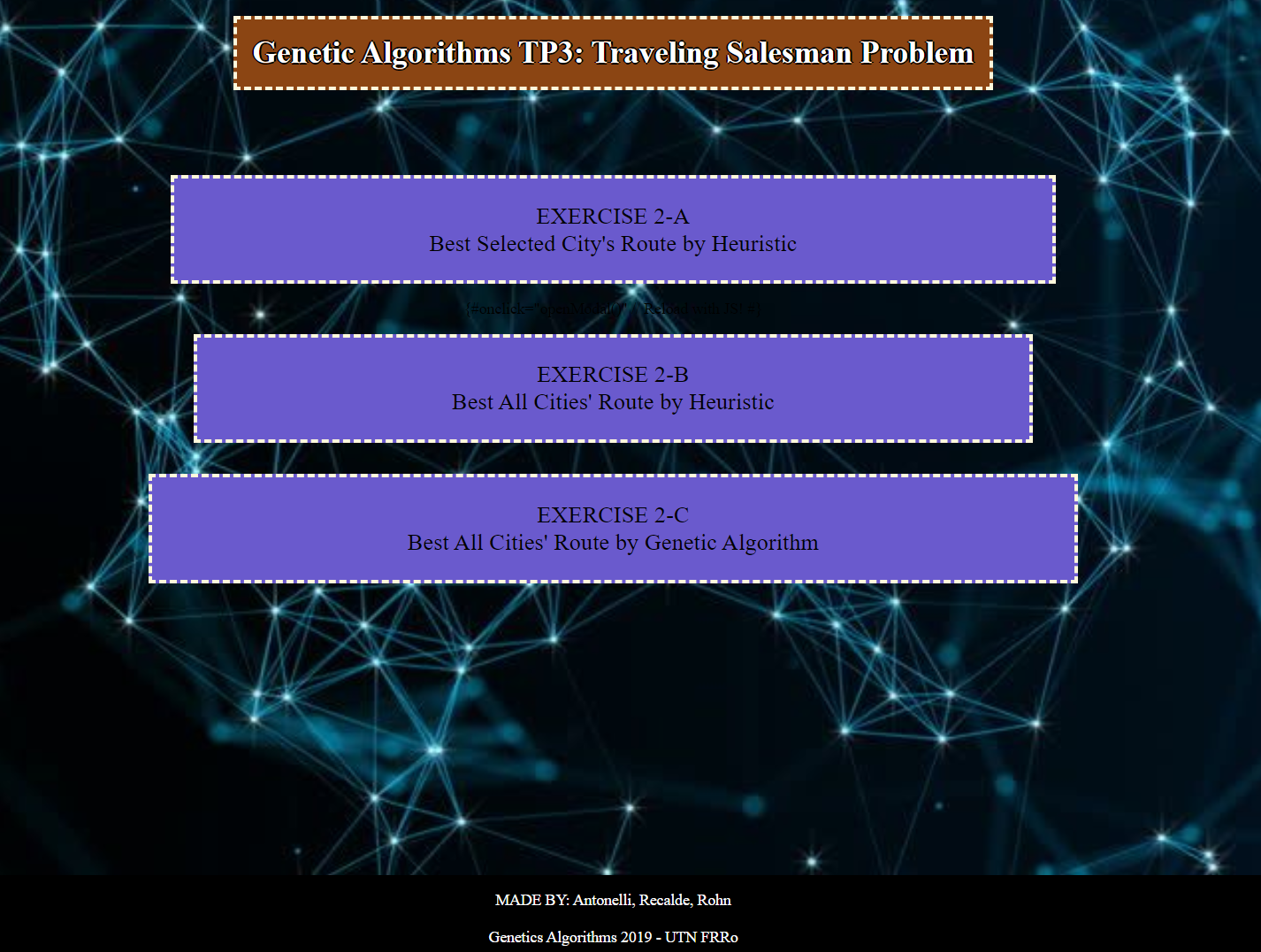
* Para enlazar Python con HTML y CSS, utilizamos un Framework llamado Flask
* Pantalla inicial index.html con la posibilidad de seleccionar el ejercicio correspondiente (Heurísticas y Algoritmo Genético)
* Pantalla showMap.html con el recorrido a exhibir en el mapa de Google Maps y la cantidad de kilómetros total, según lo que el resultado del ejercicio seleccionado le envíe
* Todos los estilos que utilizamos en styles.css.

**Archivos JavaScript (Frontend)**

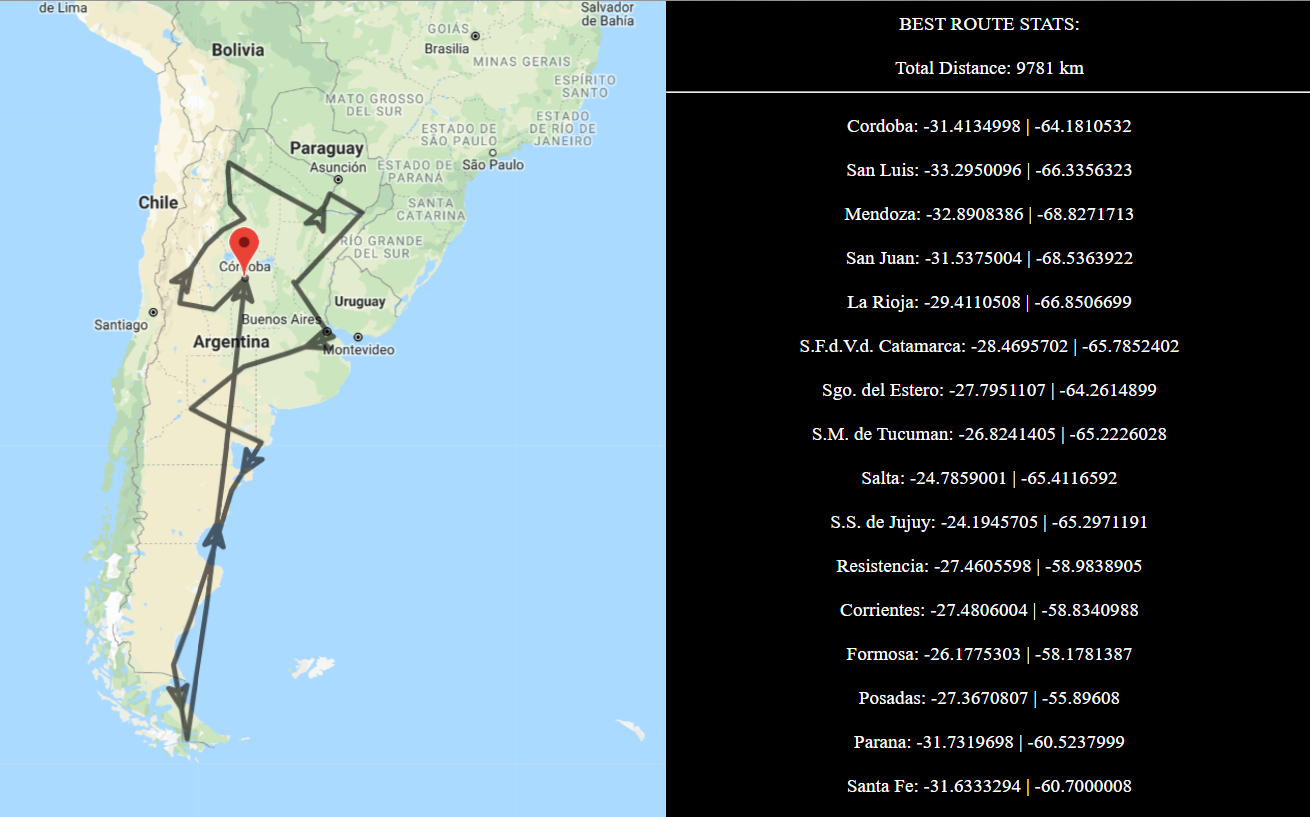
* Tenemos unos archivos: functions.js y mapConfigurations.js que es solo para comunicar correctamente la parte visual de nuestra página con la API de Google Maps, y exhibir lo que pide los ejercicios (lista de las ciudades recorridas y distancia total) más botón para volver al inicio y detalles.
* Solo lo utilizamos para eso, pues toda la lógica y funcionalidad de los ejercicios los hicimos totalmente en lenguaje Python.

# Salida de Pantalla

Menú Principal:

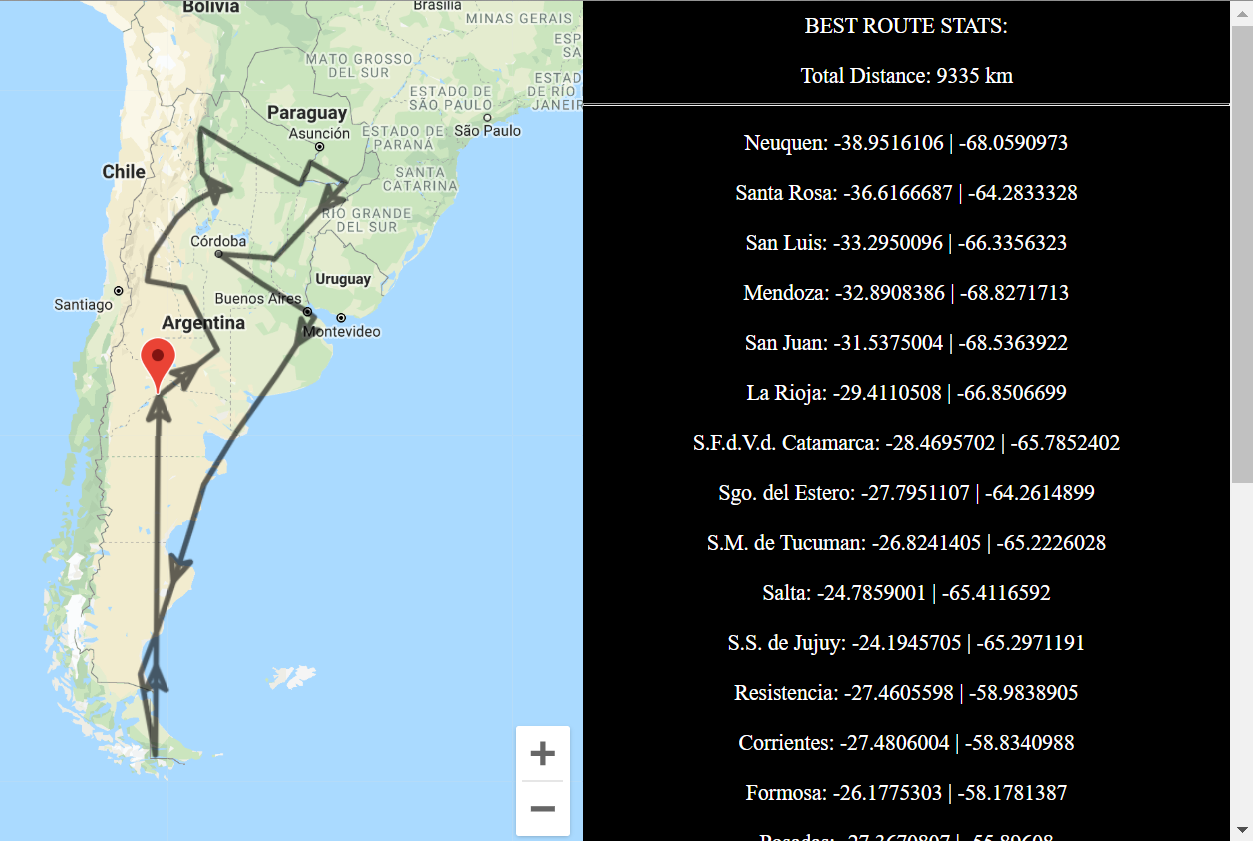


Ejercicio 2-A



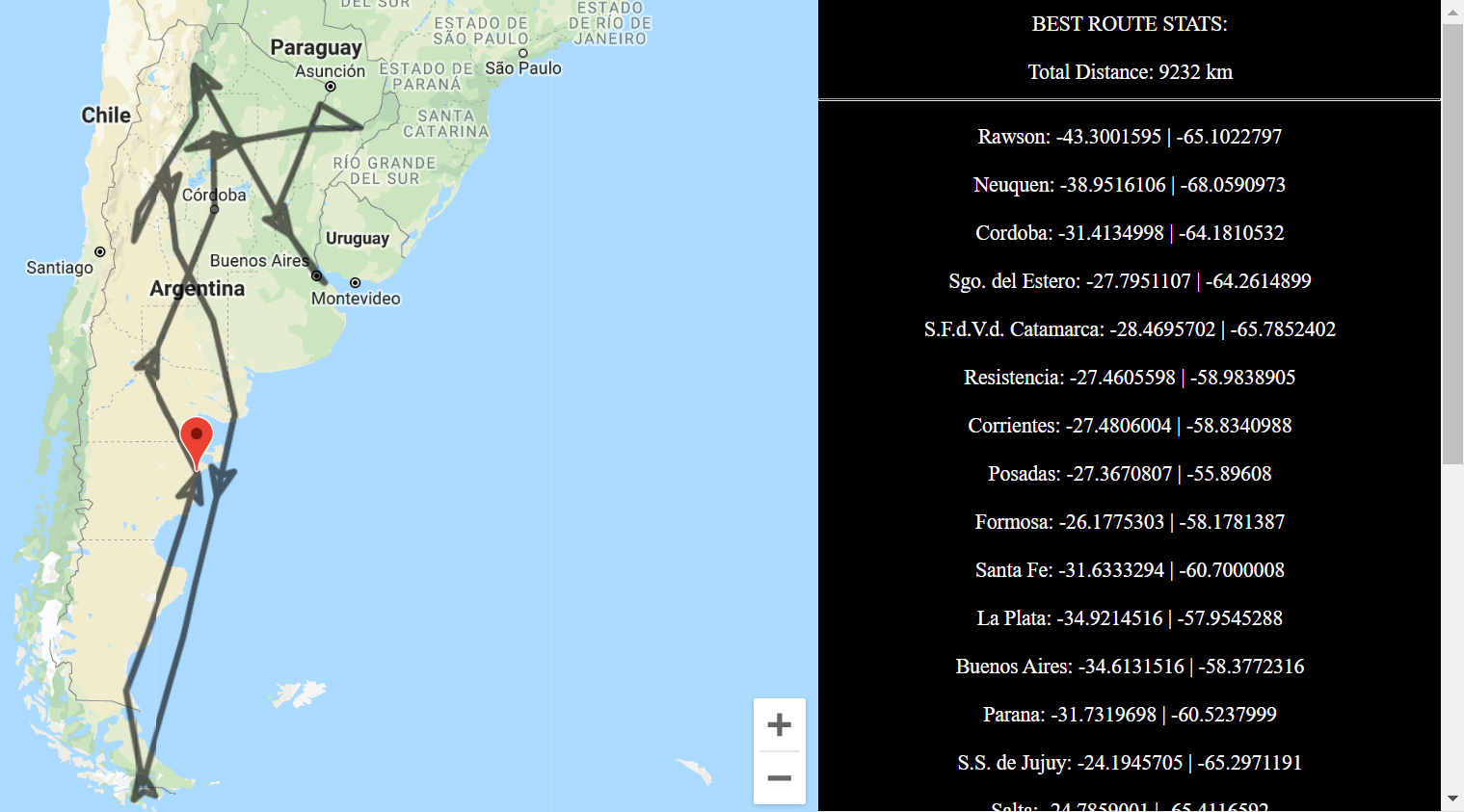
Cordoba: -31.4134998 | -64.1810532 , San Luis: -33.2950096 | -66.3356323 , Mendoza: -32.8908386 | -68.8271713 , San Juan: -31.5375004 | -68.5363922 , La Rioja: -29.4110508 | -66.8506699 , S.F.d.V.d. Catamarca: -28.4695702 | -65.7852402 , Sgo. del Estero: -27.7951107 | -64.2614899 , S.M. de Tucuman: -26.8241405 | -65.2226028 , Salta: -24.7859001 | -65.4116592 , S.S. de Jujuy: -24.1945705 | -65.2971191 , Resistencia: -27.4605598 | -58.9838905 , Corrientes: -27.4806004 | -58.8340988 , Formosa: -26.1775303 | -58.1781387 , Posadas: -27.3670807 | -55.89608 , Parana: -31.7319698 | -60.5237999 , Santa Fe: -31.6333294 | -60.7000008 , Buenos Aires: -34.6131516 | -58.3772316 , La Plata: -34.9214516 | -57.9545288 , Santa Rosa: -36.6166687 | -64.2833328 , Neuquen: -38.9516106 | -68.0590973 , Viedma: -40.8134499 | -62.9966812 , Rawson: -43.3001595 | -65.1022797 , Rio Gallegos: -51.6226082 | -69.218132 , Ushuaia: -54.7999992 | -68.3000031 , Cordoba: -31.4134998 | -64.1810532 , Total Distance = 9781

Ejercicio 2-B



Neuquen: -38.9516106 | -68.0590973 , Santa Rosa: -36.6166687 | -64.2833328 , San Luis: -33.2950096 | -66.3356323 , Mendoza: -32.8908386 | -68.8271713 , San Juan: -31.5375004 | -68.5363922 , La Rioja: -29.4110508 | -66.8506699 , S.F.d.V.d. Catamarca: -28.4695702 | -65.7852402 , Sgo. del Estero: -27.7951107 | -64.2614899 , S.M. de Tucuman: -26.8241405 | -65.2226028 , Salta: -24.7859001 | -65.4116592 , S.S. de Jujuy: -24.1945705 | -65.2971191 , Resistencia: -27.4605598 | -58.9838905 , Corrientes: -27.4806004 | -58.8340988 , Formosa: -26.1775303 | -58.1781387 , Posadas: -27.3670807 | -55.89608 , Parana: -31.7319698 | -60.5237999 , Santa Fe: -31.6333294 | -60.7000008 , Cordoba: -31.4134998 | -64.1810532 , Buenos Aires: -34.6131516 | -58.3772316 , La Plata: -34.9214516 | -57.9545288 , Viedma: -40.8134499 | -62.9966812 , Rawson: -43.3001595 | -65.1022797 , Rio Gallegos: -51.6226082 | -69.218132 , Ushuaia: -54.7999992 | -68.3000031 , Neuquen: -38.9516106 | -68.0590973 , Total Distance = 9335

Ejercicio 3



Last Generation Reached Correctly

------------

Best Track: Rawson, Neuquen, Cordoba, Sgo. del Estero, S.F.d.V.d. Catamarca, Resistencia, Corrientes, Posadas, Formosa, Santa Fe, La Plata, Buenos Aires, Parana, S.S. de Jujuy, Salta, S.M. de Tucuman, San Juan, Mendoza, La Rioja, San Luis, Santa Rosa, Viedma, Ushuaia, Rio Gallegos, Rawson (Vuelta al Inicio)

Total Distance: 9232 km, after 500 generations with 300 Chromosomes

# Conclusiones

Como vimos en el primer ejercicio la búsqueda exhaustiva no brindaría resultados en un intervalo de tiempo práctico.

Utilizando la búsqueda heurística obtuvimos, en muy poco tiempo, un muy buen resultado al recorrer todas las capitales yendo siempre a la más cercana.

Respecto a la implementación de algoritmos genéticos nosotros utilizamos Crossover Cíclico y Swap Mutation para mantener la estructura de los cromosomas lo cual nos evita que se corrompan y surjan ciudades repetidas dentro de los recorridos.

En un principio habíamos llegado a la conclusión de que, empíricamente, para obtener buenos resultados (distancias de entre 9200km y 10000km) debimos definir una población de al menos 200 cromosomas y al menos 1500 generaciones. Sin embargo, logramos mejorar esto bastante. Utilizamos un valor de probabilidad de mutación: de 0.1 y de crossover: de 0.75.

Al agregar elitismo notamos que la convergencia sucede muy tempranamente por lo que aumentamos la mutación a por lo menos un 0.4 para salir del estancamiento producido. Aun así debemos correr el programa varias veces para obtener un resultado aproximado o mejor al obtenido por la heurística. Hasta acá nuestro mejor recorrido rondaba cerca de 9150km. A razón de coste/beneficio de aumentar las iteraciones y/o el tamaño de la población, es una solución muy buena.

Sin embargo, experimentando un poco más, determinamos que aumentar la cantidad de cromosomas a 1000, y compensar parte del tiempo de procesamiento mayor requerido bajando la cantidad de generaciones a 200, y pasar de valores para la probabilidad de crossover: de 0.75 a 0.9 y de mutación: de 0.4 a 0.65, se obtenían resultados mucho mejores, y que un recorrido no sea bueno se volvió menos frecuente. Logramos obtener algunos recorridos de menos de 9000km.