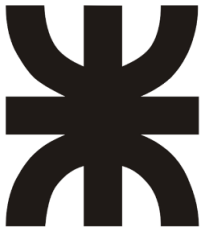
Trabajo práctico N° 3
TSP (Traveling Salesman Problem)



Autores

Fernández, Natalia
Comisión 1
Legajo 44758
nata.fernandez77@gmail.com

Benedetti, Juan Ignacio
Comisión 1
Legajo 45403
juanigbenedetti@gmail.com

Cutró, Gastón
Comisión 1
Legajo 44757
gastoncutro365@gmail.com

Gofiar, Cristian
Comisión 1
Legajo 44839
cristiangofiar@gmail.com

Cabanellas, Ignacio
Comisión 1
Legajo 44987
Ignaciomcabanellas@gmail.com

Índice

Índice de figuras

1. Metodología de desarrollo abordada.

Las funciones principales utilizadas son:

1. Mutación: Utilizando el método de swap cuando elegido aleatoriamente diferentes genes de un cromosoma son intercambiados
2. seleccross: Utilizando el método de ruleta dándoles chances proporcionales a su función objetivo para cada individuo, se eligen 2 padres para generar 2 hijos mediante un crossover cíclico
3. crossoverCiclico: Dados 2 padres devuelve 2 hijos mediante el método del crossover cíclico
4. busquedaHeuristica: Búsqueda del camino más eficiente con el método heurístico que desde cada ciudad en la que se encuentra se moverá a la ciudad más cercana a la que todavía no se haya usado
5. imprimeMapa: utilizando la librería plotly y pasándole el arreglo que representa la trayectoria del viajante se muestra en un mapa el recorrido
6. guardaListas: Guarda datos de las generaciones para poder luego utilizarlas a la hora de generar gráficas y tablas.
7. muestraGrafica: utilizando los datos guardados muestra las gráficas y crea una tabla excel con la información
8. generaCromosomas: Genera los arreglos que representan el viaje del vendedor de manera aleatorea, con 24 números que van del 0 al 23, genera nuevos hasta la cantidad requerida
9. calculaFitness: Calcula el fitness de cada individuo mediante una fórmula que le da un porcentaje adecuado la cual al sumar todas llegue al 100

2. Herramientas de programación utilizadas.

Lenguaje de programación: Python. librerías:

1. numpy para el manejo de estructuras de datos necesarias para el trabajo
2. Random para crear y manejo de números aleatorios
3. matplotlib para generar gráficas
4. pandas para manejo de estructuras de datos necesarias para otras librerías
5. math fue dependencia de otra librería
6. os para manejar datos en consola y en el sistema (como los archivos excel que generamos)
7. Xlswriter para crear el archivo excel
8. plotly para graficar el mapa

3. Descripción de la forma de trabajo abordada en equipo.

La metodología de trabajo fue, separarnos puntos del programa para cada uno (la búsqueda heurística, la genética, graficación del mapa, generación del excel, etc), proceder cada uno a investigar o preparar en parte esa categoría, luego ir a una llamada por Discord y uno por uno van planteando lo que haya investigado y entre todos aportamos para poder hacer el código por completo (también depuramos el código entre todos a medida que habían errores).

Aplicaciones utilizadas: Discord: para pasarnos archivos, comunicarnos en llamada de voz, y compartir pantalla para que vean información que queremos mostrar en el momento WhatsApp: lo utilizamos para repartir los temas y organizar fechas y horarios para conectarnos a discord y hacer el trabajo en conjunto.

4. Ejercicio 1: Método exhaustivo

Enunciado: Hallar la ruta de distancia mínima que logre unir todas las capitales de provincias de la República Argentina, utilizando un método exhaustivo. ¿Puede resolver el problema? Justificar de manera teórica.

Respuesta.
La búsqueda exhaustiva se basaría en buscar todas las combinaciones posibles de los viajes de una ciudad a otra sin repetir hasta pasar por cada ciudad de la lista una sola vez. No es posible ejecutar un algoritmo así porque, sabiendo que tenemos 23 ciudades que visitar, podemos calcular la cantidad de permutaciones calculando 23! (factorial de 23), lo que da un total de 25.852.016.738.884.976.640.000 combinaciones. Con 20 permutaciones el tiempo de ejecución aproximado serian de más de 77.000 años

5. Ejercicio 2: Utilizando heurística

5.1. Yendo a la ciudad más cercana no visitada desde cada ciudad

Permitir ingresar una provincia y hallar la ruta de distancia mínima que logre unir todas las capitales de provincias de la República Argentina partiendo de dicha capital utilizando la siguiente heurística: "Desde cada ciudad ir a la ciudad más cercana no visitada." Recordar regresar siempre a la ciudad de partida. Presentar un mapa de la República con el recorrido indicado. Además indicar la ciudad de partida, el recorrido completo y la longitud del trayecto. El programa deberá permitir seleccionar la capital que el usuario desee ingresar como inicio del recorrido.

Código del problema. El código que se encuentra debajo puede utilizarse para representar los diferentes recorridos desde y hacia diferentes capitales a elección del usuario. Para cada elección mostrará el camino óptimo a través del uso de heurística.

```
1 def menu1():
2
3     print()
```

```
    print("Ingrese una capital de partida")
    print()
    for i in range(len(nombresCapital)):
        print(i," - ", nombresCapital[i])
    capi = int(input())
    capingresada = nombresCapital[capi]
    ciudadInicio, CiudadesViajadas, distTotalRecorrida = busquedaHeuristica(
    capingresada)

    os.system('cls')
    print()
    print("Capital de partida: ", ciudadInicio)
    print("Indice de ciudades viajadas: ", CiudadesViajadas)
    print("Distancia total recorrida en km: ",distTotalRecorrida)
    print()
    for i in CiudadesViajadas:
        print(nombresCapital[i])
    imprimeMapa(CiudadesViajadas)
    input()

######## BUSQUEDA HEURISTICA ########
def busquedaHeuristica(capital):

    CiudadesViajadas = []
    distTotalRecorrida = 0
    ciudadInicio = capital
    ciudadActual = capital
    dCiudades = []

    CiudadesViajadas.append(nombresCapital.index(ciudadInicio))

    for i in range(len(nombresCapital)-1):
        iCapital = nombresCapital.index(ciudadActual)
        dCiudades = matrizDeDistancias[iCapital][:]

        for i in CiudadesViajadas:
            dCiudades[i] = 0

        dMin = mt.infinity
        iMin = -1

        for i in range(len(dCiudades)):
            if(dCiudades[i] != 0):
                if(dCiudades[i] < dMin):
                    dMin = dCiudades[i]
                    iMin = i

        distTotalRecorrida += dMin
        CiudadesViajadas.append(iMin)
        ciudadActual = nombresCapital[iMin]

    arrayDistancias = matrizDeDistancias[nombresCapital.index(ciudadActual)]
    recorridofinal = arrayDistancias[nombresCapital.index(ciudadInicio)]
    distTotalRecorrida += recorridofinal
    CiudadesViajadas.append(nombresCapital.index(ciudadInicio))

    return (ciudadInicio, CiudadesViajadas, distTotalRecorrida)
```

Listing 1: Heuristica

5.2. Encontrando el recorrido mínimo

Encontrar el recorrido mínimo para visitar todas las capitales de las provincias de la República Argentina siguiendo la heurística mencionada en el punto a. Deberá mostrar como salida el recorrido y la longitud del trayecto.

Resultados. Mediante la ejecución del código que se encuentra en el próximo apartado pudimos obtener buenos resultados. Puede observarse que, para una población formada por 50 generaciones y realizando 200 iteraciones, se obtuvo el resultado mostrado en la figura de debajo.

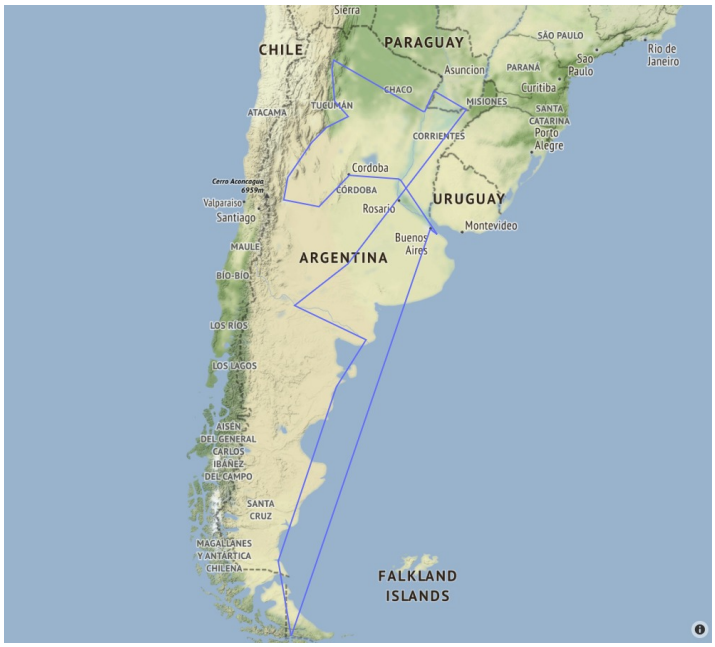


Figura 1: Mapa resuelto con heuristica.

Para esta figura, el resultado óptimo obtenido mendiane la ejecución del algoritmo fue la siguiente:

- **Capital de partida:** Neuquen.
- **Indice de ciudades viajadas:** [7, 20, 18, 6, 17, 5, 13, 21, 14, 16, 15, 11, 2, 3, 9, 8, 19, 1, 0, 4, 23, 10, 12, 22, 7]
- **Distancia total recorrida en km:** 9335.0

Las ciudades viajadas, para verlo de una manera mas representativa y entendible, las enumeraremos a continuación:

1. Neuquen
2. Santa Rosa
3. San Luis

4. Mendoza
5. San Juan
6. La Rioja
7. Fernando del Valle de Catamarca
8. Santiago del Estero
9. San Miguel de Tucumán
10. Salta
11. San Salvador de Jujuy
12. Resistencia
13. Corrientes
14. Formosa
15. Posadas
16. Paraná
17. Santa Fe
18. Córdoba
19. Ciudad de Buenos Aires
20. La Plata
21. Viedma
22. Rawson
23. Río Gallegos
24. Ushuaia
25. Neuquen

Código del problema. El detalle del codigo utilizado para resolver el problema se encuentra a continuación. Cabe aclarar que fue reutilizada la función de heurística mostrada en el ejercicio anterior.

```
######## MENU 2 ########
def menu2():

    menorDistancia = 0
    for i in nombresCapital:
        ciudadInicio, CiudadesViajadas, distTotalRecorrida = busquedaHeuristica
    (i)
        if (distTotalRecorrida < menorDistancia) or (menorDistancia == 0):
            menorDistancia = distTotalRecorrida
            menorCiudadInicio = ciudadInicio
```

```
            menorCiudadesViajadas = CiudadesViajadas

    os.system('cls')
    print("Segun la heuristica, el menor recorrido corresponde a:")
    print()
    print("Capital de partida: ", menorCiudadInicio)
    print("Indice de ciudades viajadas: ", menorCiudadesViajadas)
    print("Distancia total recorrida en km: ", menorDistancia)
    print()
    print("Lista de ciudades viajadas:")
    print()
    for i in menorCiudadesViajadas:
        print(nombresCapital[i])
    imprimeMapa(CiudadesViajadas)
    input()
```

Listing 2: Heuristica

5.3. Algoritmo genético

Hallar la ruta de distancia mínima que logre unir todas las capitales de provincias de la República Argentina, utilizando un algoritmo genético.

Resolución del problema. Para resolverlo con algoritmos genéticos hicimos uso de un crossover cíclico el cual nos permite realizar la reproducción entre dos cromosomas, cuando los mismos no son binarios. Como en nuestro caso estamos trabajando con índices de ciudades y no con bits, nos resulta util implementarlo.
Se utilizó una población inicial de tamaño 50 generada de forma aleatoria así como una cantidad de generaciones de 200. El resultado obtenido fue el siguiente:
Parametros iniciales

- Ingrese tamaño de poblacion: 50
- Ingrese cantidad de generaciones : 200
- Ingrese tasa de mutación (%): 5
- Ingrese tasa de crossover (%): 75

Ejecución Del total de las generaciones se obtuvo el cromosoma óptimo (en nuestro caso será el recorrido que posea una distancia total mínima). El resultado de esto fue el siguiente:

- Cromosoma óptimo: [10, 22, 12, 7, 20, 6, 17, 21, 1, 8, 16, 15, 3, 9, 2, 11, 0, 23, 18, 5, 13, 14, 19, 4, 10]
- Función objetivo óptima: 13348.0

A modo de representar la ejecución podemos mostrar gráficamente lo que fue ocurriendo con el pasar de las generaciones. Para esto presentamos una gráfica que muestra el cromosoma mínimo, máximo y promedio de cada generación. El cromosoma máximo se muestra para representar el peor de todos los cromosomas existentes de cada generación. Entonces la gráfica resulta:

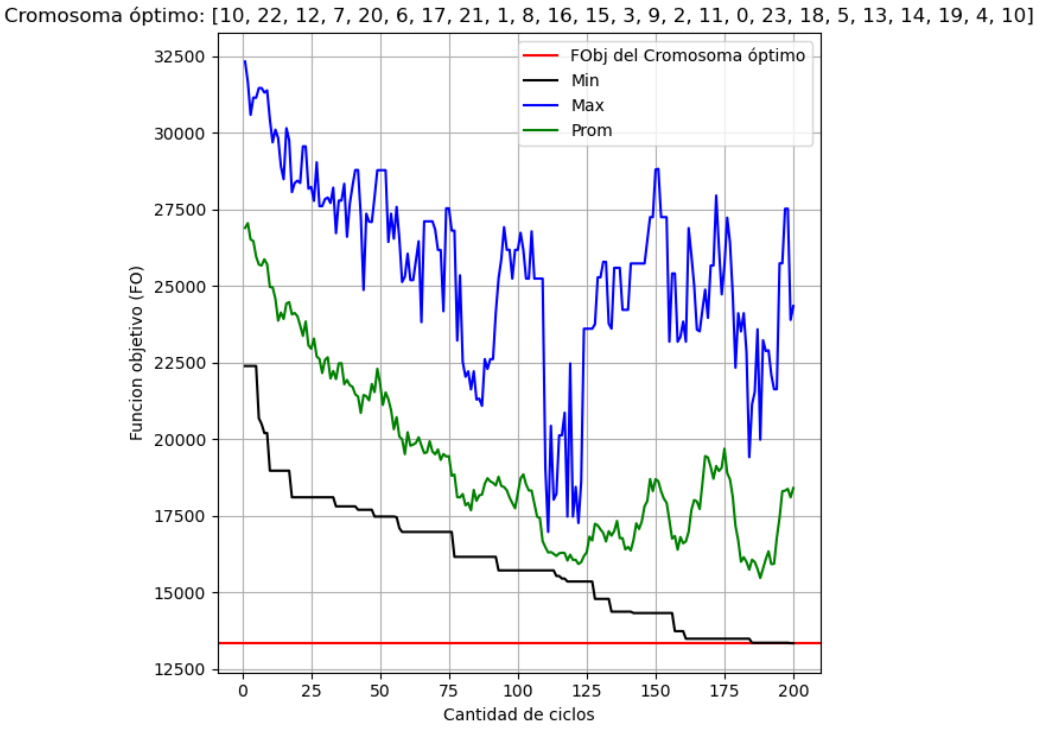


Figura 2: Resuelto con Algoritmos genéticos.

Ejecutándolo varias veces obtuvimos resultados que oscilaban en torno a los mismos valores y esto nos da un indicio de que se esta ejecutando de forma correcta.

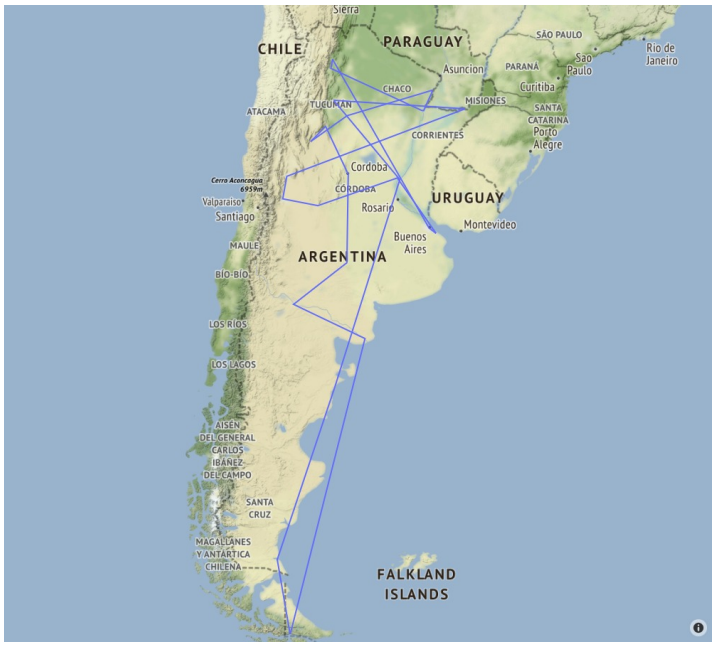


Figura 3: Mapa resuelto con algoritmos genéticos.

A continuación se muestra la tabla Excel obtenida en la misma ejecución. Debido a que se parametrizó con un número alto de generaciones (200), sólo se presentarán tres partes, a fin de no incluir demasiadas imágenes dentro del informe.

Inicio: donde se puede observar que el promedio de la función objetivo es muy alto. Recordemos que estamos intentando minimizar una función, es decir, la distancia total recorrida por el viajante (en Km).

Generacion	Minimo FO	Maximo FO	Promedio FO
1	21603	31427	27031.2
2	21603	31488	27089.8
3	19982	31088	26062.98
4	19216	31088	26159.5
5	19216	31353	26321.52
6	19216	31088	26618.14
7	19216	31439	26499.8
8	18929	31079	26473.32
9	18929	31303	26440.76
10	18929	31303	26948.28
11	18929	31390	27072.14
12	18929	31390	26811.92
13	18929	31137	26364.18
14	18929	31639	26353.28
15	18929	32040	26248.52
16	18929	30940	25816.06
17	18929	30940	25591.22
18	18929	30940	25203.34
19	18929	30940	25089.58
20	18929	30940	24725.18
21	18929	30940	24702.46
22	18929	30940	25823.46
23	18929	30940	25437.52
24	18929	30940	25469.56
25	18929	30940	24671.74
26	18929	30406	24523.64

Figura 4: Tabla Excel algoritmos genéticos. Parte 1

Parte intermedia: podemos ver que los promedios han disminuido escasamente.

80	15628	29800	21880.84
81	15628	29129	21350.94
82	15628	28666	20602.08
83	15628	28666	20122.6
84	15628	29472	21002.8
85	15628	29485	20830.84
86	15628	29355	20613.3
87	15628	29355	20687.04
88	15628	29472	20582.3
89	15628	29151	20987.04
90	15628	29151	20554.86
91	15628	28267	20202.74
92	15628	28253	18896.78
93	15628	29151	18369.1
94	15628	26206	17943.42
95	15393	25705	18548.44
96	15393	24324	19469.12
97	15393	24006	19168.82
98	15393	25089	19963.52
99	15393	24101	19964.32
100	15393	24101	19371.76
101	15393	26035	19751.22
102	15324	25875	18745.7
103	15324	25532	18888.56
104	15324	27639	18916.14
105	15324	27311	18772.62
106	15324	27311	19448.9
107	15324	27247	19359.22
108	15112	27311	19040.3
109	15112	27238	19422.76
110	15011	27639	19622.42
111	15011	27238	19152.26
112	15011	27921	20424.52

Figura 5: Tabla Excel algoritmos genéticos. Parte 2

Final: se aprecia que el algoritmo converge a un valor sustancialmente bajo.

168	14123	22256	16108.44
169	14123	22256	15704.98
170	14123	23254	15530.74
171	14123	20367	15154.66
172	14123	23713	15233.96
173	14123	22057	15300.94
174	14123	27044	15511.7
175	14123	19822	15262.8
176	14123	19599	15168.1
177	14123	23129	15666.46
178	14123	22770	15866.44
179	14123	24343	15963.14
180	14123	23054	16087.82
181	14123	22930	16281.52
182	14123	19599	15701.3
183	14123	20299	16022.84
184	14123	21293	16142.7
185	14123	19296	16240.12
186	14123	21957	16223.22
187	14123	19772	16263.1
188	13956	19182	15864.64
189	13871	20717	15822.82
190	13871	20717	16123.62
191	13871	21358	16595.5
192	13871	21358	17057.64
193	13871	21358	17238.64
194	13871	22870	17836.06
195	13871	23005	18094.7
196	13871	24560	17774.38
197	13871	24560	18034.2
198	13871	24560	17324.3
199	13871	23483	17374.2
200	13871	22543	17072.64

Figura 6: Tabla Excel algoritmos genéticos. Parte 3

Codigo utilizado

```
######## GENERA CROMOSOMAS ########
def generaCromosomas():

    global cromosomas
    cromosomas = []

    for i in range(cantPob):
        cromosomas.append(rand.sample(list(listaNumeros), 24))
        cromosomas[i].append(cromosomas[i][0])


######## FUNCION OBJETIVO ########
def calcula_f_obj():

    for i in range(cantPob):
        cromosoma = cromosomas[i]
        suma = 0
        for j in range (len(cromosoma)-1):
            distancia = matrizDeDistancias[cromosoma[j]][cromosoma[j+1]]
            suma += distancia
        f_obj[i] = suma


######## FITNESS ########
def calcula_fitness():

    sumatoria = sum(f_obj)
    for i in range(cantPob):
        fitness[i] = (1 - (f_obj[i]/sumatoria))


######## RULETA ########
def calcula_ruleta():

    nuevoFitness = list(np.zeros(cantPob))
    sumaFitness = sum(fitness)
    for i in range(len(fitness)):
        nuevoFitness[i] = (fitness[i]/sumaFitness)
    frec_acum = []
    frec_acum.append(nuevoFitness[0])
    for i in range(1,cantPob):
        acumulado = frec_acum[i - 1] + nuevoFitness[i]
        frec_acum.append(acumulado)

    return frec_acum


######## TIRADA DE RULETA ########
def tiradas(ruleta):
    padres = []
    for m in range(2):
        frec = rand.uniform(0,1)

        for i in range(cantPob):
            if(ruleta[i] > frec):
                cromosomas[i].pop()
                padres.append(cromosomas[i])
```

```
                cromosomas[i].append(cromosomas[i][0])
                break

    return padres[0], padres[1]


######## CROSSOVER CICLICO ########
def crossoverCiclico(c1, c2):

    band = True
    hijo = c2.copy()
    indice = 0
    finDeCiclo = c1[indice]

    while(band):
        hijo[indice] = c1[indice]
        numAValidar = c2[indice]
        indice = c1.index(numAValidar)
        if(numAValidar == finDeCiclo):
            band = False

    return hijo


######## CROSSOVER ########
def crossover(c1, c2):

    a = crossoverCiclico(c1, c2)
    b = crossoverCiclico(c2, c1)

    return a, b


######## SELECCION Y CROSSOVER ########
def selec_cross():

    global cromosomas

    ruleta = calcula_ruleta()
    aux = np.array(f_obj)
    min1 = aux.argsort()[0]
    min2 = aux.argsort()[1]
    aux = list(aux)
    aux[0] = cromosomas[min1]
    aux[1] = cromosomas[min2]

    for i in range(2, len(cromosomas)-1, 2):
        c1,c2 = tiradas(ruleta)
        c1,c2 = crossover(c1,c2)
        aux[i] = c1
        aux[i+1] = c2
    cromosomas = aux[:]


######## MUTACION ########
def mutacion():

    for i in range(len(cromosomas)):
        x = np.random.randint(0, 101)
```

```
117             cromosomas[i].pop()
118             if x <= probMut:
119                 posicion1 = np.random.randint(0, len(nombresCapital))
120                 posicion2 = np.random.randint(0, len(nombresCapital))
121                 aux1 = cromosomas[i][posicion1]
122                 aux2 = cromosomas[i][posicion2]
123                 cromosomas[i][posicion1] = aux2
124                 cromosomas[i][posicion2] = aux1
125             cromosomas[i].append(cromosomas[i][0])
```

Listing 3: Heuristica

6. Aportes Prácticos del TSP

El uso de los algoritmos de resolución de tsp tienen muchos usos, que aunque no suelen ser cotidianos, son más abundantes y necesarios de lo que podríamos pensar, entre ellos están:

- Algoritmos de resolución de TSP en tiempo real para la optimización de rutas para encontrar caminos óptimos en fracciones de segundos, le da a delivers de organizaciones la habilidad para planear rutas rápida y eficientemente.
- La toma de decisiones en el orden de las perforaciones realizadas en circuitos electrónicos. Para poder conectar un conductor de un lado del circuito con uno del otro lado o para posicionar pines de circuitos integrados, se deben hacer perforaciones en la placa. Ya que las perforaciones pueden ser de diferentes tamaños, para hacer agujeros de diferentes tamaños de manera consecutiva, la maquinaria debe moverse hasta una caja de herramientas y cambiar el equipamiento de perforación. Esto puede tomar bastante tiempo, por lo que se vuelve claro que se deberá elegir un diámetro, hacer todas las perforaciones de ese diámetro, cambiar el cabezal del taladro, y hacer todos las perforaciones del nuevo tamaño, y repetir ese proceso hasta terminar. Así, este problema de perforaciones se puede ver como una serie de TSPs(traveler salesman problem o problema del viajante), una por cada diámetro de perforación, donde las “ciudades” son las posiciones iniciales y el set de todos los agujeros que se pueden perforar con el mismo taladro. La distancia entre cada par de ciudades se da por el tiempo que le toma mover el taladro de una posición a la otra. El enfoque es minimizar el tiempo de viaje para el cabezal de la máquina.
- Reconstrucciones de turbinas de gas
- Rayos x en cristalografía para analizar la estructuras de cristales
- Cableado de computadores
- Orden de Extracción de materiales en depósitos
- Recorridos vehiculares (como por ejemplo un repartidor)
- Organización de una imprenta
- Diseñar trayectorias para colectivos
- ”Crew scheduling problem” que hace referencia a la organización de un grupo que tiene que recolectar dinero de múltiples bancos de manera simultánea

- Organización de entrevistas para agentes turísticos
- Optimización de la preparación para la utilización de una máquina de laminación en caliente
- Planificación de rutas para misiones militares
- Diseño de ruta sistemas satelitales

7. Código extra: Menúes, visualización del mapa y gráfica.

```
######## MENU ########
def menu():

    while True:
        print()
        print(''Menu de opciones - Problema del viajante'')
        print()
        print("1 Busqueda heuristica dada una capital de partida")
        print("2 Busqueda heuristica general")
        print("3 Busqueda mediante algoritmos geneticos")
        print("0 Salir")
        print()
        op = input()
        if op == "1":
            os.system('cls')
            menu1()
            os.system('cls')
        elif op == "2":
            os.system('cls')
            menu2()
            os.system('cls')
        elif op == "3":
            os.system('cls')
            menu3()
            os.system('cls')
        elif op == "0":
            break
        else:
            os.system('cls')
            print()
            print("  - Opcion invalida -")

######## IMPRIME MAPA ########
def imprimeMapa(CiudadesViajadas):

    listaOrdenada = []
    dfCapitales = pd.read_csv('provincias.csv')
    listaCapitales = dfCapitales.values.tolist()
    for i in CiudadesViajadas:
        listaOrdenada.append(listaCapitales[i])
        fig = px.line_mapbox(listaOrdenada, lat=1, lon=2, zoom=3, width=1000,
    height=900)
        fig.update_layout(mapbox_style="stamen-terrain", mapbox_zoom=3.8,
    mapbox_center_lat = -40,margin={"r":0,"t":0,"l":0,"b":0})
    fig.show()
```

Listing 4: Heuristica

8. Conclusión.

A lo largo del trabajo práctico fuimos visualizando diferentes características de cada método utilizado y es por esta razón que queremos diferenciar, entre los 3 algoritmos, su desempeño en diferentes categorías que en nuestra opinión sirven muy bien para comprarlos:

Exhaustiva:

- Complejidad a la hora de programarlo: Simple.
- Precisión del resultado: perfecta. (Si se pudiese resolver en un tiempo razonable)
- Tiempo de ejecución: Virtualmente infinita.

Heurística:

- Complejidad a la hora de programarlo: intermedia.
- Precisión del resultado: Muy buena
- Tiempo de ejecución: pocos segundos

Genético:

- Complejidad a la hora de programarlo: Compleja.
- Precisión del resultado: Buena
- Tiempo de ejecución: pocos segundos.

Con estos resultados podemos asumir que al no poder ejecutar la búsqueda exhaustiva por el tiempo requerido, la próxima mejor opción es la heurística, pero que tanta diferencia en resultados hay entre la heurística y la genética ? Heurística: Entre 9000 y 10.500 dependiendo de en qué ciudad inicia Genética: con los parámetros 100 de población inicial, 2 % de mutación, 9 % de crossover, y 250 generaciones, conseguimos un promedio de 12.451 (en 10 ejecuciones del programa diferentes), lo que no está tan alejado a los resultados con heurística pero es una diferencia clara. Dicho esto consideramos que los resultados del algoritmo genético denotan claramente que inclusive acercándose a resultados óptimos. algoritmos destinados a resolver el problema en específico tienden a dar mejores resultados.

Referencias

[1] TSP PROBLEM In *https://www.intechopen.com/books/traveling-salesman-problem-theory-and-applications/traveling -salesman-problem-an-overview-of-applications-formulations-and-solution-approaches* , 2020.