Universidad Tecnológica Nacional

FRRO

TP N° 3 – Título:

“Aplicaciones para la resolución del problema del viajante”

Año: 2023 – Comisión 303

Docentes:

Teoría: Daniela E. Díaz

Práctica: Victor Lombardo

Integrantes del grupo:

Nombre y Apellido	Email	N° de Legajo
Marcos del Solar	m4rc0s.d3l.s0l4r@gmail.com	49856
Rodrigo Marí	rorro2001@gmail.com	49612
Lautaro Teta Musa	lautarotetamusa@gmail.com	49695

Fecha Presentación: 23/10/2023

Contenido

Enunciado 3
Código 5
Extraer datos del Excel 5
Main 5
Heurístico 7
Genético 8
Lógica 12
Búsqueda exhaustiva 12
Búsqueda heurística 12
Por Algoritmo Genético 14
Conclusiones 16
Mejor recorrido con búsqueda heurística 16
Mejor recorrido usando AG 17
Aportes prácticos de TSP 17
Problema de la red de basuras 17
Problema de placa de circuitos impresos PCB 18

Enunciado

1. 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.

2. Realizar un programa que cuente con un menú con las siguientes opciones:

   a. 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.

   b. 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.

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

Recomendaciones para el algoritmo:

- N = 50 Número de cromosomas de las poblaciones.
- M = 200 Cantidad de ciclos.

Cromosomas: permutaciones de 23 números naturales del 1 al 23 donde cada gen es una ciudad. Las frecuencias de crossover y de mutación quedan a criterio del grupo. Se deberá usar crossover cíclico.

Comparar los resultados obtenidos entre la resolución a través de heurísticas y con algoritmos genéticos a través de una conclusión que deberá anexarse al informe. Agregar en el informe un apartado final denominado «Aportes Prácticos del TSP» donde se expliquen algunas aplicaciones en las que actualmente se use el problema del viajante. Tomar por lo menos dos y explicarlas.

El programa debe tener las siguientes opciones:

- Resolución por Búsqueda heurística:
    - Mostrar el recorrido mínimo, nombrar todas las ciudades que lo componen y la cantidad de kilómetros recorridos.
    - Permitir seleccionar una ciudad y mostrar el recorrido partiendo de la misma y volviendo a ésta habiendo visitado una vez todas las ciudades. También mostrar la cantidad de kilómetros recorridos.

- Resolución por AG:
    - Mostrar el recorrido mínimo, nombrar todas las ciudades que lo componen y la cantidad de kilómetros recorridos

Código

Extraer datos del Excel

```
import csv


#Cargar las distancias desde un archivo .csv
#Devuelve la matriz de distancias
#Devuelve el mapa de nombres de las ciudades
def load_distancias(file_path = "TablaCapitales.csv"):
    file_tabla = file_path
    distancias = []

    with open(file_tabla, "r") as tabla:
        reader = csv.reader(tabla);
        header = next(reader)
        ciudades = header[1:len(header)]
        for row in reader:
            first = True
            d = []
            for col in row:
                if not first:
                    if col != "":
                        d.append(int(col))
                    else:
                        d.append(0)
                else:
                    first = False
            distancias.append(d)

    return distancias, ciudades
```

Main

```
from genetico import metodo_genetico, N_CIUDADES, N_CICLOS
from draw_map import dibujar_ruta
from heuristica import metodo_heuristico, heuristico_general
from util import load_distancias
import sys
import os

#distancias: list[list[int]] = []

def usage():
    print("----------- USAGE ------------")
    print("python main.py <Method>\n")
```

```
    print("METHODS: ")
    print("   'heuristico origen'  Método Heuristico partiendo desde una ciudad")
    print("   'heuristico general' Método Heuristico general")
    print("   'genetico' Método Genetico")

commands = {
    "heuristico origen": metodo_heuristico,
    "heuristico general": "TODO",
    "genetico": metodo_genetico,
}

if __name__ == "__main__":
    if (len(sys.argv) > len(commands) or len(sys.argv) < 2):
        usage()
        exit(1)
    else:
        #Almacenamos la matriz de distancias y los nombres de las ciudades
        distancias, ciudades = load_distancias()

        if (sys.argv[1] == "heuristico"):
            if (sys.argv[2] == "origen"):
                origen = int(input("Ingrese una ciudad: "))
                while (origen <= 0 or origen > N_CIUDADES):
                    origen = int(input("La ciudad ingresada no existe, ingrese otra:
"))

                camino, dist = metodo_heuristico(origen, distancias)

            elif (sys.argv[2] == "general"):
                camino, dist = heuristico_general(distancias)

            else:
                print(" Opcion Incorrecta ")
                usage()
                exit(1)

        elif (sys.argv[1] == "genetico"):
            cromosoma = metodo_genetico(distancias)
            camino = cromosoma.genes
            dist = cromosoma.dist

        else:
            print(" Opcion Incorrecta ")
            usage()
            exit(1)

        dibujar_ruta(camino, ciudades)
        print("Generacion terminada.")
```

```
        print("Mapa de rutas generado en")
        print(f"file://{os.getcwd()}/ruta.html")
        print("Cantidad de iteraciones: ", N_CICLOS)
        print("Distancia minima: ", dist)
```

Heurístico

```
from genetico import N_CIUDADES
import copy

def print_distancias(distancias):
    for ciudad in distancias:
        for dist in ciudad:
            try:
                print("{:4d}".format(dist), end=" ")
            except Exception as e:
                print(dist)
        print("\n")

#Calcula el mejor camino partiendo de una ciudad de origen dado
def metodo_heuristico(origen, distancias):
    cant_visitadas = 0
    dist_total = 0
    camino: list[int] = [origen]
    dist_origen = distancias[origen].copy() #Guardamos las distancias desde el
origen para no perderlas

    print_distancias(distancias)
    while cant_visitadas < N_CIUDADES:
        min = int(1e9)
        i = 0
        for dist in distancias[origen]:
            if dist < min and dist != 0:
                min = dist
                index = i
            i += 1

        #Actualizamos los valores de la ciudad ya recorrida
        for i in range(N_CIUDADES+1):
            distancias[i][origen] = 0 #Columna
            distancias[origen][i] = 0 #Fila

        camino.append(index)
        origen = index
        dist_total += min
        cant_visitadas += 1

    #Volvemos a la ciudad de origen
```

```
    dist_total += dist_origen[index]
    return camino, dist_total

#Calcula el mejor camino con el método heuristico pero no partiendo de una ciudad
sino de cualquiera
def heuristico_general(distancias):
    min_dist = 1e9
    min_camino = []

    for ciudad in range(N_CIUDADES):
        camino, dist = metodo_heuristico(ciudad, copy.deepcopy(distancias))
        print("origen:", ciudad)
        print(dist, camino)

        if (dist < min_dist):
            min_dist = dist
            min_camino = camino

    return min_camino, min_dist

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

Genético

```
import random
from util import load_distancias

#----- CONSTANTES ------ #
N_CIUDADES = 24
N_CROMOSOMAS = 50
N_CICLOS = 500
P_CROSSOVER = 0.75
P_MUTACION = 0.1
#---------------------- #

#----- DEFINICIONES ---- #
poblacion = []
#---------------------- #

class Cromosoma:
    genes: list = []
    fitness: float = 0.0
    dist: 0

    def __init__(self):
        #random sample elige N_CIUDADES de una lista=[1..N_CIUDADES] sin repeticion
        self.genes = random.sample(range(0, N_CIUDADES), N_CIUDADES)
```

```
        self.fitness = 0.0

    #Calcular la distancia total de un recorrido dado
    def set_dist(self, distancias) -> int:
        sum = 0
        for i in range(1, N_CIUDADES-1):
            sum += distancias[self.genes[i]][self.genes[i+1]]
        self.dist = sum

    def set_fitness(self, total: int):
        #self.fitness = self.dist / total
        #self.fitness = 1 / self.dist
        self.fitness = (total / self.dist) / N_CROMOSOMAS

    def print_gen(self):
        for value in self.genes:
            print("{:02d}".format(value), end=" ")
        print("\n")

    def test_genes(self, ciclo, nro_cromosoma):
        max = len(self.genes)
        for i in range(max):
            if self.genes[i] in self.genes[i+1:max]:
                print_poblacion()
                print(self.genes)
                print("Gen repetido", self.genes[i], i)
                print("Cromosoma: ", nro_cromosoma)
                print("Ciclo: ", ciclo)
                return False

        return True

#Generar la poblacion inicial
def gen_poblacion():
    for _ in range(N_CROMOSOMAS):
        poblacion.append(Cromosoma())

#Mostrar la poblacion actual
def print_poblacion():
    for cromosoma in poblacion:
        cromosoma.print_gen()

#Mostrar el fitness de todos los cromosomas de la poblacion
def print_fitness():
    i = 0
    for cromosoma in poblacion:
        print(i, cromosoma.fitness, cromosoma.dist)
        i += 1
```

```python
#Funcion de crossover, cruza dos padres y obtiene dos hijos
#Utilizamos el método de crossover circular
def crossover(padre: Cromosoma, madre: Cromosoma):
    hijo1, hijo2 = Cromosoma(), Cromosoma()
    hijo1.genes = padre.genes.copy()
    hijo2.genes = madre.genes.copy()

    first = padre.genes[0] #Guardar el primer gen del padre
    salto = N_CIUDADES + 1

    i = 0
    while first != salto: #Mientras no volvamos al mismo gen donde empezamos
        salto = madre.genes[i]
        i = padre.genes.index(salto)

        hijo1.genes[i], hijo2.genes[i] = hijo2.genes[i], hijo1.genes[i]

    return hijo1, hijo2

#Elegimos dos genes al azar e intercambiamos sus valores
def mutacion(cromosoma: Cromosoma):
    indexs = random.sample(cromosoma.genes, 2)
    cromosoma.genes[indexs[0]], cromosoma.genes[indexs[1]] =
cromosoma.genes[indexs[1]], cromosoma.genes[indexs[0]]

#Eliminar los cromosomas con peor fitness hasta dejar N_CROMOSOMAS
def borrar_peores():
    global poblacion

    #Ordenar por fitness de modo creciente
    poblacion.sort(key=lambda C: C.dist)

    poblacion = poblacion[:N_CROMOSOMAS]

#Calcular las distancias de cada cromosoma y su funcion fitness
def fitness_poblacion(distancias):
    total = 0
    for cromosoma in poblacion:
        cromosoma.set_dist(distancias)
        total += cromosoma.dist
    for cromosoma in poblacion:
        cromosoma.set_fitness(total)

#Metodo de seleccion, Ruleta con elitismo
def metodo_seleccion():

    #Los dos mejores individuos pasaran directamente a la siguiente generacion
```

```
    poblacion.sort(key=lambda C: C.fitness, reverse=True)
    #Ponemos el fitness de los dos mejores individuos en 0 para que no puedan ser
seleccionados
    poblacion[N_CROMOSOMAS-1].fitness = 0
    poblacion[N_CROMOSOMAS-2].fitness = 0

    #Seleccionamos N_CROMOSOMAS, los que tengan mejor fitness tendrán más
posibilidades de ser seleccionados
    return random.choices(poblacion, [c.fitness for c in poblacion], k =
N_CROMOSOMAS)

#
def ciclo(distancias):
    global poblacion

    fitness_poblacion(distancias)
    seleccion = metodo_seleccion()

    for i in range(2, N_CROMOSOMAS, 2):
        #hijos = []

        if random.random() < P_CROSSOVER:
            padre = seleccion[i]
            madre = seleccion[i+1]
            hijo1, hijo2 = crossover(padre, madre)
            #hijos.append(hijo1)
            #hijos.append(hijo2)
            #Los hijos se ponen en lugar de los padre
            poblacion[i] = hijo1
            poblacion[i+1] = hijo2

            if random.random() < P_MUTACION:
                mutacion(poblacion[i])

            if random.random() < P_MUTACION:
                mutacion(poblacion[i+1])

        fitness_poblacion(distancias)

def metodo_genetico(distancias):
    gen_poblacion()

    i_ciclo = 0
    while i_ciclo < N_CICLOS:
        ciclo(distancias)
        i_ciclo += 1
    print_poblacion()
    print_fitness()
```

```
    return poblacion[0]

if __name__ == "__main__":
    distancias, ciudades = load_distancias()
    metodo_genetico()
```

Lógica

Búsqueda exhaustiva

La búsqueda exhaustiva es una técnica para resolver problemas en el que se analizan todos los posibles casos utilizando los elementos dados. A pesar de ser la que minimiza al máximo la distancia para el problema dado, también es la que más tarda por tener una complejidad de 23!, lo que equivale a casi 820 millones de años de pruebas a nivel computacional.

El método de búsqueda se basa en un árbol lleno de ramificaciones con todos los casos posibles. Hay muchas maneras de recorrerlo, pero entre las mas conocidas se encuentra el Backtracking y el Branch and Bound. Como nos podemos imaginar, se van a estar analizando casos que van a ser desechados en el futuro casi todo el tiempo.

Por lo antes mencionado, el código si que se puede modelar, pero nunca llegaríamos al resultado al momento de compilarlo. Estos tipos de problemas entran dentro de los llamados NP-Completos debido a que se encuentran en ese grupo de solución perfecta imposible.

Búsqueda heurística

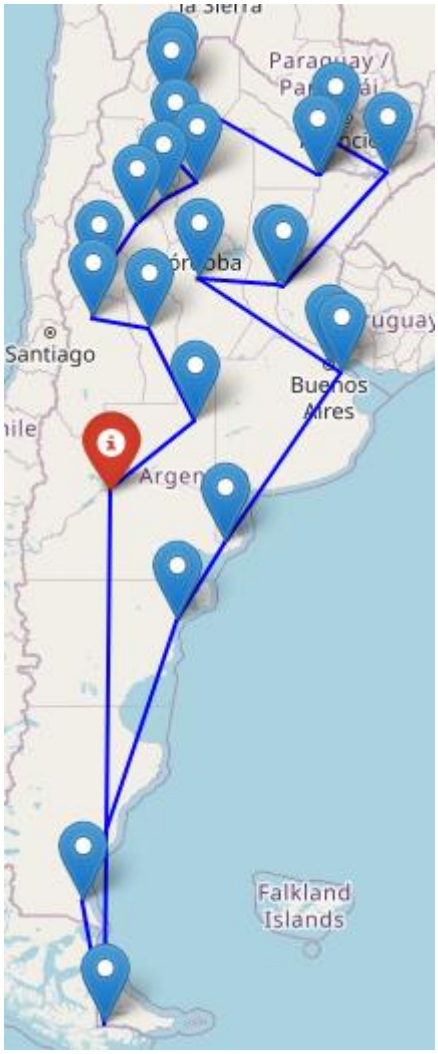
Para resolver el problema por búsqueda heurística hicimos que el recorrido, partiendo desde un punto de partida, se mueva buscando el destino mas cercano NO visitado. Este método fue con la ayuda de las distancias brindadas en el Excel en kilómetros.

Como el enunciado indica, hicimos 2 formatos:

- El usuario brinda una ciudad de partida, el código busca la menor distancia a la próxima ciudad, guarda dicha ciudad (esta, si recorremos la tabla por columnas, va a ser encontrada en la fila relacionada a la mínima distancia) y modifica los valores de dicha ciudad en la tabla igualándolo a 9999 para que así en las próximas comparativas no vuelva a salir (ciudad visitada). Cabe aclarar que como la tabla es simétrica, estos se modifican en la columna como en la fila de la ciudad actual.

  Ejemplo saliendo desde Paraná

- Heurística general analiza los 23 recorridos posibles poniendo como puntos de partida a todas las ciudades y eligiendo entre ellas la del camino más corto. Este caso es para tener datos generales y podamos ver con nuestros propios ojos qué recorrido es mejor que otro.



Por ultimo y para los 2 formatos se muestra en kilómetros la distancia total recorrida

Por Algoritmo Genético

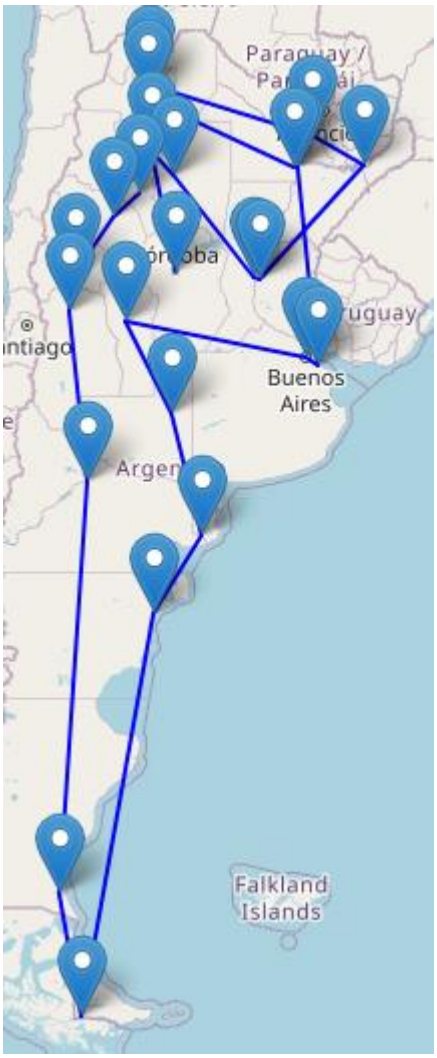
Para resolverlo con este algoritmo elegimos 50 cromosomas de 23 genes cada uno como población inicial y una probabilidad de crossover y de mutación de 75% y de 10% respectivamente. La cantidad de ciclos la centramos en 500 porque consideramos que es cuando menos fluctuaciones de resultados encontramos, pero también se consiguen buenos resultados con menos ciclos.

La población la generamos aleatoriamente, utilizamos un crossover cíclico y una mutación de intercambio.

Previo al crossover y a la mutación hicimos que por cada cromosoma calcule la distancia, así como el complemento sobre el total de dicha distancia, mejor llamada fitness.

El método de selección que usamos es el de Ruleta con elitismo en el cual va eligiendo los que mejores fitness tengan.

Recorrido de un camino de 7640km



Conclusiones

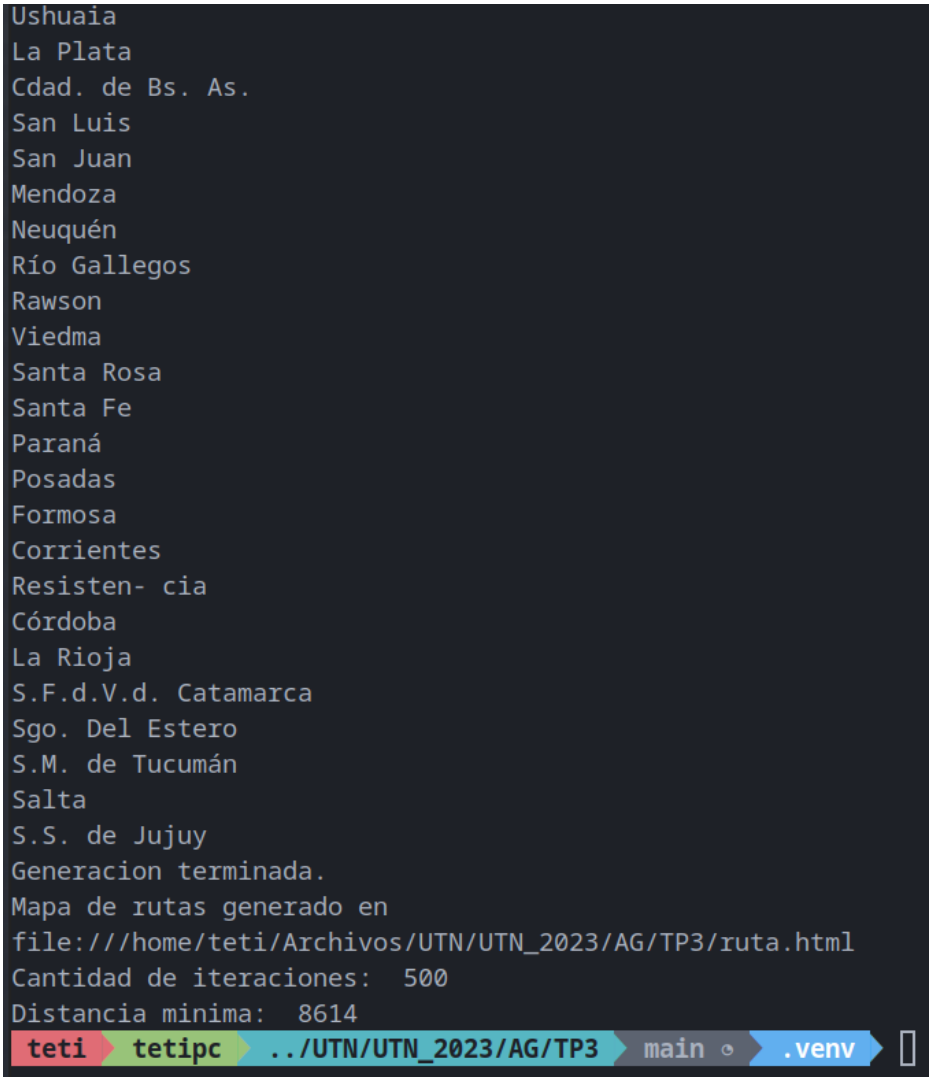
Como pudimos observar comparando los 3 métodos, el que mejores resultados daría es el exhaustivo. Por razones mencionadas anteriormente, este método es imposible de aplicar con 23 ciudades (con menos si sería posible) lo que nos da pie a comparar los otros 2 métodos para sacar unas conclusiones acordes.

A pesar de que uno podría imaginar que la búsqueda heurística es la mejor, esta puede desviarse por un camino que no es optimo al momento de volver o de ir a la próxima ciudad. Por consiguiente, el método que mejores resultados nos dio es en el que utilizamos AG.

Como extra podemos decir que el AG puede ser utilizado en muchos casos y no necesariamente en el problema del viajante. También tiene una ventaja frente al heurístico y es que mejoran aún más los resultados a medida que se aumentan los ciclos.

Mejor recorrido con búsqueda heurística

Mejor recorrido usando AG



Después de probar con todos los orígenes utilizando el heurístico general pudimos encontrar que el mejor recorrido es saliendo desde Neuquén con 9300km. Arriba agregamos una foto con una de las pruebas usando AG y se puede comprobar que el mejor recorrido luego de 500 ciclos es de 8600km. No obstante, estos oscilan entre 7500km y 10000km así que no siempre se obtienen mejores resultados que en el heurístico

Aportes prácticos de TSP

Problema de la red de basuras

El problema de la gestión de residuos se puede categorizar en tres principales ámbitos: residuos domésticos, comerciales e industriales. La recolección de residuos domésticos se centra principalmente en atender a hogares individuales, y la frecuencia de recolección puede variar, aunque generalmente se realiza una vez al

día. En el caso de la recolección de residuos comerciales e industriales, se encarga de recoger los desechos de tiendas, restaurantes y edificios comerciales o industriales. Los objetivos de estos problemas pueden ser diversos, como minimizar el número de camiones utilizados o la distancia total recorrida.

Para abordar el problema de minimizar la distancia recorrida en la recolección de residuos, se puede utilizar el enfoque del "problema del viajante de comercio". En este enfoque, se identifican los contenedores de basura o los puntos de recogida como las ciudades a visitar, y se busca encontrar la ruta óptima que permita al recolector visitar todos estos puntos de manera eficiente. Además, estos mismos principios se pueden aplicar en otros contextos, como en empresas de transporte, servicios postales y logística en general, para optimizar las rutas de entrega y minimizar los costos asociados.

Problema de placa de circuitos impresos PCB

El problema de la placa de circuitos impresos, o PCB, plantea desafíos interesantes que se pueden abordar mediante el problema del viajante. Este problema se divide en dos aspectos clave:

1. **Orden óptimo de perforación de placas:** En este subproblema, consideramos las ubicaciones en las placas que deben perforarse como si fueran ciudades en un mapa. Las distancias entre estas ubicaciones representan el tiempo que una máquina de perforación necesita para moverse de un punto a otro. También se incluyen puntos de inicio y finalización donde la máquina permanece inactiva. Es esencial programar eficientemente estas máquinas, ya que el tiempo que demoren en perforar un orificio en comparación con otro puede tener un impacto significativo en la eficiencia de la producción de placas de circuito impreso.
2. **Conexión óptima de chips:** En este subproblema, el objetivo es minimizar la cantidad de cable necesario para conectar todos los puntos (o pines) en una placa de circuito sin que se produzcan interferencias. Dado que los chips son de tamaño limitado y no se pueden usar más de dos cables en un solo pin, podemos abordar este problema considerando los pines como ciudades y la longitud de cable requerida para conectarlos como la distancia entre ellas. En este caso, estamos tratando esencialmente el problema como una variante del problema del viajante de comercio.

Ambos subproblemas relacionados con la creación de placas de circuitos impresos se benefician de la optimización de rutas y distancias, lo que puede mejorar la eficiencia de la producción y la conectividad de los circuitos sin desperdiciar recursos como tiempo y material.