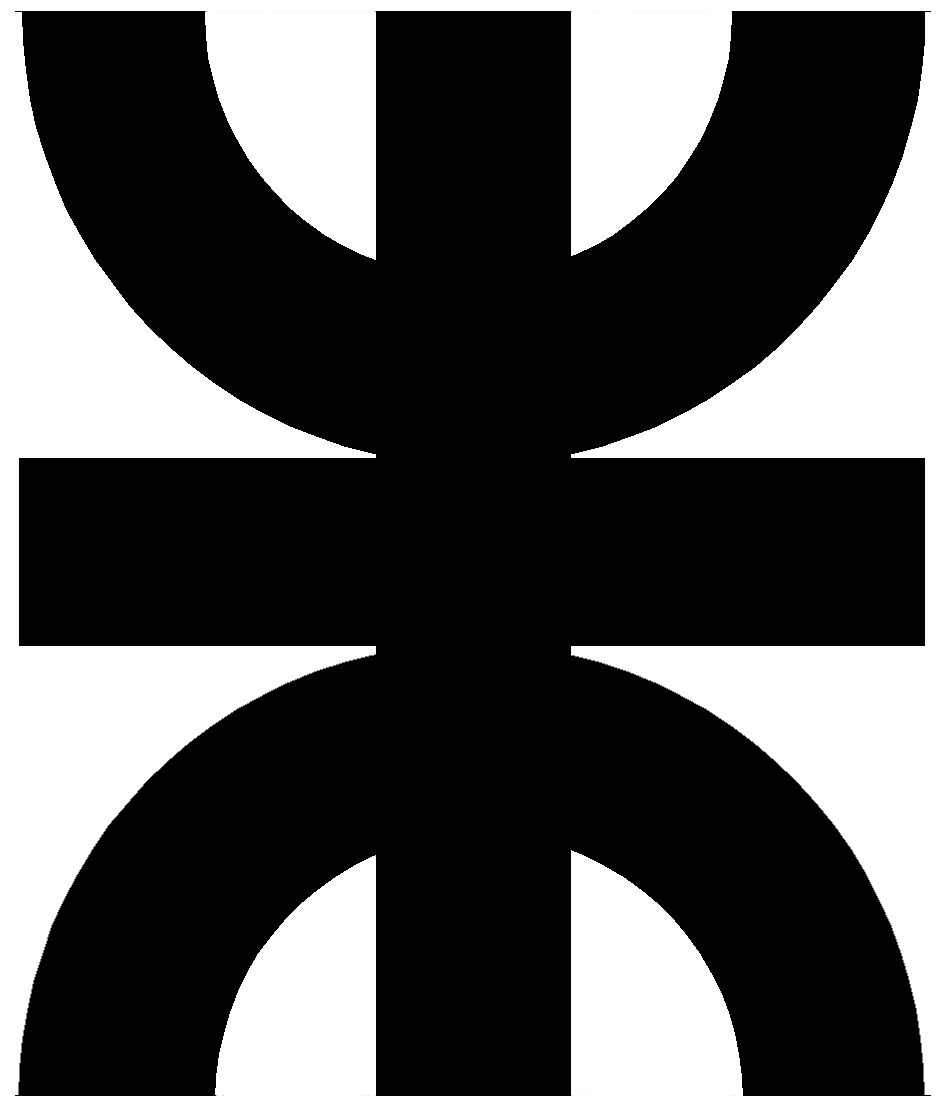
TRABAJO PRÁCTICO Nº 3

Cátedra: Algoritmos Genéticos



Integrantes: Legajos:

Albizuri, Gastón 40412

Belletti, Kristal 40568

Giordano, Nicolás 40467

**UNIVERSIDAD TECNOLÓGICA NACIONAL**

**FACULTAD REGIONAL ROSARIO**

**INGENIERÍA EN SISTEMAS DE INFORMACIÓN**

Índice

**Pág.**

**Introducción** **1**

**Problemática** **2**

**Resolución** **3**

**Código Fuente** **8**

**Conclusiones** **16**

**Aportes prácticos del TSP**…………………………………………………………………………..**18**

Introducción

El Problema del Viajante (o The Traveling Salesman Problem - TSP), consiste en encontrar la ruta que lleva a cabo un vendedor, que comenzando por un origen visita un determinado y preestablecido conjunto de ciudades y vuelve a ese origen, de manera que la distancia recorrida total es mínima y cada ciudad sólo se visita una vez.

El TSP es uno de los más típicos y mejor estudiados problemas de la optimización combinatoria, pero a pesar de ello aún no se conocen algoritmos que lo resuelvan de forma exacta. Los algoritmos utilizados para su resolución sólo ofrecen aproximaciones, y la solución óptima sólo es alcanzable para casos concretos del problema.

Dentro de esta optimización combinatoria de la cual el TSP es objeto de estudio, se le cataloga como un problema NP-completo.

Problemática

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.
2. Hallar la ruta de distancia mínima que logre unir todas las capitales de provincias de la República Argentina, 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 y la longitud del trayecto.
3. 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.

Resolución

1. **Método Exhaustivo:**

El problema no puede resolverse mediante el método exhaustivo; no se disponen los recursos computacionales para poder ejecutar y resolver dicha problemática.

El TSP se lo cataloga como problema NP-Completo, es decir, que el esfuerzo computacional para encontrar una solución óptima (uso del método exhaustivo) crece de forma exponencial con la entrada del problema, que en el caso concreto de TSP sería el número de ciudades; por lo tanto este número, va a ser fundamental en determinar la complejidad del problema:

Cuanto mayor sea el número de ciudades, mayor va a ser el número de rutas posibles, y por lo tanto mayor será el esfuerzo requerido para calcular todas ellas. Así, el número de rutas posibles entre N ciudades va a ser igual a N!, lo que hace que la resolución del TSP mediante la obtención de todas las rutas posibles y comparación entre ellas sea poco factible incluso para un número de nodos no elevado.

Por esta razón, la resolución del TSP mediante el método exhaustivo, es decir, el método de obtención y comparación de todas las rutas es absolutamente inabordable mediante los medios computacionales disponibles actualmente.

**2. Búsqueda heurística:**

Para la resolución del problema con esta búsqueda, utilizamos una función, que toma el índice de la ciudad de inicio elegida por el usuario, crear el trayecto a partir de la heurística propuesta (“*Desde cada ciudad ir a la ciudad más cercana no visitada”*) y acumula la distancia recorrida.

Al comenzar el viaje, se inicializan el recorrido (la primer ciudad, la de inicio, corresponde al índice de la ciudad que el usuario determinó como inicial) y el acumulador del trayecto (el contador se pone en cero).

Luego, comienza la parte de la heurística: se recorre el arreglo de distancias, se busca la menor distancia que existe entre la ciudad de la que se parte con respecto al resto de ciudades, la menor distancia encontrada, corresponde a la ciudad a la cual vamos a movernos y así sucesivamente pasando de ciudad de ciudad, índice a índice (aplicando la heurística); llegado el caso, se valida que la menor distancia no corresponda a una ciudad ya visitada (se comprueba que la distancia menor entre una ciudad y otra, no corresponda con una ciudad que ya se haya visitado).

**RESULTADOS:**

Si ingresamos que la ciudad de origen sea Santa Fe, el resultado obtenido, aplicando la heurística, es el siguiente:



***Resultado en pantalla:*** *Recorrido en mapa de la República Argentina*

**3. Algoritmo Genético:**

El algoritmo genéticonecesita una codificación o representación del problema, que resulte adecuada al mismo. Como ya hemos realizado en el Trabajo Práctico N° 1, consiste de las siguientes etapas:

1. Generar una población aleatoria de n individuos con una codificación de r-bits.
2. Calcular la aptitud de cada individuo.
3. Seleccionar a los individuos probabilísticamente en base a su aptitud.
4. Aplicar operadores genéticos (cruza y mutación) para generar la siguiente población.

Se repiten las etapas 2, 3, y 4 hasta que se cumpla cierto número m de generaciones.

Para este problema, tenemos las siguientes características:

* n = 50 cromosomas (individuos) que forman nuestras poblaciones.
* m = 200 Cantidad de ciclos.
* r = 23 números naturales del 0 al 22, sin repetir donde cada gen es una ciudad.
* Método de selección: ruleta.
* Crossover: Cíclico
* Mutación: De Intercambio.
* Probabilidad de Crossover de 0.75 y Probabilidad de Mutación de 0.05

**PASOS**

1. **Codificación y Decodificación:** Para este problema, los cromosomas (individuos de la población) son generados de manera aleatoria: Se generan los números del 0 al 22 y se ordenan aleatoriamente, para no repetir ningún número, (únicos, sin repetir para cada cromosoma) para cada individuo de la población (es decir, se repite esto 50 veces). No se utiliza codificación binaria.
2. **Evaluación en la función:** Se calcula la distancia acumulada de cada cromosoma. Es decir, se toma un cromosoma, se recorre completo, como si fuera un recorrido, se parte del primer gen (ciudad origen), se mueve al siguiente, obteniendo la distancia a dicha ciudad (gen), y se va acumulando a medida que se sigue recorriendo el cromosoma. Por lo tanto, para cada población, vamos a tener 50 distancias totales recorridas, cada una correspondiendo a un cromosoma/individuo de la población.
3. **Fitness de cada individuo:** A partir de la sumatoria de todas las distancias obtenidos de haber evaluado cada cromosoma en la función, se calcula para cada individuo su Fitness: en este caso, como intentamos **minimizar** nuestra función objetivo (distancia / recorrido), los valores del Fitness para cada individuo deben determinarse teniendo en cuenta que el menor recorrido tenga mayor Fitness que uno que tenga mayor recorrido. Por lo tanto, hemos partido del siguiente ejemplo, aplicando el concepto de complemento:

*DISTANCIAS: 2 – 3 – 5*

*TOTAL = 10*

*FITNESS: 0,2 – 0,2 – 0,5*

*Como podemos observar, los valores de Fitness calculados, muestran que la aptitud para una distancia de 5 es mayor que para una de 2. Aplicando el concepto, los genes que tienen mayor aptitud se harán más comunes, es decir, habrá más distancias de 5 que de 2 o 3; pero nosotros no queremos eso, por lo que debemos darle mayor frecuencia a las distancias menores: la distancia 2 debe ser más frecuente que la 3, y ésta más que la de 5.*

*Por lo que usamos, el concepto de complemento:*

*COMPLEMENTO: TOTAL – DISTANCIA: 8 – 7 – 5*

*TOTAL = 20*

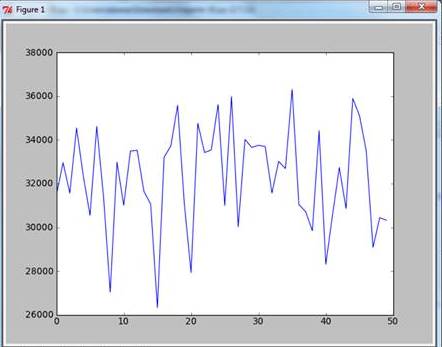
*FITNESS RESULTANTE: 0,4 – 0,35 – 0,25*

*Así, la distancia de 2 va a ser más común (más frecuente, tiene mayor probabilidad) que la distancia de 3 y de 5.*

1. **Ruleta:** Para construirla, a cada cromosoma se le asigna un arco de circunferencia igual a su Fitness: el individuo uno tendrá un arco de 0 a fitness[0]; el indivudo dos un arco de fitness[0] a fitness[1] y asi sucesivamente hasta el individuo 50 que tendrá desde fitness[49] a 1.
2. **Selección por ruleta:** Giramos la ruleta 2 veces y guardamos los 2 cromosomas seleccionados, estos cromosomas son los que van a ser sometidos al crossover y la mutación. Éste proceso se repite 25 veces.
3. **Crossover (Cruce cíclico):** Se consulta la probabilidad de crossover (determinada por un número aleatorio generado): si no supera dicha frecuencia habrá cruce, en cambio, si supera, no hay cruza y dicho par pasa a la población siguiente sin alterarse. En el caso de haber cruza, se sigue un **cruce cíclico**.
4. **Mutación de Intercambio Recíproco:** Con una frecuencia de mutación de 0.05, se examina cada individuo hijo, generamos un número aleatorio, si dicho número está por debajo de la probabilidad de mutación, se eligen dos ciudades aleatoriamente (se generan dos números aleatorios comprendidos entre 0 y 22, que corresponden a dos índices del individuo, dos genes de un cromosoma) y se intercambian.

**RESULTADOS:**

Por ejemplo, con una frecuencia de cruce de 75% y una de mutación de 5%, estos son los resultados obtenidos:



***Menores distancias recorridas.***

Código Fuente

|  |
| --- |
|  |
|  | From turtle import \*  from Tkinter import \* |
|  | import random |
|  | import ctypes |
|  | from random import randint, random, shuffle |
|  | import matplotlib.pyplot as plt |
|  |  |
|  |  |
|  |  |
|  | #bloque def constantes |
|  | distancia = [] |
|  | puntos = [] |
|  |  |
|  | distancia.append([0,1543,1510,1203,1043,1191,1023,478,940,1040,480,715,1150,1110,790,1155,1050,620,1158,960,1455,2635,3228]) |
|  | distancia.append([1543,0,99,340,500,960,860,1107,883,1198,1138,930,770,1220,1320,572,1345,1530,2200,2124,2385,3565,4158]) |
|  | distancia.append([1510,99,0,307,467,948,780,1074,803,1118,1105,897,695,1145,1245,539,1227,1497,2082,2091,2352,3532,4125]) |
|  | distancia.append([1203,340,307,0,160,936,768,767,791,1106,798,590,338,838,938,232,1005,1190,1860,1784,2045,3225,3818]) |
|  | distancia.append([1043,500,467,160,0,776,610,607,633,948,638,430,360,810,850,212,977,1030,1567,1624,1885,3065,3658]) |
|  | distancia.append([1191,960,948,936,776,0,168,713,191,506,744,1043,1136,1543,1463,988,1710,1523,2060,2117,2378,3558,4151]) |
|  | distancia.append([1023,860,780,765,610,168,0,545,23,338,576,875,970,1420,1295,822,1587,1475,2012,2069,2210,3390,3983]) |
|  | distancia.append([478,1107,1074,767,607,713,545,0,568,883,31,330,765,830,625,770,885,810,1347,1404,1665,2845,3438]) |
|  | distancia.append([940,883,803,791,633,191,23,568,0,315,590,898,993,1398,1318,845,1565,1378,1989,2046,2187,3367,3960]) |
|  | distancia.append([1040,1198,1118,1106,948,506,338,883,315,0,820,1213,1308,1758,1633,1160,1925,1660,2198,2000,2495,3675,4268]) |
|  | distancia.append([480,1138,1105,798,638,744,576,31,590,820,0,361,796,861,656,801,916,841,1378,1435,1696,2876,3469]) |
|  | distancia.append([715,930,897,590,430,1043,875,330,898,1213,361,0,435,500,420,440,670,600,1137,1194,1455,3635,3228]) |
|  | distancia.append([1150,770,695,388,360,1136,970,765,993,1308,796,435,0,450,550,156,617,1035,1472,1629,1890,3070,3663]) |
|  | distancia.append([1110,1220,1145,838,810,1543,1420,830,1398,1758,861,500,450,0,320,606,167,825,1022,1419,1680,2860,3453]) |
|  | distancia.append([790,1320,1245,938,850,1463,1295,625,1318,1633,656,420,550,320,0,705,260,505,883,1099,1360,2540,3133]) |
|  | distancia.append([1145,572,539,232,212,988,822,770,845,1160,801,440,156,606,705,0,773,1040,1588,1634,1895,3075,3668]) |
|  | distancia.append([1050,1345,1227,1005,977,1710,1587,885,1565,1925,916,670,617,167,260,773,0,765,855,1359,1620,2800,3393]) |
|  | distancia.append([620,1530,1497,1190,1030,1523,1475,810,1378,1660,841,600,1035,825,505,1040,765,0,537,594,855,2035,2628]) |
|  | distancia.append([1158,2200,2082,1860,1567,2060,2012,1347,1989,2198,1378,1137,1472,1022,883,1588,855,537,0,660,750,1930,2523]) |
|  | distancia.append([960,2124,2091,1784,1624,2117,2069,1404,2046,2000,1435,1194,1629,1419,1099,1634,1359,594,660,0,495,1675,2268]) |
|  | distancia.append([1455,2385,2352,2045,1885,2378,2210,1665,2187,2495,1696,1455,1890,1680,1360,1895,1620,855,750,495,0,1180,1773]) |
|  | distancia.append([2635,3565,3532,3225,3065,3558,3390,2845,3367,3675,2876,2635,3070,2860,2540,3075,2800,2035,1930,1675,1180,0,593]) |
|  | distancia.append([3228,4158,4125,3818,3658,4151,3983,3438,3960,4268,3469,3228,3660,3453,3133,3668,3393,2628,2523,2268,1773,593,0]) |
|  |  |
|  | puntos.append([36,69.5]) |
|  | puntos.append([-56,239.5]) |
|  | puntos.append([-57.1,228.5]) |
|  | puntos.append([-54,194.5]) |
|  | puntos.append([-44,180.5]) |
|  | puntos.append([50,206.5]) |
|  | puntos.append([34,189.5]) |
|  | puntos.append([9,118.5]) |
|  | puntos.append([42,187.5]) |
|  | puntos.append([85,184.5]) |
|  | puntos.append([14,115.5]) |
|  | puntos.append([-40,123.5]) |
|  | puntos.append([-77,150.5]) |
|  | puntos.append([-101,118.5]) |
|  | puntos.append([-68,88.5]) |
|  | puntos.append([-64,171.5]) |
|  | puntos.append([-103,96.5]) |
|  | puntos.append([-39,35.5]) |
|  | puntos.append([-91,-4.5]) |
|  | puntos.append([-23,-34.5]) |
|  | puntos.append([-50,-71.5]) |
|  | puntos.append([-86,-210.5]) |
|  | puntos.append([-70,-258.5]) |
|  |  |
|  | #bloque funciones |
|  |  |
|  | def capitalInicial(): |
|  | print("Lista de ciudades\n" |
|  | " 1- Buenos Aires \n 2- San S. de Jujuy \n 3-Salta \n 4- S.M. de Tucuman\n" |
|  | " 5- Sgo. del Estero \n 6- Formosa \n 7-Resistencia \n 8- Santa Fe\n" |
|  | " 9- Corrientes \n 10- Posadas \n 11- Parana \n 12- Cordoba \n 13- La Rioja \n" |
|  | " 14- San Juan \n 15- San Luis \n 16- Catamarca \n 17- Mendoza \n 18- Santa Rosa \n" |
|  | " 19- Neuquen \n 20- Viedma \n 21- Rawson \n 22- Rio Gallegos \n 23- Ushuaia ") |
|  | ciudadActual = input("Seleccione la ciudad de origen: ")-1 |
|  | return menorRecorrido(ciudadActual) |
|  |  |
|  | def SinCapitalInicial(): |
|  | kilometros = [] |
|  | for x in xrange(22): |
|  | recorrido = menorRecorrido(x) |
|  | kms = contarKilometros(recorrido) |
|  | kilometros.append(kms) |
|  | num = min(kilometros) |
|  | index = kilometros.index(num) |
|  | return menorRecorrido(index) |
|  |  |
|  | def geneticos(): |
|  | inicial = [[0 for c in range(23)] for y in range(50)] |
|  | auxiliar = [[0 for c in range(23)] for y in range(50)] |
|  | for h in xrange(200): |
|  | #genero poblacion inicial y calculo recorrido total |
|  | total = 0; |
|  | for c in xrange(50): |
|  | for y in xrange(23): |
|  | inicial[c][y] = y |
|  | shuffle(inicial[c]) |
|  | total = total + contarKilometros(inicial[c]) |
|  | #acumulo los complementos |
|  | totalCompl = 0 |
|  | for c in xrange(50): |
|  | totalCompl = totalCompl + (total - contarKilometros(inicial[c])) |
|  | #calculo el fitness de cada individuo |
|  | fitness = [] |
|  | for c in xrange(50): |
|  | obj = contarKilometros(inicial[c]) / float(totalCompl) |
|  | fit = 1 - obj |
|  | fitness.append(fit) |
|  | #tiro la ruleta 25 veces, por pares |
|  | for m in xrange(25): |
|  | sumFitness = 0 |
|  | tirada1 = random() |
|  | for n in xrange(50): |
|  | if((tirada1 > sumFitness) & (tirada1 < (sumFitness + fitness[n]))): |
|  | elegido1 = n |
|  | break |
|  | else: |
|  | sumFitness = sumFitness + fitness[n] |
|  |  |
|  | sumFitness = 0 |
|  | tirada2 = random() |
|  | for n in xrange(50): |
|  | if((tirada2 > sumFitness) & (tirada2 < (sumFitness + fitness[n]))): |
|  | elegido2 = n |
|  | break |
|  | else: |
|  | sumFitness = sumFitness + fitness[n] |
|  | crom1 = inicial[elegido1] |
|  | crom2 = inicial[elegido2] |
|  | #crossOver |
|  | rnd = random() |
|  | if(rnd < 0.75): |
|  | estan = [] |
|  | estan.append(crom1[0]) |
|  | comparador = crom2[0] |
|  | i=0 |
|  | while(comparador != crom1[0]): |
|  | if(comparador is not estan): |
|  | if(comparador == crom1[i]): |
|  | estan.append(crom1[i]) |
|  | comparador = crom2[i] |
|  | if(i>=22): |
|  | i=0 |
|  | else: |
|  | i=i+1 |
|  |  |
|  | for z in xrange(23): |
|  |  |
|  | if(crom1[z] is not estan): |
|  | aux = [] |
|  | aux = crom1[z] |
|  | crom1[z] = crom2[z] |
|  | crom2[z] = aux |
|  | #mutacion |
|  | rnd1 = random() |
|  | if(rnd1 < 0.05): |
|  | pos1 = randint(0,22) |
|  | pos2 = randint(0,22) |
|  | aux = crom1[pos1] |
|  | crom1[pos1] = crom1[pos2] |
|  | crom1[pos2] = aux |
|  | rnd2 = random() |
|  | if(rnd2 < 0.05): |
|  | pos1 = randint(0,22) |
|  | pos2 = randint(0,22) |
|  | aux = crom2[pos1] |
|  | crom2[pos1] = crom2[pos2] |
|  | crom2[pos2] = aux |
|  | auxiliar[2\*m] = crom1 |
|  | auxiliar[2\*m+1] = crom2 |
|  |  |
|  | return inicial |
|  |  |
|  |  |
|  | def menorRecorrido(ciudadActual): |
|  | visitadas = [] |
|  | visitadas.append(ciudadActual) |
|  | aux = [] |
|  | aux = list(distancia[ciudadActual]) |
|  | aux.pop(ciudadActual) |
|  | minimo = min(aux) |
|  | ciudadActual = distancia[ciudadActual].index(minimo) |
|  |  |
|  | for x in xrange(22): |
|  | visitadas.append(ciudadActual) |
|  | aux = [] |
|  | aux = list(distancia[ciudadActual]) |
|  |  |
|  | for z in visitadas: |
|  | aux.remove(distancia[ciudadActual][z]) #elimino ciudades visitadas de la lista auxiliar |
|  |  |
|  | if(len(aux) != 0): |
|  | minimo = min(aux) |
|  | indice = distancia[ciudadActual].index(minimo) #guardo el indice del minimo |
|  | ciudadActual = indice |
|  | return visitadas |
|  |  |
|  | def contarKilometros(resultado): |
|  | cont = 0 |
|  | for i in xrange(22): #contar el recorrido total |
|  | cont = cont + distancia[resultado[i]][resultado[i+1]] |
|  | cont = cont + distancia[resultado[22]][resultado[0]] |
|  | return cont |
|  |  |
|  | def dibujar(recorrido): |
|  | setup(400,627,0,0) |
|  | bgpic("350px-Argentina.gif") |
|  |  |
|  | penup() |
|  |  |
|  | goto(puntos[recorrido[0]][0],puntos[recorrido[0]][1]) |
|  | dot(10, "blue") |
|  | pendown() |
|  | speed(1) |
|  | pencolor("lightblue") |
|  | pensize(3) |
|  |  |
|  | for x in xrange(1,len(recorrido)): |
|  | goto(puntos[recorrido[x]][0],puntos[recorrido[x]][1]) |
|  | dot(10, "blue") |
|  | goto(puntos[recorrido[0]][0],puntos[recorrido[0]][1]) |
|  | done() |
|  |  |
|  | def imprimirResultados(resultado): |
|  | cont = contarKilometros(resultado) |
|  | dibujar(resultado) |
|  | ctypes.windll.user32.MessageBoxA(0, "Recorrido total: "+str(cont)+"km", "Mapa de Argentina", 1) #ventana muestra recorrido total |
|  |  |
|  | #programa Principal |
|  | opc = 4 |
|  | while(opc >= 3 or opc <=1): |
|  | print("Bienvenido al sistema del Viajante, el sistema recorrera las distintas capitales de la republica argentina y devolvera el menor recorrido") |
|  | print("Seleccione la opcion deseada") |
|  | print(" 1- Seleccionando Capital de inicio \n 2- Distancia minima por Heuristico \n 3- Distancia minima por Algoritmos Geneticos") |
|  | opc = input() |
|  |  |
|  | if(opc == 1): |
|  | resultado = capitalInicial() |
|  | imprimirResultados(resultado) |
|  |  |
|  | if(opc == 2): |
|  | resultado = SinCapitalInicial() |
|  | imprimirResultados(resultado) |
|  | if(opc == 3): |
|  | totales = [] |
|  | resultado = geneticos() |
|  | for x in xrange(50): |
|  | print("Recorrido: ") |
|  | print(resultado[x]) |
|  | print("Distancia Recorrida: ") |
|  | print(contarKilometros(resultado[x])) |
|  | totales.append(contarKilometros(resultado[x])) |
|  |  |
|  | plt.plot(totales) |
|  |  |
|  | plt.show() |
|  |  |
|  | opc=4 |

Conclusiones

El Problema del Agente Viajero o del Viajante (TSP) es un problema cuya solución ha sido estudiada desde los inicios de la Inteligencia Artificial considerando que su aplicación puede ser en cualquier área de estudio cuyos problemas reflejen una situación donde se tienen diferentes puntos a visitar con un costo considerado en el enlace entre dichos puntos (costo, recursos empleados como distancia, tiempo, monto económico, etc.).

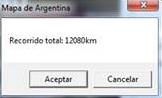
La esencia de esta búsqueda es encontrar el mínimo de las características totales (distancia, en nuestro caso), y el viajante tiene que pasar por todas las 23 ciudades una vez, y volver a la ciudad con la que empezó.

Para este trabajo, se han tratado tres (3) formas de resolver este problema, cada una con una perspectiva diferente empleando técnicas distintas: mediante el método exhaustivo, con búsqueda heurística y a través de un algoritmo genético.

Como hemos demostrado, la resolución del TSP mediante el método exhaustivo, el método de obtención y comparación de todas las rutas posibles, es absolutamente inabordable mediante los medios computacionales disponibles actualmente.

Lo que hace que sea necesario utilizar otras vías que aunque no obtengan la solución óptima sí que ofrecen una respuesta aproximada lo suficientemente OPTIMIZADA: esta vía se identifica con la búsqueda heurística.

A partir de la heurística planteada, podemos decir que dependiendo de la ciudad de la que se salga, se genera una única ruta que cumpla con esa esencia (heurística), por lo tanto, hemos descubierto que existen 23 rutas: de todas esas 23 rutas, la menor es la que corresponde a la que comienza en Neuquén con 12080 km. A partir de este análisis, podemos decir que este recorrido, es una solución factible aproximada a la posible solución óptima (y mejor) del problema.

***Presentación gráfica del menor recorrido de todos los posibles (Neuquén) en la resolución con búsqueda heurística.***

Conclusiones

El Problema del Agente Viajero o del Viajante (TSP) es un problema cuya solución ha sido estudiada desde los inicios de la Inteligencia Artificial considerando que su aplicación puede ser en cualquier área de estudio cuyos problemas reflejen una situación donde se tienen diferentes puntos a visitar con un costo considerado en el enlace entre dichos puntos (costo, recursos empleados como distancia, tiempo, monto económico, etc.).

La esencia de esta búsqueda es **encontrar el mínimo de las características totales (distancia, en nuestro caso)**, y el viajante tiene que **pasar por todas las 23 ciudades una vez**, y volver a la ciudad con la que empezó.

Para este trabajo, se han tratado **tres (3) formas de resolver este problema**, cada una con una perspectiva diferente empleando técnicas distintas: mediante el **método exhaustivo**, con **búsqueda heurística** y a través de un **algoritmo genético**.

Como hemos demostrado, la resolución del TSP mediante el **método exhaustivo**, el método de obtención y comparación de todas las rutas posibles, es **absolutamente inabordable** mediante los medios computacionales disponibles actualmente.

Lo que hace que sea necesario utilizar **otras vías** que aunque no obtengan la solución óptima sí que **ofrecen una respuesta aproximada** lo suficientemente **OPTIMIZADA**: esta vía se identifica con la **búsqueda heurística**.

A partir de la heurística planteada, podemos decir que dependiendo de la ciudad de la que se salga, se genera una única ruta que cumpla con esa esencia (heurística), por lo tanto, hemos descubierto que existen 23 rutas: de todas esas 23 rutas, la menor es la que corresponde a la que comienza en **Neuquén** con **12080 km**. A partir de este análisis, podemos decir que este recorrido, es una solución factible aproximada a la posible solución óptima (y mejor) del problema.

Por último, aplicando un **algoritmo genético** como el que hemos desarrollo podemos concluir que: al ser un método de búsqueda dirigida basada en la **probabilidad**, siguiendo una condición muy débil (que el algoritmo guarde siempre al mejor elemento de la población sin hacerle ningún cambio) se puede demostrar que el algoritmo **converge en probabilidad al óptimo**.

Sin embargo, a través de la experiencia, pudimos observar que esta convergencia al óptimo, **depende de la probabilidad de cruce y mutación**: mientras el operador cruza tiene una tasa o probabilidad elevada en todo algoritmo genético, el parámetro probabilidad de mutación tiene generalmente valores menos significativos: variando dichos parámetros pudimos observar como convergía a ciertos valores posibles.

Aunque a esto debemos agregarle otra dificultad, la **aleatoriedad**, A la hora tirar la ruleta, probar si hay cruza y si hay mutación, resolvemos estos problemas con la **generación de números aleatorios**.

Es por ello, que durante los ciclos de ejecución podemos notar que ciertos puntos no siguen la regularidad o uniformidad del resto de la población, esto se debe a la aleatoriedad en la generación de los números: pueden cruzar muchas veces seguidas, al igual que mutar.

Aportes prácticos del TSP.

Aplicaciones del TSP:

El TSP tiene varias aplicaciones, incluso en su más pura formulación, como la planificación, la logística y la fabricación de microchips. Ligeramente modificado aparece como un sub-problema en muchas áreas, como en la secuenciación del ADN. En estas aplicaciones, la ciudad representa el concepto, por ejemplo, los clientes, puntos de soldadura, o fragmentos de ADN, y el concepto distancia representa los tiempos de viaje o el costo, o una medida de similitud entre partes de ADN. En muchas aplicaciones, las restricciones adicionales, como los recursos limitados o los límites de tiempo hacen que el problema sea considerablemente más difícil. También los cambios de las distancias mientras se va moviendo el viajante complicarían bastante el problema.

TSP se puede emplear en cualquier situación que requiere seleccionar nodos en cierto orden que reduzca los costos:

* Reparto de productos. Mejorar una ruta de entrega para seguir la más corta.
* Transporte. Mejorar el recorrido de caminos buscando la menor longitud (figura 1).
* Robótica. Resolver problemas de fabricación para minimizar el número de desplazamientos al realizar una serie de perforaciones en un circuito impreso.
* Turismo y agencias de viajes. Aun cuando los agentes de viajes no tienen un conocimiento explícito del Problema del Agente Viajero, las compañías dedicadas a este giro utilizan un software que hace todo el trabajo. Estos paquetes son capaces de resolver instancias pequeñas del TSP.
* Horarios de transportes laborales y/o escolares. Estandarizar los horarios de los transportes es claramente una de sus aplicaciones, tanto que existen empresas que se especializan en ayudar a las escuelas a programarlos para optimizarlos en base a una solución del TSP.
* Inspecciones a sitios remotos. Ordenar los lugares que deberá visitar un inspector en el menor tiempo.
* Secuencias. Se refiere al orden en el cual n trabajos tienen que ser procesados de tal forma que se minimice el costo total de producción.

Algunas de las aplicaciones más interesantes de este problema son:

* **Problema de Scheduling:**

Este problema es realmente complejo de resolver. Se formula de la siguiente forma: hay T tareas que realizar y m procesadores. Se busca una planificación en m procesadores para T minimizando el tiempo. Si ahondamos un poco más en este tipo de problemas, observamos que existen una gran cantidad de variantes asociadas a si hay orden parcial en T (es decir, algunas actividades son preferentes a otras)...

En este caso las actividades pueden verse como las ciudades y los tiempos como las distancias. El objetivo es determinar una secuencia (schedule) de tal forma que se lleven a cabo en una cantidad mínima de tiempo. Para resolverlo se usa la versión asimétrica del TSP, es decir, debemos tener en cuenta que el costo de pasar de la tarea A a la B, puede ser diferente del costo de recorrer el camino inverso. Como aplicación más cercana, los problemas de scheduling los encontramos en la televisión, donde es necesario planificar un secuenciamiento óptimo de los comerciales durante un corte publicitario.

* **Problema de placa de circuitos impresos PCB.**

Ésta es sin duda una de las utilidades más ingeniosas que puede plantear el problema del viajante de comercio: la creación de placas de circuitos. Este problema se enfoca en dos suproblemas: el orden óptimo de taladrar las placas y los caminos óptimos que comunican los chips. En los **problemas de perforado** hemos de tomar las ciudades como las posiciones a perforar y las distancias entre ellas como el tiempo que necesita la máquina en trasladarse de una otra. El punto inicial y final será un punto adicional donde permanece la perforadora mientras descansa. Claramente si estas máquinas no son correctamente programadas el tiempo que tarde en recorrer un orificio u otro puede ser significativo con lo que la producción de placas bajaría en un período de tiempo.

En el **problema de conexión de chips** la idea es minimizar la cantidad de cable necesaria para unir todos los puntos de una placa sin que haya interferencias. Como los chips son de pequeño tamaño no se pueden poner más de dos cables en un único pin. Tomando las ciudades como los pins y la cantidad de cable necesaria para unirlos como la distancia, el problema es equivalente al de viajante de comercio.

* **Aplicaciones en internet.**

Supongamos que el viajante de comercio es un bit de datos, y que las ciudades son servidores de Red distribuidos por todo el planeta. Esta variante del problema del viajante de comercio es algo inherente al uso óptimo de una plataforma masiva de distribución como es Internet. No olvidemos que en cada ruta puede haber miles de ciudades en este caso. Es curiosos como para resolver esta variante algunos investigadores se han inspirado en el comportamiento de las hormigas.

* **Problema de la red de basuras.**

El problema de la recolección de residuos puede dividirse en 3 grandes tipos: domiciliaria, comercial e industrial. La recolección domiciliaria consiste en atender fundamentalmente casas particulares. La frecuencia puede variar aunque normalmente las rutas suelen repetirse una vez todos los días. La recolección comercial o industrial se encarga de las tiendas, restaurantes o edificios de cocinas. Los objetivos en este tipo de problemas pueden ser diversos: minimizar el número de camiones, la distancia recorrida... Si queremos minimizar la distancia usaremos el problema del viajante de comercio identificando los contenedores o puntos de recogida como las ciudades a visitar. Estos mismos problemas se pueden generalizar a los llamados problemas de vehículos o de reparto. Se usan en las empresas de transportes, en correos...