UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

MIARFID Máster en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

DISEÑO, IMPLEMENTACIÓN Y EVALUACIÓN DE ALGORITMO GENÉTICO Y ALGORITMO DE ENFRIAMIENTO SIMULADO PARA EL PROBLEMA DEL VIAJANTE DE COMERCIO (TSP)

Autor: Pascual Andrés Carrasco Gómez
Curso: 2016/17

Índice

INTRODUCCIÓN

1. Definición del problema

El problema del viajante de comercio (TSP en inglés) o problema del viajante, responde a la siguiente pregunta: Dada una lista de ciudades y las distancias entre cada par de ellas, ¿cuál es la ruta más corta posible que visita cada ciudad exactamente una vez y regresa a la ciudad origen? Este es un problema NP-duro dentro de la optimización combinatoria, muy importante en la investigación de operaciones y en la ciencia de la computación.

La combinatoria en este problema viene definida por las permutaciones posibles del número de ciudades que tiene una instancia del problema, es decir, si tenemos 5 ciudades la combinatoria es 5! = 120, si tenemos 10 ciudades la combinatoria es 10! = 3628800, observamos que el problema explota computacionalmente y por este motivo es importante implementar una metaheurística que nos proporcione una solución que consideremos buena.

Este problema es un problema de optimización pero no de satisfactibilidad ya que obtener una solución del problema es trivial, lo costoso se encuentra en encontrar aquella solución que minimice el recorrido.

2. Métodos aplicados

En este trabajo se han implementado dos metaheurísticas para resolver el problema del viajante de comercio:

- Algoritmo genético: Algoritmos que se inspiran en los procesos de evolución natural y genética. Se centran en evolucionar a partir de una población inicial de soluciones intentando conseguir nuevas generaciones de soluciones que sean mejores que la anterior.
- Algoritmo por enfriamiento simulado: Están basados en el proceso de enfriar lentamente un material en estado líquido hasta un estado cristalino de mínima energía.

  Es una metaheurística de mejora, por lo que necesita de una solución inicial para poder funcionar.

3. Entorno de trabajo

El entorno de trabajo que se ha utilizado es Python2.7. Se han implementado ambos algoritmos desde cero con la finalidad de comprender mejor el funcionamiento interno de los algoritmos que han sido explicados en las clases de teoría.

ALGORITMO GENÉTICO

1. Diseño del algoritmo

1.1 Representación

Un individuo en nuestro problema es un recorrido formado por todas las ciudades sin repetirse, empezando y acabando el recorrido con la misma ciudad.
Por lo tanto un individuo en nuestro problema es una lista de longitud (número de ciudades + 1) que representa una solución a la instancia del problema TSP dado.
Imaginemos que tenemos tres ciudades: Valencia, Barcelona y Madrid. El número de soluciones posibles a esta instancia del problema TSP sería 3! = 6:

[Valencia,Barcelona,Madrid,Valencia]
[Valencia,Madrid,Barcelona,Valencia]
[Madrid,Barcelona,Valencia,Madrid]
[Madrid,Valencia,Barcelona,Madrid]
[Barcelona,Valencia,Madrid,Barcelona]
[Barcelona,Madrid,Valencia,Barcelona]

Nuestro programa trabaja directamente con ficheros TSPLIB que encontramos en paginas como:
http://www.tsp.gatech.edu/world/countries.html
http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/
Estos ficheros tienen instancias de problemas TSP reales formados por tres columnas, donde la primera columna representa la ciudad y las dos columnas siguientes representan las coordenadas (x,y) de esa ciudad. Las ciudades están codificadas numéricamente en un rango que va desde 1 hasta el número de ciudades de la instancia del problema TSP.
Python trabaja con listas que comienzan con indice 0, por comodidad y para gestionar las EDA's que comentaremos en el apartado de implementación de la memoria, nuestros individuos trabajan en un rango que va desde 0 hasta el (numero de ciudades – 1) de la instancia del problema dada.
Para la instancia de tres ciudades vista anteriormente el formato TSPLIB sería el siguiente:

1 coordenadaX_nodo1 coordenadaY_nodo1
2 coordenadaX_nodo2 coordenadaY_nodo2
3 coordenadaX_nodo3 coordenadaY_nodo3

La representación de soluciones para este problema en nuestro programa sería:
[[0,1,2,0],[0,2,1,0],[1,0,2,1],[1,2,0,1],[2,0,1,2],[2,1,0,2]]

1.2 Aptitud: Función fitness

La función fitness en el problema del viajante de comercio (TSP) es la distancia (euclídea) recorrida para una solución (individuo) dada del problema. En la representación por la que hemos optado en nuestro trabajo un individuo = recorrido = solución al problema.

1.3 Población inicial

En nuestro programa la población inicial se genera aleatoriamente, el número de individuos que tiene la población inicial viene definido por un parámetro de entrada al programa.
Para comprender mejor el diseño de nuestro algoritmo vamos a exponer un ejemplo que iremos ilustrando en cada parte del diseño.
El ejemplo esta compuesto por cuatro ciudades y le hemos indicado a nuestro programa que genere seis individuos como población inicial, los cuales a generado de forma aleatoria:
[[2, 1, 3, 0, 2], [3, 1, 0, 2, 3], [3, 1, 2, 0, 3], [1, 3, 2, 0, 1], [2, 1, 0, 3, 2], [1, 3, 2, 0, 1]]

1.4 Ciclo evolutivo

1.4.1 Selección

La selección en el problema de TSP viene dada por aquellos individuos cuya función fitness sea menor, ya que lo que buscamos es minimizar el recorrido, es decir, buscamos aquella solución con distancia mínima. El número de individuos que seleccionamos en la población viene definido por otro parámetro de entrada a nuestro programa. Para el ejemplo ilustrativo hemos indicado que queremos seleccionar tres individuos, los individuos seleccionados reciben el nombre de padres:

- Función fitness obtenida:
  Individuo 0 : 24.1934853067
  Individuo 1 : 19.0908326497
  Individuo 2 : 24.1934853067
  Individuo 3 : 19.0908326497
  Individuo 4 : 18.313755208
  Individuo 5 : 19.0908326497
- Selección:
  Padre 0: [2, 1, 0, 3, 2] (Individuo 4)
  Padre 1: [3, 1, 0, 2, 3] (Individuo 1)
  Padre 2: [1, 3, 2, 0, 1] (Individuo 3)

1.4.2 Cruce

Hemos implementado dos versiones de cruce en nuestro algoritmo, una versión se basa en un cruce uniforme que explicamos a continuación para el ejemplo ilustrativo que llevamos a cabo:
El número de hijos (individuos obtenidos a partir del cruce de dos padres) en nuestro diseño es (número de padres – 1) por lo tanto para nuestro ejemplo ilustrativo tendremos dos hijos.

- Hijo 1: obtenido de Padre 0 y Padre 1
  [2, 1, 0, 3, 2] [3, 1, 0, 2, 3] → [2,3,1,0,2]
- Hijo 2: obtenido de Padre 1 y Padre 2
  [3, 1, 0, 2, 3] [1, 3, 2, 0, 1] → [3,1,0,2,3]

Los dos nuevos individuos (hijos) son: [[2,3,1,0,2],[3,1,0,2,3]]
Este tipo de cruce es un cruce que nos proporciona poca información de los padres, esto de primera impresión puede verse como algo negativo pero la finalidad que buscamos al implementar este cruce es obtener hijos lo mas aleatorios posibles, es decir, buscamos mas la exploración que la explotación del problema.

La segunda versión se basa en realizar un corte de un punto de un padre y unirlo con el corte de un punto de otro padre obteniendo un hijo que sea la combinación de ambos padres, lo vemos en el ejemplo ilustrativo que llevamos a cabo para que sea mas fácil de entender:

- Hijo 1: obtenido de Padre 0 y Padre 1:
  [2, 1, 0, 3, 2] [3, 1, 0, 2, 3] → [2,1,0,2] → Se corrige la repetición → [2,1,0,3] → Se añade el nodo inicial como final → [2,1,0,3,2]
- Hijo 2: obtenido de Padre 1 y Padre 2:
  [3, 1, 0, 2, 3] [1, 3, 2, 0, 1] → [3,1,2,0] → No hay repetición → [3,1,2,0] → Se añade el nodo inicial como final → [3,1,2,0,3]
- Hijo 3: obtenido de Padre 2 y Padre 0:
  [1, 3, 2, 0, 1] [2, 1, 0, 3, 2] → [1,3,0,3] → Se corrige la repetición → [1,3,0,2] → Se añade el nodo inicial como final → [1,3,0,2,1]

Este tipo de cruce nos proporciona gran parte información de los padres al generar un hijo, esto puede verse como algo mas coherente pero es interesante estudiar ambos cruces y sacar conclusiones. En este cruce buscamos mas la explotación que la exploración del problema.
Los siguientes pasos del algoritmo son comunes para ambas versiones, con lo cual los explicaremos con la versión de cruce uniforme.

1.4.3 Mutación

En nuestro diseño se obtienen dos posiciones de la lista de forma aleatoria sin tener en cuenta las posiciones de la ciudad origen y destino (que como ya sabemos es la misma). A todos los hijos obtenidos mediante el cruce se le aplica una mutación por intercambio recíproco variando esas dos posiciones. En el ejemplo ilustrativo imaginemos que han salido las posiciones 1 y 2 de la lista a intercambiar:
Hijo 1: [2,3,1,0,2] → [2,1,3,0,2]
Hijo 2: [3,1,0,2,3] → [3,0,1,2,3]
Los dos individuos (hijos) son: [[2,1,3,0,2],[3,0,1,2,3]]

1.4.4 Reemplazo

Reemplazamos de la población actual tantos individuos como hijos tengamos, los individuos que remplazamos son aquellos con mayor valor de función fitness (peores individuos) y además se ha de cumplir que no sean padres (se ha decidido de forma personal en el diseño):

- Peores individuos:
  Individuo 0 : 24.1934853067
  Individuo 2 : 24.1934853067
- Población actualizada:
  Individuo 0 : [2, 1, 3, 0, 2]
  Individuo 1 : [3, 1, 0, 2, 3]
  Individuo 2 : [3, 0, 1, 2, 3]
  Individuo 3 : [1, 3, 2, 0, 1]
  Individuo 4 : [2, 1, 0, 3, 2]
  Individuo 5 : [1, 3, 2, 0, 1]

El criterio establecido para que en el remplazo no se puedan remplazar aquellos individuos que han sido seleccionados como padres crea una restricción a la hora de llamar al programa python2.7 que es necesario comentar para que se realice la ejecución correctamente:

n_padres_selección <= n_individuos_poblacion/2

También es importante comentar, aunque ya nos habremos dado cuenta al comprender la selección y el remplazo de nuestro algoritmo, que en todas las generaciones (iteraciones) tenemos el mismo número de individuos en la población.

1.5 Condición de parada

En nuestro algoritmo la condición de parada es el número de generaciones (iteraciones) que indiquemos como parámetro al ejecutar el programa.

2. Implementación

Parámetros de entrada que nos permiten configurar la ejecución de nuestro programa para resolver el problema TSP indicado:

```
17  # Comprobacion de parametros de entrada
18  if len(sys.argv) != 6:
19      print "--------------------------------------------------------------------------------"
20      print "Uso: python2.7 viajante.py <f_tsp> <n_gen> <n_ind_pob> <n_padres> <verbose>"
21      print "--------------------------------------------------------------------------------"
22      print "<f_tsp>: Fichero con las coordenadas del problema TSP (string)"
23      print "<n_gen>: Iteraciones = generaciones en el bucle (int)"
24      print "<n_ind_pob>: Cantidad de individuos en la poblacion (int)"
25      print "<n_padres>: Numero de padres en la seleccion (int)"
26      print "<verbose>: Verbose por pantalla (boolean)"
27      sys.exit(0)
28
29  # Parametros generales para el algoritmo genetico
30  f_tsp = sys.argv[1]
31  n_generaciones = int(sys.argv[2])
32  n_ind = int(sys.argv[3])
33  n_padres_seleccion = int(sys.argv[4])
34  modo_verbose = True if sys.argv[5] == "True" else False
35  mejor_individuo = []
36  mejor_puntuacion = float("inf")
37
38  # Restriccion en el remplazo del algoritmo genetico
39  if float(n_padres_seleccion) > float(n_ind)/2:
40      print "Error: El numero de n_padres_seleccion ha de ser < que n_ind/2"
41      sys.exit(0)
```

Imagen 1: Parámetros al ejecutar el algoritmo genético “viajante.py”

Nuestro programa trabaja directamente con los datos de ficheros TSPLIB, trabajamos con dos EDA que nos permiten estructurar los datos, la primera EDA es una lista de nodos que representan las ciudades del problema, y la segunda EDA es una lista en la que se almacena, para cada nodo de la lista de ciudades, una lista con la distancia euclidea de ese nodo respecto a los demás nodos.

```
# ------------------------------------------------------------------------
# GENERACION DE EDAs PARA EL ALGORITMO GENETICO
# ------------------------------------------------------------------------
# Fichero con coordenadas del mapa
nombre_f = f_tsp

# Abrimos el fichero para trabajar con el
fichero = open(nombre_f, 'r')

# Insertamos los datos del fichero en una lista para trabajar con ella
# dato[0] = nodo; dato[1] = x; dato[2] = y;
datos = []
for linea in fichero:
    datos_linea = linea.split()
    datos.append(datos_linea)

# Creamos EDAs para trabajar con el algoritmo genetico
nodos = [] # nodos = ciudades codificadas en numeros
distancias = [] # Distancia de cada nodo con todos los demas
for dato_actual in datos:
    nodos.append(int(dato_actual[0])-1) # Insertamos los nodos
    x_actual = float(dato_actual[1])
    y_actual = float(dato_actual[2])
    # Calculo de distancia euclidea de un nodo a los demas nodos
    aux_distancias = []
    for dato_siguiente in datos:
        x_siguiente = float(dato_siguiente[1])
        y_siguiente = float(dato_siguiente[2])
        x = x_siguiente-x_actual
        x = math.pow(x,2)
        y = y_siguiente-y_actual
        y = math.pow(y,2)
        aux_distancias.append(math.sqrt(x+y))
    distancias.append(aux_distancias)
```

Imagen 2: Creación de EDA's a partir de un fichero de entrada.

En nuestro programa tenemos definidas tres funciones que comentamos a continuación:

```
# fitness = distancia de la secuencia
def f_fitness(individuo,distancias):
    d = 0
    for i in range(0,len(individuo)-1):
        nodo_actual = individuo[i]
        nodo_siguiente = individuo[i+1]
        d = d + distancias[nodo_actual][nodo_siguiente]
    return d

# Genera una poblacion inicial de n_individuos
def generar_poblacion_incial(nodos,n_individuos):
    pob_inicial = []
    for i in range(n_individuos):
        pob = []
        for j in range(len(nodos)):
            nodo = random.randint(0,len(nodos)-1)
            while nodo in pob: # No se repiten los nodos en un individuo
                nodo = random.randint(0,len(nodos)-1)
            pob.append(nodo)
        pob.append(pob[0]) # nodo inicial = nodo final
        pob_inicial.append(pob)
    return pob_inicial

# Generar un plot de un individuo
def generar_plot_individuo(individuo,datos):
    # Plot de la solucion obtenida
    # datos[0] = nodo; datos[1] = x; datos[2] = y
    x = []
    y = []
    for i in individuo:
        x.append(datos[i][1])
        y.append(datos[i][2])
    plt.plot(x,y,'g-d')
    plt.show()
```

Imagen 3: Funciones definidas en el algoritmo genético “viajante.py”.

- f_fitness(individuo,distancias): Calcula la función fitness (distancia) dado un individuo y una lista de distancias que corresponde a la segunda EDA que hemos generado, que guarda para cada nodo la distancia euclidea con respecto a los demás nodos.
- generar_poblacion_incial(nodos,n_individuos): Genera una población inicial de talla n_individuos utilizando la EDA de lista de nodos obtenida mediante el fichero TSPLIB.
- generar_plot_individuo(individuo,datos): Genera un plot de un individuo que nos permite ver la solución obtenida por ese individuo al problema que intentamos resolver.

A continuación vamos a mostrar la implementación que hemos realizado para el algoritmo genético que hemos ilustrado en el apartado de diseño de la memoria de la práctica, vamos a dividir el código según la estructura de un algoritmo genético.

Generación de población inicial:

```
117  # Creamos la poblacion inicial
118  poblacion = generar_poblacion_incial(nodos,n_ind)
119  if modo_verbose:
120      print "-----------------------------------"
121      print "POBLACION INICIAL"
122      print "-----------------------------------"
123      print poblacion
```

Imagen 4: Implementación de la población inicial del algoritmo genético.

Bucle de generaciones (iteraciones): Condición de parada.

```
124  # Bucle generaciones
125  for i in range(n_generaciones):
126      if modo_verbose:
127          print "-----------------------------------"
128          print "GENERACION ",i
129          print "-----------------------------------"
130          print poblacion
```

Imagen 5: Implementación del bucle de generaciones (condición de parada) del algoritmo genético.

Estimación de la función fitness de cada individuo de la población y actualizamos los valores de mejor individuo obtenido hasta el momento en caso de que sea necesario:

```
131      # Aplicamos funcion fitness a la poblacion
132      puntuaciones = []
133      for individuo in poblacion:
134          puntuacion = f_fitness(individuo,distancias)
135          if puntuacion < mejor_puntuacion:
136              mejor_puntuacion = puntuacion
137              mejor_individuo = individuo
138          puntuaciones.append(puntuacion)
139      puntuaciones_ord = sorted(puntuaciones)
140      if modo_verbose:
141          print "-----------------------------------"
142          print "PUNTUACIONES"
143          print "-----------------------------------"
144          for j in range(len(poblacion)):
145              print j,": ",puntuaciones[j]
```

Imagen 6: Función fitness de cada individuo y almacenamiento de mejor individuo global obtenido.

Selección de n_padres con menor función fitness:

```
146     # Seleccion: n_padres_seleccion con menor puntuacion de fitness
147     indices_padres = []
148     for j in range(n_padres_seleccion):
149         aux_i = 1
150         aux_indice = puntuaciones.index(puntuaciones_ord[j])
151         while aux_indice in indices_padres:
152             aux_indice = puntuaciones.index(puntuaciones_ord[j],aux_i)
153             aux_i += 1
154         indices_padres.append(aux_indice)
155     if modo_verbose:
156         print "-----------------------------------"
157         print "SELECCION"
158         print "-----------------------------------"
159         for j in indices_padres:
160             print "Padre: ",j
```

Imagen 7: Implementación de la selección de los mejores individuos de la población actual.

Cruce: Cruce uniforme por cada par de padres obteniendo (n_padres – 1) hijos:

```
161     # Cruce: Cruce uniforme sin repetir nodos (ciudades)
162     hijos_cruzados = []
163     for j in range(len(indices_padres)-1):
164         padre1 = poblacion[indices_padres[j]]
165         padre2 = poblacion[indices_padres[j+1]]
166         hijo = []
167         hijo.append(padre1[0])
168         p1 = 1
169         p2 = 0
170         while len(hijo) != len(padre1)-1: # len(padre1)-1: porque
171             while (p2 < len(padre2)) and (padre2[p2] in hijo):
172                 p2 += 1
173             if p2 < len(padre2):
174                 hijo.append(padre2[p2])
175             while (p1 < len(padre1)) and (padre1[p1] in hijo):
176                 p1 += 1
177             if p1 < len(padre1):
178                 hijo.append(padre1[p1])
179         hijo.append(hijo[0]) # El nodo destino es el nodo origen
180         hijos_cruzados.append(hijo)
181     if modo_verbose:
182         print "-----------------------------------"
183         print "CRUCE"
184         print "-----------------------------------"
185         for hijo in hijos_cruzados:
186             print "Hijo cruzado: ",hijo
```

Imagen 8: Versión de cruce uniforme del algoritmo genético “viajante_py”

Cruce: Cruce de corte de un punto por cada par de padres obteniendo (n_padres) hijos:

```
161     # Cruce: Cruce por corte de un punto, sin repetir nodos (ciudades)
162     hijos_cruzados = []
163     conjunto_nodos = set(nodos)
164     for j in range(len(indices_padres)):
165         if j == len(indices_padres)-1: # Cruce ultimo nodo con el primero
166             padre1 = poblacion[indices_padres[j]]
167             padre2 = poblacion[indices_padres[0]]
168         else:
169             padre1 = poblacion[indices_padres[j]]
170             padre2 = poblacion[indices_padres[j+1]]
171         particion = int(round((len(padre1)-1)/2.0))
172         hijo = padre1[0:particion]
173         hijo += padre2[particion:len(padre2)-1]
174         conjunto_hijo = set(hijo)
175         nodos_a_insertar = list(conjunto_nodos - conjunto_hijo)
176         nodos_visitados = []
177         for z in range(len(hijo)):
178             if hijo[z] in nodos_visitados:
179                 hijo[z] = nodos_a_insertar.pop()
180             else:
181                 nodos_visitados.append(hijo[z])
182         hijo.append(hijo[0]) # nodo inicial = nodo final
183         hijos_cruzados.append(hijo)
184     if modo_verbose:
185         print "-----------------------------------"
186         print "CRUCE"
187         print "-----------------------------------"
188         for hijo in hijos_cruzados:
189             print "Hijo cruzado: ",hijo
```

Imagen 9: Versión de cruce por corte de un punto del algoritmo genético “viajante_v2.py”

Mutación por intercambio recíproco de dos posiciones obtenidas aleatoriamente:

```
187     # Mutacion: intercambiar dos nodos (sin tener en cuenta el inicial y el final)
188     hijos = []
189     pos_1 = random.randint(1,len(nodos)-2)
190     pos_2 = random.randint(1,len(nodos)-2)
191     while pos_2 == pos_1:
192         pos_2 = random.randint(1,len(nodos)-2)
193     for hijo_cruzado in hijos_cruzados:
194         hijo = hijo_cruzado[:] # Copia de la lista hijo_cruzado
195         hijo[pos_1] = hijo_cruzado[pos_2]
196         hijo[pos_2] = hijo_cruzado[pos_1]
197         hijos.append(hijo)
198     if modo_verbose:
199         print "-----------------------------------"
200         print "MUTACION"
201         print "-----------------------------------"
202         print "Indices intercambio: ",pos_1,pos_2
203         for hijo in hijos:
204             print "Hijo: ",hijo
```

Imagen 10: Implementación de la mutación del algoritmo genético.

Remplazamos los hijos obtenidos por los individuos de la población con mayor puntuación fitness:

```
205     # Remplazo
206     indices_individuos_peores = []
207     for j in range(len(hijos)):
208         aux_i = 1
209         aux_indice = puntuaciones.index(puntuaciones_ord[len(puntuaciones_ord)-1-j])
210         while (aux_indice in indices_padres) or (aux_indice in indices_individuos_peores): # Coger hijos que no sean padres
211             aux_indice = puntuaciones.index(puntuaciones_ord[len(puntuaciones_ord)-1-j],aux_i)
212             aux_i += 1
213         indices_individuos_peores.append(aux_indice)
214         poblacion[aux_indice] = hijos[j]
215     if modo_verbose:
216         print "-----------------------------------"
217         print "REMPLAZO:"
218         print "-----------------------------------"
219         print("Indices_peores_individuos:")
220         for indice in indices_individuos_peores:
221             print "Indice: ",indice
222         print "Poblacion nueva:"
223         for individuo in poblacion:
224             print "Individuo: ",individuo
```

Imagen 11: Implementación del remplazo en el algoritmo genético.

Al finalizar el bucle de generaciones nuestro programa devuelve por pantalla la solución que corresponde al mejor individuo que hemos obtenido, mostramos por pantalla el individuo obtenido y su correspondiente función fitness.
También se ha implementado que nos muestre el individuo obtenido mediante un plot para poder comparar nuestra solución con la mejor solución encontrada por las paginas que nos han proporcionado los ficheros TSPLIB.

```
227   # ---------------------------------------------------------
228   # SALIDA POR PANTALLA
229   # ---------------------------------------------------------
230   # Mostramos el mejor individuo obtenido en la ejecución
231   print "-----------------------------------"
232   print "MEJOR INDIVIDUO"
233   print "-----------------------------------"
234   print "Individuo: ",mejor_individuo
235   print "Puntuacion : ",mejor_puntuacion
236
237   # Generamos el mejor individuo obtenido
238   generar_plot_individuo(mejor_individuo,datos)
```

Imagen 12: Implementación de la salida por pantalla del algoritmo genético.

NOTA: Para poder plotear el resultado se tiene que instalar el modulo matplotlib de python2.7, si no esta instalado hay dos opciones, instalarlo mediante el comando pip, o comentar la última línea de código “generar_plot_individuo(mejor_individuo,datos)”.

Para ejecutar nuestro programa necesitamos saber que parámetros hemos de introducirle como entrada y en que orden, para ello simplemente ejecutamos el nombre del programa y nos proporciona la información que requiere:

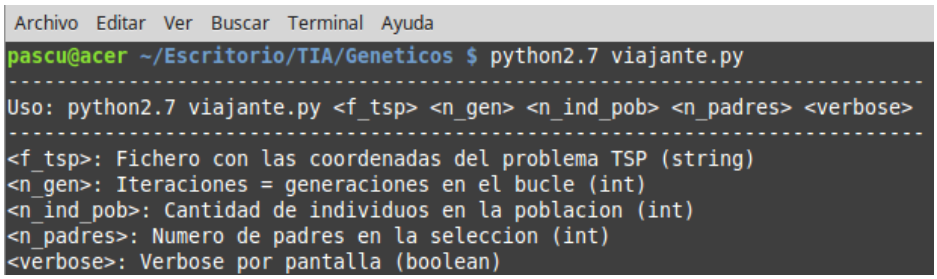


Imagen 13: Parámetros de entrada del algoritmo genético.

Vamos a realizar dos ejemplos de ejecución para familiarizarnos con el programa, una ejecución va a ser sin modo verbose y la otra con modo verbose. Vamos a utilizar un fichero TSPLIB trivial generado por mi para ver claramente los resultados, vamos a obtener una única generación, con una población de cuatro individuos, seleccionando dos individuos como padres:

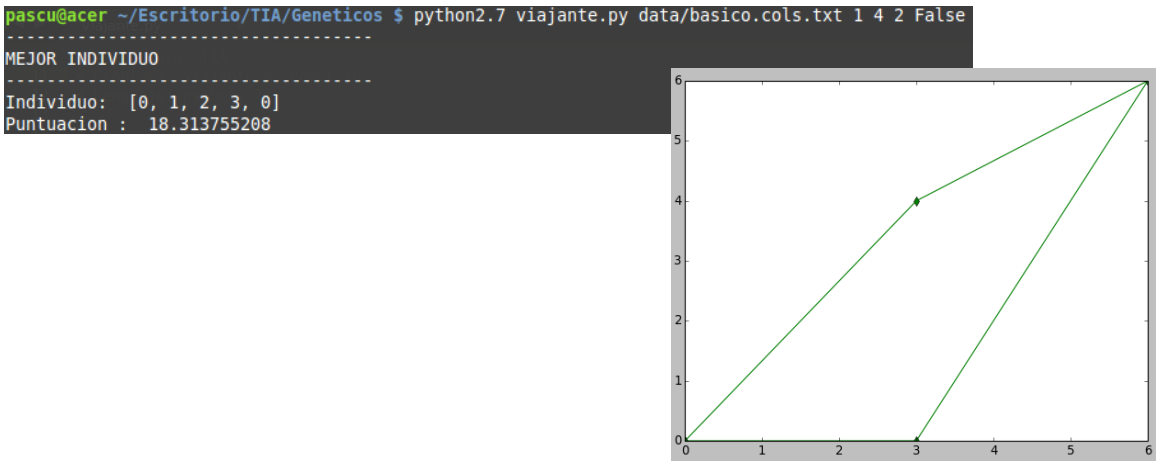


Imagen 14: Ejemplo de ejecución sin modo verbose del algoritmo genético.

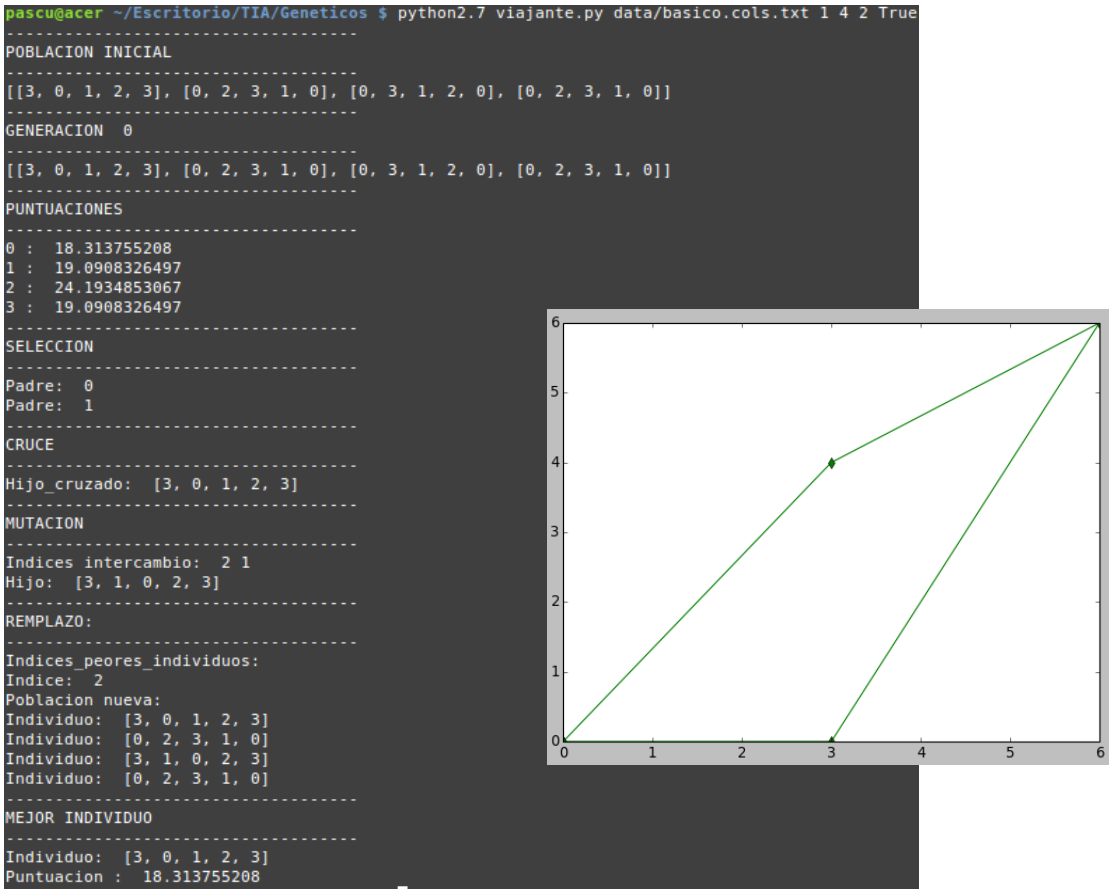


Imagen 15: Ejemplo de ejecución con modo verbose del algoritmo genético.

NOTA: El programa python2.7 “viajante_v2.py” se ejecuta y muestra por pantalla lo mismo que el programa python2.7 “viajante.py”, la diferencia entre los dos programas únicamente es el método de cruce aplicado en el algoritmo genético.

3. Evaluación

Para realizar la evaluación del algoritmo genético del viajante de comercio vamos a utilizar datos reales de ciudades de países que podemos obtener en la url:

http://www.math.uwaterloo.ca/tsp/world/countries.html

En concreto vamos a utilizar los tres países con un número de ciudades considerablemente diferentes Western Sahara, Qatar y Uruguay. Vemos los problemas de forma gráfica a continuación:

Western Sahara

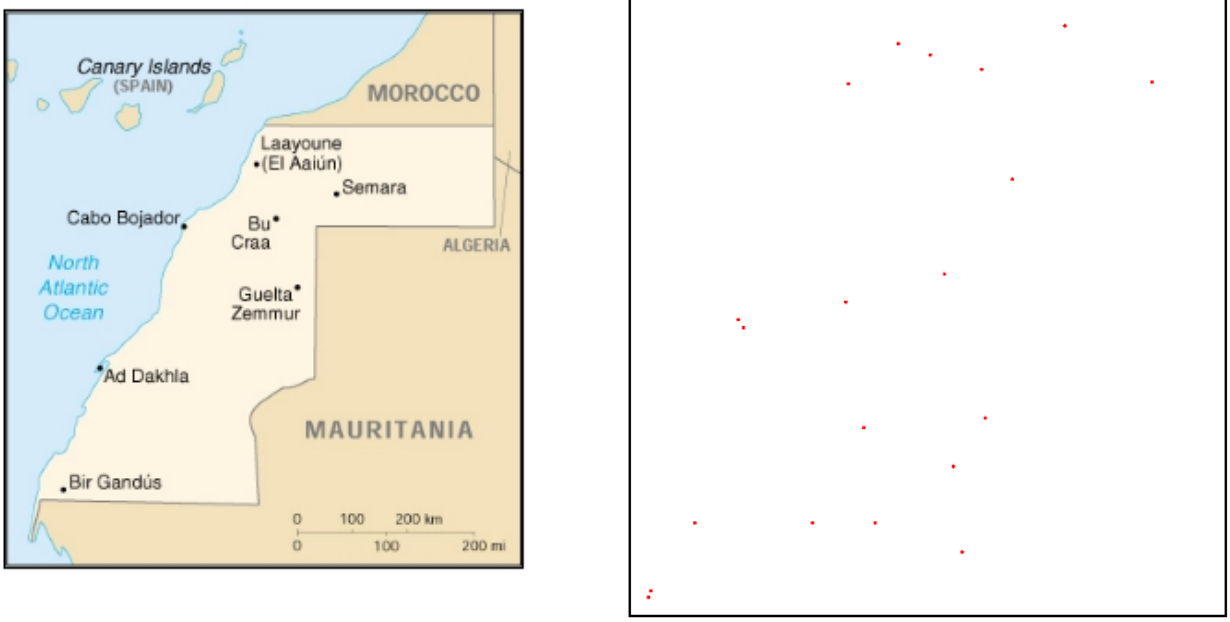


Imagen 16: Izquierda, Mapa Western Sahara. Derecha, PointSet de 29 ciudades del país.

Qatar

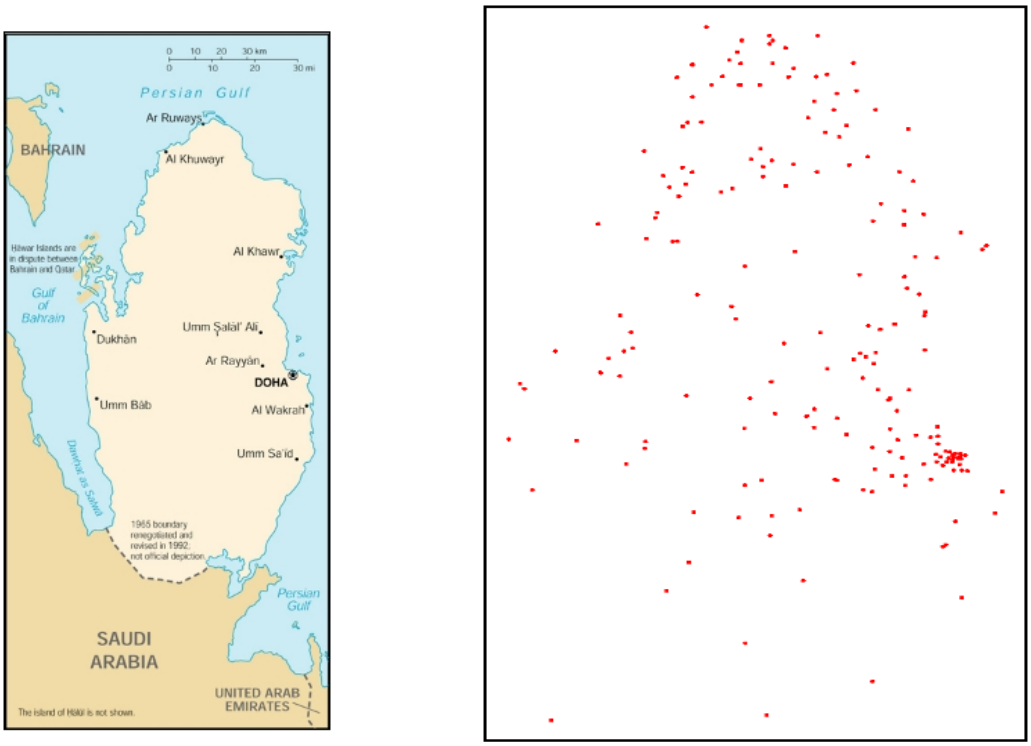


Imagen 17: Izquierda, Mapa Qatar. Derecha, PointSet de 194 ciudades del país.

Uruguay

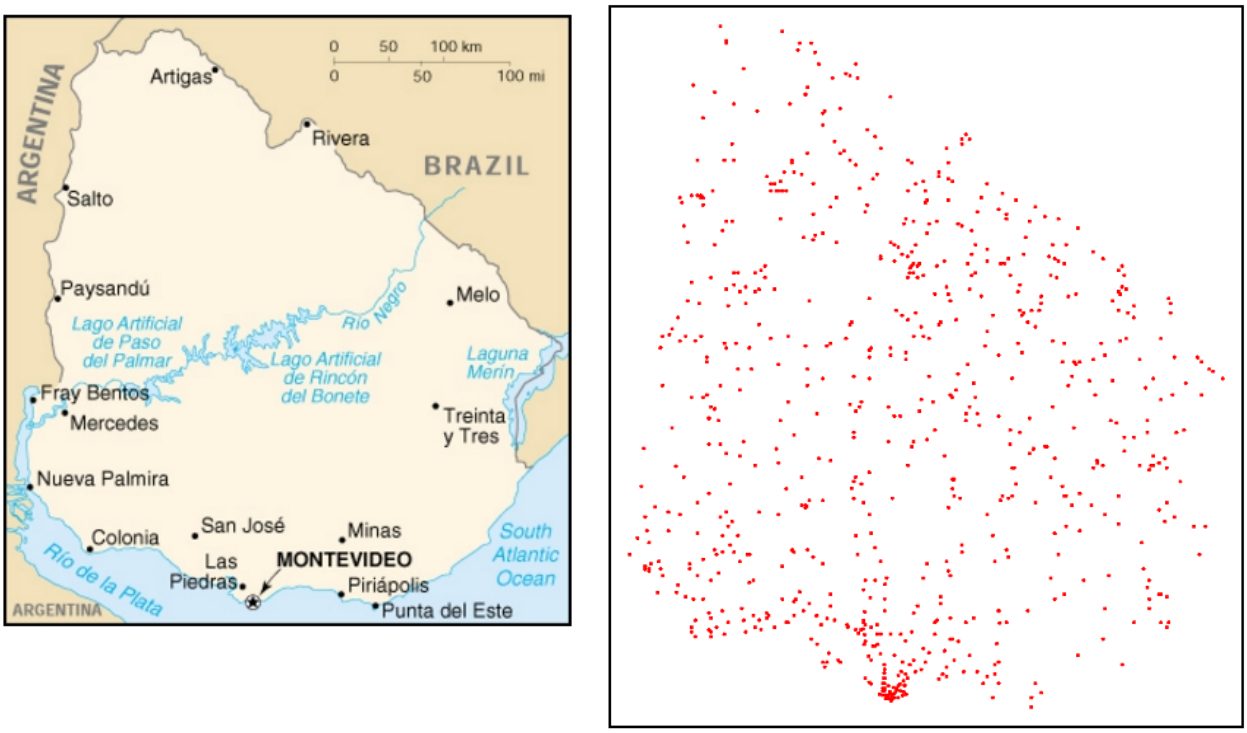


Imagen 18: Izquierda, Mapa Uruguay. Derecha, PointSet de 734 ciudades del país.

Para cada uno de los PointSet se han realizado pruebas comparando las soluciones que nos aporta nuestro algoritmo genético respecto a la solución óptima que nos proporciona la página web.
Las ejecuciones se han realizado para las dos versiones de cruce que hemos expuesto en el diseño.
Cada celda de la tabla representa la ejecución del algoritmo genético con el número de generaciones (iteraciones del bucle) especificado por columna y el número de individuos por población especificado por fila. El número de padres (selección) se ha definido en todas las ejecuciones como:

n_padres_seleccion = n_individuo_poblacion / 2

Cada resultado que se observa en cada celda corresponde al valor de la mediana obtenida sobre cinco ejecuciones con la finalidad de obtener un resultado mas real del problema.
NOTA: Para probar los scripts del anexo de la memoria es recomendable que se ejecuten sobre el problema Western Sahara (wi29.data.txt), por el motivo del tiempo de computo, sobre mi maquina el script con el problema de Uruguay (uy734.data.txt) tardo en finalizar 10 horas aproximadamente.

PointSet Western Sahara (wi29.data.txt): cruce uniforme				
Nº Generaciones	100	1000	10000	30000
5	73380.30	65205.36	59515.89	57237.84
10	54455.87	38982.98	35636.66	34613.86
20	52395.60	33465.52	37199.73	35709.80
30	47107.30	34581.97	33906.03	33345.62

Tabla 1: Resultados de algoritmo genético para Western Sahara aplicando cruce uniforme.

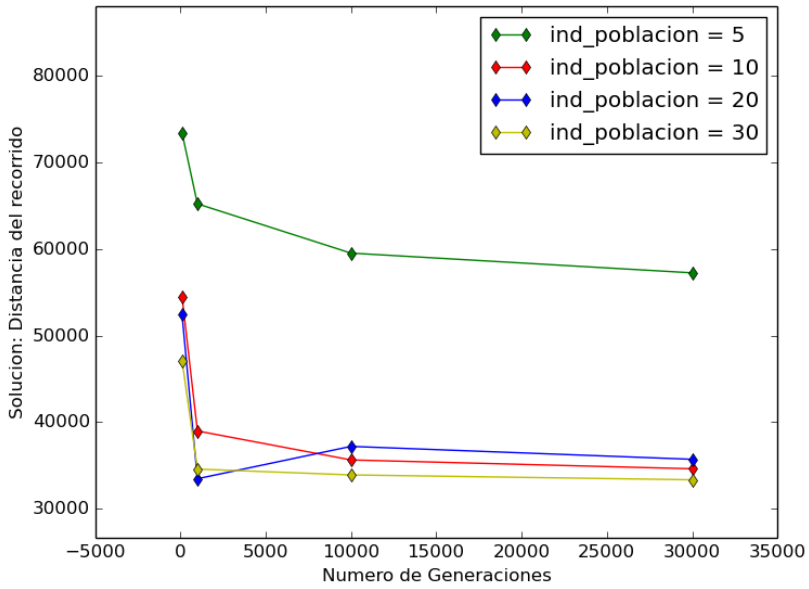


Imagen 19: Gráfico obtenido con los datos de la tabla 1.

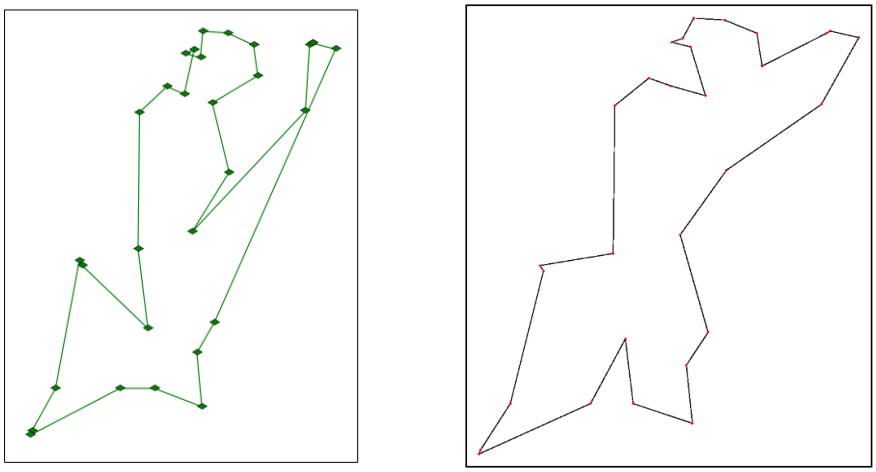


Imagen 20: Izquierda, mejor solución obtenida en la tabla 1 (33345.62). Derecha, solución optima (27603)

PointSet Western Sahara (wi29.data.txt): cruce por corte de un punto				
Nº Generaciones	100	1000	10000	30000
5	66904.47	44078.27	38812.11	39372.62
10	56012.63	41184.03	37960.46	40131.54
20	52952.70	38915.05	38716.76	34809.36
30	53173.48	40545.60	35433.94	35245.46

Tabla 2: Resultados de algoritmo genético para Western Sahara aplicando cruce por corte de un punto.

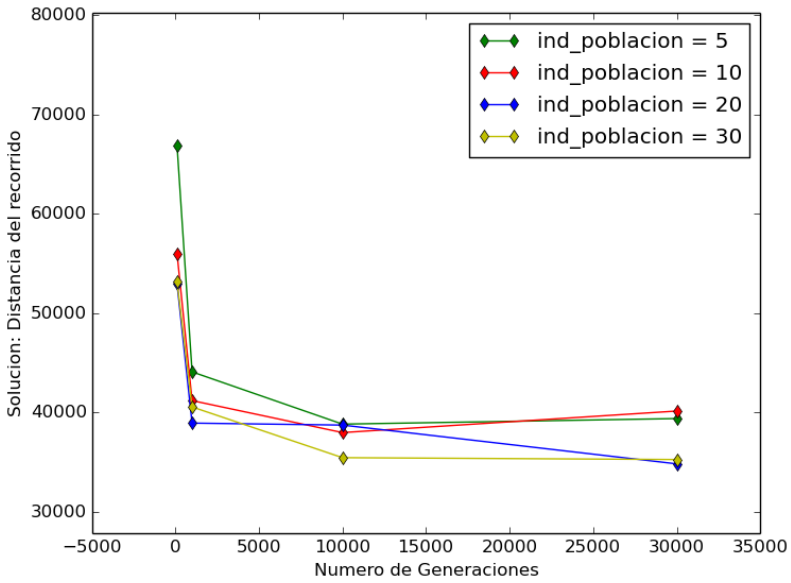


Imagen 21: Gráfico obtenido con los datos de la tabla 2.

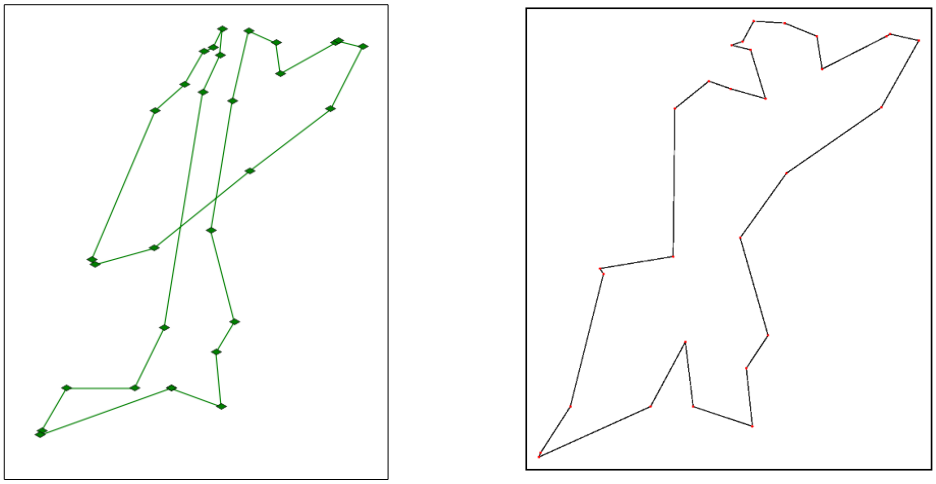


Imagen 22: Izquierda, mejor solución obtenida en la tabla 2 (34809.36). Derecha, solución optima (27603)

PointSet Qatar (Qa194.data.txt): cruce uniforme				
Nº Generaciones	100	1000	10000	30000
5	88086.25	89814.72	85377.29	87065.26
10	83729.55	52045.32	27731.14	19856.49
20	68307.69	40231.01	19386.97	15810.08
30	67763.37	40968.078	18591.76	15854.99

Tabla 3: Resultados de algoritmo genético para Qatar aplicando cruce uniforme.

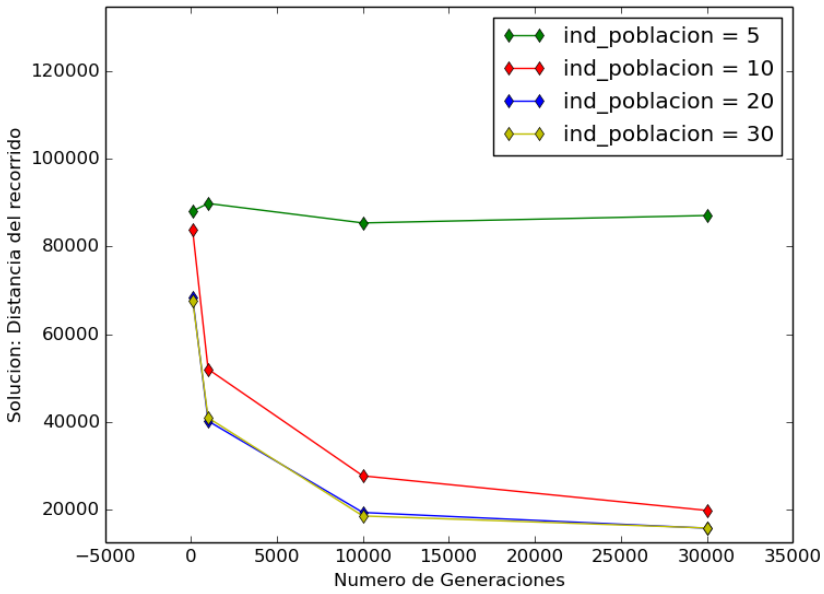


Imagen 23: Gráfico obtenido con los datos de la tabla 3.

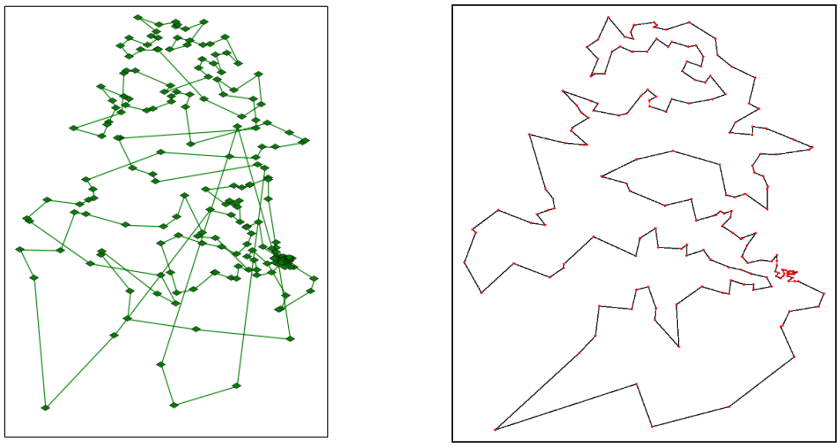


Imagen 24: Izquierda, mejor solución obtenida en la tabla 3 (15810.08). Derecha, solución optima (9352)

PointSet Qatar (Qa194.data.txt): cruce por corte de un punto				
Nº Generaciones	100	1000	10000	30000
5	75252.08	48319.46	27308.09	23901.89
10	70277.33	45715.02	28706.86	23044.88
20	68216.68	46367.02	28553.09	23217.74
30	67761.35	45008.25	28200.53	23247.88

Tabla 4: Resultados de algoritmo genético para Qatar aplicando cruce por corte de un punto.

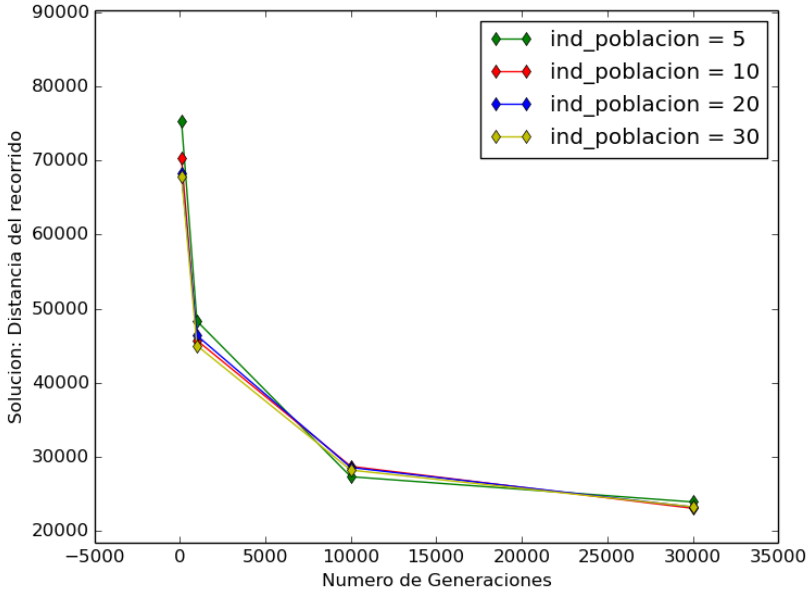


Imagen 25: Gráfico obtenido con los datos de la tabla 4.

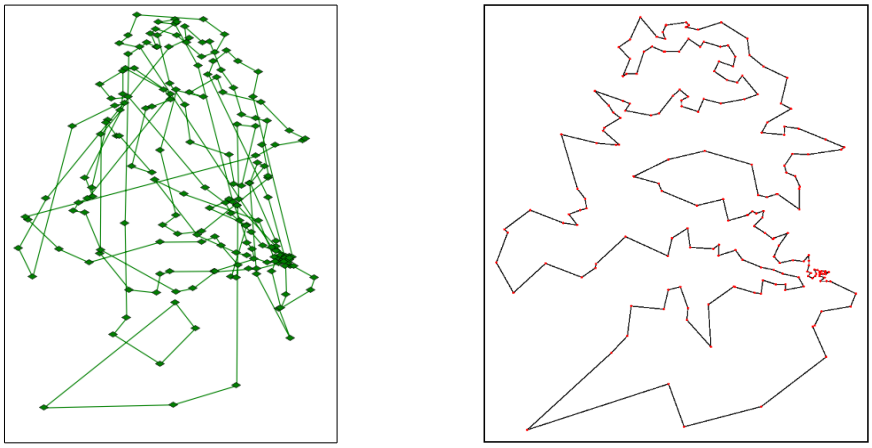


Imagen 26: Izquierda, mejor solución obtenida en la tabla 4 (23044.88). Derecha, solución optima (9352)

PointSet Uruguay (uy734.data.txt): cruce uniforme				
Nº Generaciones	100	1000	10000	30000
5	1600332.83	1587337.11	1612415.91	1596721.26
10	1576692.12	1583799.23	730064.35	558305.18
20	1475270.60	1062710.49	553001.46	387217.38
30	1457964.58	987067.34	533905.11	338058.88

Tabla 5: Resultados de algoritmo genético para Uruguay aplicando cruce uniforme.

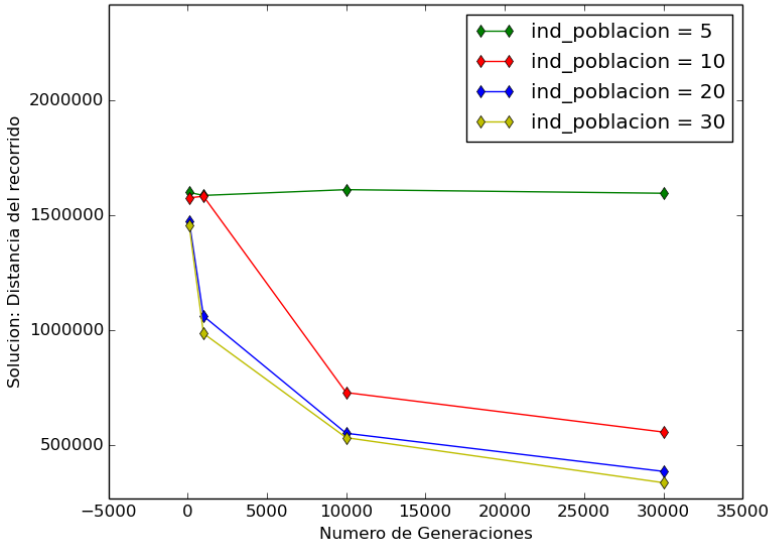


Imagen 27: Gráfico obtenido con los datos de la tabla 5.

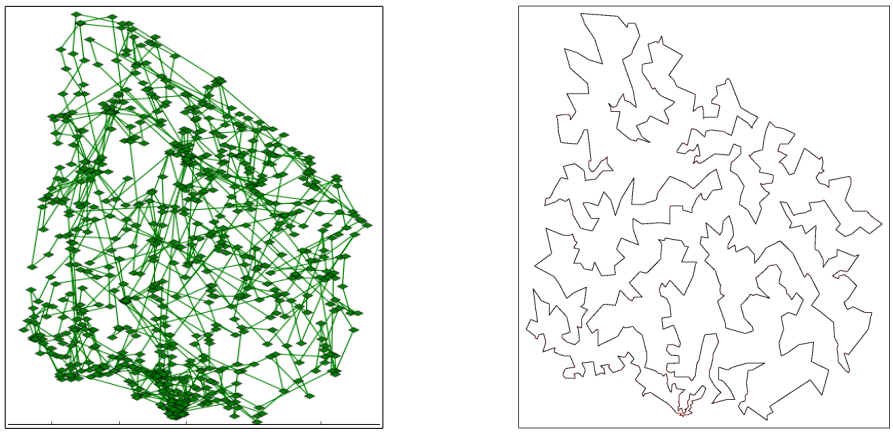


Imagen 28: Izquierda, mejor solución obtenida en la tabla 5 (338058.88). Derecha, solución optima (79114)

PointSet Uruguay (uy734.data.txt): cruce por corte de un punto				
Nº Generaciones	100	1000	10000	30000
5	1502436.43	1175195.05	690294.65	527610.17
10	1434357.30	1136148.79	665475.23	512442.95
20	1415782.59	1117262.64	670201.93	509460.05
30	1425937.22	1109606.83	662905.19	512410.25

Tabla 6: Resultados de algoritmo genético para Uruguay aplicando cruce por corte de un punto.

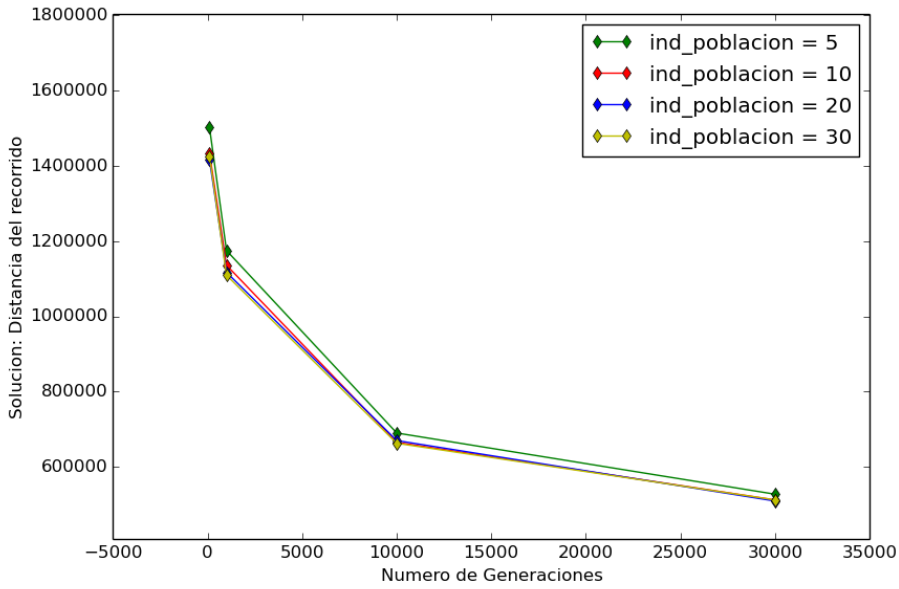


Imagen 29: Gráfico obtenido con los datos de la tabla 6.

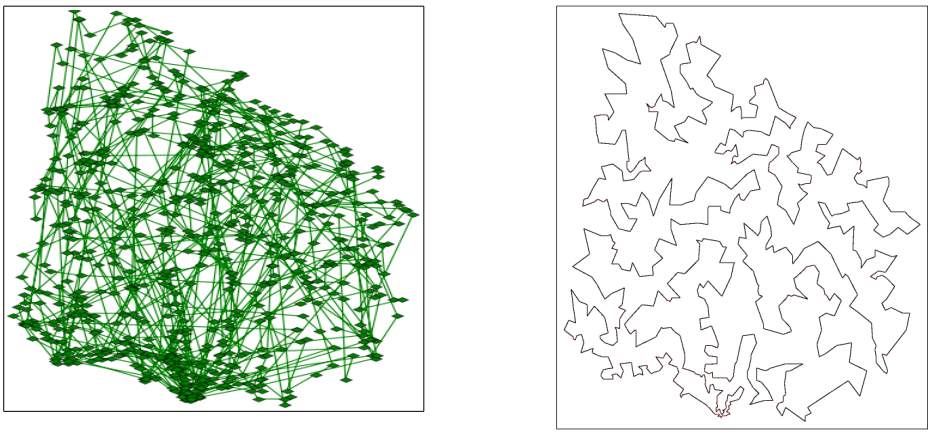


Imagen 30: Izquierda, mejor solución obtenida en la tabla 6 (509460.05). Derecha, solución optima (79114)

4. Conclusiones

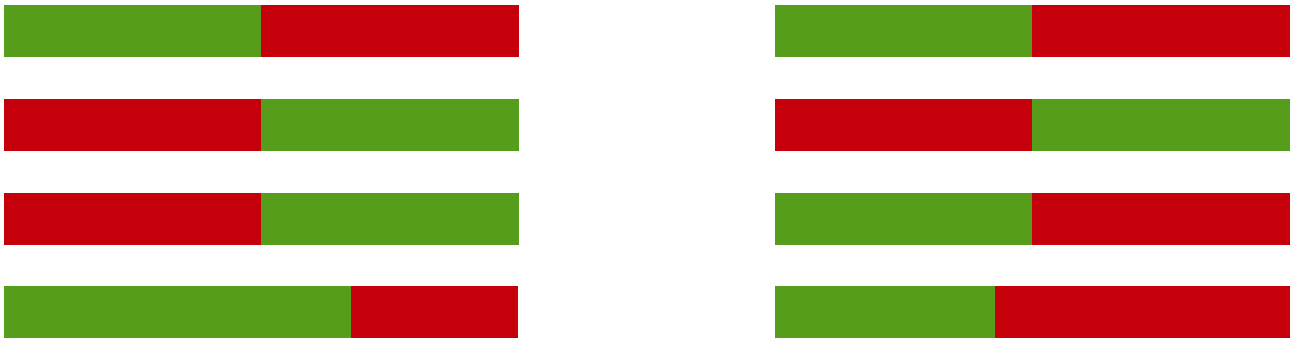
La métrica en este problema es la relación entre el tiempo de computo y la solución obtenida de la instancia del problema, se puede trabajar con tiempo de computo siempre y cuando las pruebas se realicen con el mismo ordenador, pero el tiempo de computo es un parámetro dependiente de cada maquina y no nos permite comparar nuestra solución con soluciones obtenida desde otras máquinas, por este motivo sustituimos el tiempo de computo por el número de nodos evaluados en una ejecución.
El algoritmo genético que se ha implementado en este trabajo siempre tiene el mismo número de individuos en la población de cada generación, por lo tanto el número de nodos evaluados en una ejecución se calcula de la siguiente forma:

nodos_evaluados = n_generaciones * n_individuos_población

La evaluación en este trabajo está orientada en estudiar la diferencia entre una metaheurística centrada en la exploración (en nuestro caso cruce uniforme) con una metaheurística centrada en la explotación (en nuestro caso cruce por corte en un punto). Observamos que aplicando un cruce uniforme los hijos que obtenemos contienen poca información de los padres, esto provoca que las soluciones que proporcionan los hijos tengan un factor de aleatoriedad muy elevado realizando saltos continuamente en el espacio de búsqueda de soluciones (+ exploración y - explotación). Las tablas obtenidas en el apartado de evaluación referentes al cruce uniforme generalmente mejoran los resultados obtenidos cuando se tienen mas individuos en la población y el número de generaciones es mayor, es decir, cuando el número de nodos_evaluados es mayor se realizan un mayor número de saltos en el espacio de búsqueda y la probabilidad de obtener una solución mejor es mayor, hablamos de forma general porque pueden darse ejecuciones en las que la aleatoriedad no nos proporcione un conjunto de soluciones buenas y por ello no podemos garantizar matemáticamente que a mayor número de individuos por población y mayor número de generaciones obtengamos “siempre” una solución mejor, lo podemos comprobar en las tablas de cruce uniforme viendo que tenemos soluciones con número de individuos por población igual a 20 y número de generaciones igual a 30000 mejores que las correspondientes soluciones con número de individuos igual a 30 y generaciones igual a 30000.
En el diseño del algoritmo con cruce de corte por un punto lo que buscamos es que los hijos generados contengan la mayor parte de información de los padres siguiendo el criterio de búsqueda mas informada, este criterio teóricamente parece más inteligente pero existen inconvenientes al aplicar esta búsqueda mas informada en el problema del viajante de comercio, los comentamos a continuación ya que es necesario conocerlos para comprender los resultados obtenidos de la tablas de cruce por corte de un punto.

1) Imaginemos que tenemos dos padres donde el color verde nos indica la parte de la solución del padre que es buena y en rojo la parte de la solución del padre que es mala:

Este cruce por corte de un punto nos devolvería un hijo que hereda las dos partes de los padres donde la solución es muy buena, pero se puede dar los casos contrarios, es decir:



...

Una posible mejora a este problema sería realizar un cruce por corte de dos puntos, si analizamos los casos que se pueden producir observamos que también nos proporcionaría problemas:

Imaginemos que el hijo hereda las dos partes extremas del padre de la izquierda y la parte central del padre de la izquierda:

Un trabajo futuro para solucionar este problema consistiría en implementar algoritmos con cruce por corte de n puntos y lanzar ejecuciones y llegar a una serie de conclusiones, por desgracia no se dispone de tanto tiempo para realizar este trabajo, pero es interesante estudiar los problemas de nuestro algoritmo y proponer posibles soluciones.

2) Al aplicar la búsqueda mas informada que hemos propuesto mediante el cruce por corte de un punto, para el problema del viajante de comercio también existe otro problema que es también perjudicial, el problema se encuentra en la combinación de los padres para obtener nuevos hijos, ya que al combinar nodos (ciudades) de diversos padres la solución de los hijos puede contener nodos (ciudades) repetidos. En nuestro diseño se ha optado por corregir los nodos repetidos añadiendo por cada nodo repetido un nodo que no aparezca en la solución, esta corrección provoca la perdida de información de los padres aplicando un grado de aleatoriedad a la solución. La corrección puede ser mala, en el caso de que se desconfigure mucho la parte de la solución que hereda el hijo del padre, o puede ser buena en el caso de que la parte que hereda el hijo del padre sea una parte mala de la

configuración del padre y la intente mejorar de forma aleatoria. Una alternativa a este problema consistiría en descartar a los hijos generados que contengan nodos repetidos en su solución, esta alternativa eliminaría la parte aleatoria que hemos descrito pero existirían muchas generaciones en las cuales no se realizaría ningún remplazo en la población y por lo tanto la evolución en el algoritmo genético sería mucho mas lenta.

En el algoritmo genético utilizando cruce por corte de un punto observamos en las tablas que también mejora la solución conforme incrementa el número de n_nodos evaluados.
Es interesante comparar las soluciones de ambos diseños de cruce, cuando trabajamos con un número de individuos de población igual a 5 se puede ver claramente en los resultados la diferencia entre exploración y explotación, en el cruce uniforme observamos que las soluciones varían muy poco para cada ejecución con número de generaciones distintas, esto se debe a que en cada generación solo aparecen dos hijos nuevos y la exploración es muy lenta, en cambio para cruce por corte de un punto observamos que con 5 individuos en la población las soluciones van mejorando conforme aumentamos el número de generaciones, por el motivo de que los hijos obtenidos están mas informados. Si aumentamos el número de individuos de la población observamos que ocurre justo lo contrario, a mayor número de individuos por población y a mayor número de generaciones el cruce uniforme mejora considerablemente, esto se produce porque crece rápidamente la exploración en la búsqueda, en cambio en el cruce por corte de un punto observamos que las soluciones aunque mejoren al aumentar el número de individuos por población y el número de generaciones, lo hacen de forma mas lenta obteniendo peores resultados respecto al cruce uniforme, esto se debe a que la búsqueda es mas lenta a causa de la explotación.
Es importante estudiar la talla del problema (número de ciudades), observamos que cuando la talla del problema es mas pequeña el algoritmo genético obtiene soluciones mas cercanas a la solución optima (wi29), esto se debe a que trabajamos en un espacio de búsqueda mas pequeño. En el caso de tener un problema con una talla muy elevada (uy734) el espacio de búsqueda explota y observamos que las mejores soluciones obtenidas se alejan en mayor grado a la solución optima.

ENFRIAMIENTO SIMULADO

1. Diseño del algoritmo

El diseño de un algoritmo de enfriamiento simulado es muy sencillo de implementar, vemos su estructura en la siguiente ilustración:

Procedimiento de Enfriamiento simulado

$S_{ACTUAL} := S_{INICIAL}$; $S_{MEJOR} := S_{ACTUAL}$; $T := T_{INICIAL}$; I=0;

Mientras *"Existan sucesores de* S_{ACTUAL}*"* y *"criterio_terminación=falso"* **hacer**

I=I+1;

$S_{NUEVO} \leftarrow$ Operador-aleatorio (S_{ACTUAL}) *;movimiento aleatorio, no el mejor!*

$\Delta f \leftarrow f(S_{NUEVO}) - f(S_{ACTUAL})$

si $\Delta f > 0$ entonces *{mejora la solución: se acepta el movimiento}*

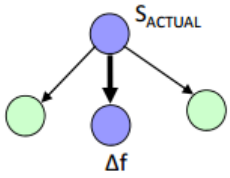
$S_{ACTUAL} \leftarrow S_{NUEVO}$

Si S_{NUEVO} mejor que S_{MEJOR} entonces $S_{MEJOR} := S_{NUEVO}$

si no

$S_{ACTUAL} \leftarrow S_{NUEVO}$ con probabilidad $e^{\Delta f/T}$

$T = \alpha(I, T)$ *; Actualizar T acorde planificación del enfriamiento, por ejemplo, T=T/(1+kT), siendo 0<k<1*



finMientras

Devolver S_{MEJOR}

Imagen 31: Estructura de algoritmo de enfriamiento simulado.

La estructura del algoritmo nos muestra los pasos a seguir de forma genérica pero hemos de adaptarlo al problema que queremos resolver, para ello vamos a exponer nuestro diseño para resolver el problema de viajante de comercio (TSP):

NOTA: La representación y la función fitness es la misma que para el algoritmo genético.

1.1 Condición de parada

La condición de parada en nuestro algoritmo depende del número de iteraciones, el número de iteraciones es introducido como parámetro en el programa.

1.2 Generación de vecinos de la s_actual

Dada una solución actual del problema se generan un conjunto de sucesores (soluciones vecinas) que nos permiten elegir de forma aleatoria una solución nueva al problema, esta generación de vecinos y la selección de la nueva solución es un factor importante dentro del algoritmo porque la

selección de la nueva solución puede ser mas informada o menos informada teniendo siempre en cuenta la importancia de que la selección también sea estocástica para evitar caer en soluciones locales. En nuestro algoritmo la solución de vecinos se realiza de la siguiente manera:
Dada una solución en ese momento (actual) del problema:

s_actual = [1,2,3,0,1]

Generamos un conjunto de vecinos, cuyo número viene definido por un parámetro de entrada al programa, la generación de cada vecino se realiza de la siguiente forma:

1. Eliminamos el último elemento = [1,2,3,0]
2. Realizamos un corte de un punto e intercambiamos las partes: [1,2,3,0] → [3,0,1,2]
3. Aplicamos un intercambio reciproco entre dos nodos de forma aleatoria:
   $pos_1 = 0$
   $pos_2 = 2$
   [3,0,1,2] → [1,0,3,2]
4. Indicamos que el nodo final es el nodo inicial: [1,0,3,2,1]

1.3 Selección de un vecino del conjunto de vecinos de la s_actual

Dado el conjunto de vecinos generado en el paso anterior, una forma trivial de elegir la nueva solución sería elegir de forma aleatoria, con la misma probabilidad de selección, uno de los vecinos generados, pero parece mas inteligente seleccionar un subconjunto de vecinos con mejor función de evaluación (fitness) y de dicho subconjunto elegir un vecino de forma aleatoria con una probabilidad equiprobable (misma probabilidad de selección):
El criterio que se ha seguido es coger ⅓ de los vecinos generados con menor fitness, siendo el número de vecinos generados un valor pasado como parámetro al programa como ya he comentado anteriormente. Ese subconjunto de vecinos seleccionados lo he denominado vecinos_candidatos, la nueva solución es la selección de forma aleatoria (equiprobable) de un vecino que pertenece a vecinos_candidatos.

1.4 Incremento de energía: Mejora de la solución (minimizar o maximizar)

En el esquema del algoritmo de enfriamiento simulado se describe la solución a un problema de maximización, el problema de viajante de comercio (TSP) es un problema de minimización por lo tanto la mejora de la solución se realiza de la siguiente forma:
$\Delta f \leftarrow f(s\_actual) - f(s\_nueva)$
$\Delta f > 0$: mejora la solución (minimizar)

1.5 Técnica de disminución de la temperatura

Se ha optado por utilizar la técnica de disminución de la temperatura:

$$\alpha(i, T) = k*Ti, \text{ con } 0<k<1, \text{ típicamente } \{0,8, 0,99\}$$

Donde α es el decremento.

2. Implementación

Parámetros de entrada que nos permiten varias las opciones de ejecución en el programa python2.7 “viajante_vecinos.py”

```
# ------------------------------------------------------------------------------------
# PARAMETROS DEL ALGORITMO
# ------------------------------------------------------------------------------------
# Comprobacion de parametros de entrada
if len(sys.argv) != 7:
    print "-------------------------------------------------------------------------------------------------"
    print "Uso: python2.7 viajante_vecinos.py <f_tsp> <n_ite> <T> <k> <n_vecinos> <verbose>"
    print "-------------------------------------------------------------------------------------------------"
    print "<f_tsp>: Fichero con las coordenadas del problema TSP (string)"
    print "<n_ite>: Iteraciones en el bucle (int)"
    print "<T>: Valor de Temperatura (float)"
    print "<k>: Factor k para decrementar temperatura [0...1]     (float)"
    print "<n_vecinos>: Cantidad de sucesores para la solucion actual (int)"
    print "<verbose>: Verbose por pantalla (boolean)"
    sys.exit(0)

# Parametros generales para el algoritmo enfriamiento simulado
f_tsp = sys.argv[1]
n_ite = int(sys.argv[2])
T = float(sys.argv[3])
k = float(sys.argv[4])
n_vecinos = int(sys.argv[5])
modo_verbose = True if sys.argv[6] == "True" else False
```

Imagen 32: Parámetros de entrada para el programa “viajante_vecinos.py”.

El programa trabaja con ficheros TSPLIB del mismo modo que hemos explicado en la implementación del algoritmo genético, la creación de EDA's es la siguiente:

```
# ------------------------------------------------------------------------------------
# GENERACION DE EDAs PARA EL ALGORITMO
# ------------------------------------------------------------------------------------
# Fichero con coordenadas del mapa
nombre_f = f_tsp

# Abrimos el fichero para trabajar con el
fichero = open(nombre_f, 'r')

# Insertamos los datos del fichero en una lista para trabajar con ella
# dato[0] = nodo; dato[1] = x; dato[2] = y;
datos = []
for linea in fichero:
    datos_linea = linea.split()
    datos.append(datos_linea)

# Creamos EDAs para trabajar con el algoritmo genetico
nodos = [] # nodos = ciudades codificadas en numeros
distancias = [] # Distancia de cada nodo con todos los demas
for dato_actual in datos:
    nodos.append(int(dato_actual[0])-1) # Insertamos los nodos
    x_actual = float(dato_actual[1])
    y_actual = float(dato_actual[2])
    # Calculo de distancia euclidea de un nodo a los demas nodos
    aux_distancias = []
    for dato_siguiente in datos:
        x_siguiente = float(dato_siguiente[1])
        y_siguiente = float(dato_siguiente[2])
        x = x_siguiente-x_actual
        x = math.pow(x,2)
        y = y_siguiente-y_actual
        y = math.pow(y,2)
        aux_distancias.append(math.sqrt(x+y))
    distancias.append(aux_distancias)
```

Imagen 33: Creación de EDA's para trabajar con los datos.

A continuación mostramos la implementación de las funciones que utiliza el algoritmo por enfriamiento simulado:

```python
# fitness = distancia de la secuencia
def f_fitness(individuo,distancias):
    d = 0
    for i in range(0,len(individuo)-1):
        nodo_actual = individuo[i]
        nodo_siguiente = individuo[i+1]
        d = d + distancias[nodo_actual][nodo_siguiente]
    return d

# Genera un individuo (solucion) de forma aleatoria
def generar_solucion(nodos):
    individuo = []
    for j in range(len(nodos)):
        nodo = random.randint(0,len(nodos)-1)
        while nodo in individuo: # No se repiten los nodos en un individuo
            nodo = random.randint(0,len(nodos)-1)
        individuo.append(nodo)
    individuo.append(individuo[0])
    return individuo

# Genera n_vecinos a partir de la s_actual
def generar_vecinos(s_actual,n_vecinos):
    lista_vecinos = []
    particion = int(round(len(s_actual)/2))
    for i in range(n_vecinos):
        vecino = []
        vecino += s_actual[particion:len(s_actual)-1]
        vecino += s_actual[0:particion]
        # Intercambio reciproco de dos nodos aleatorios
        pos1 = random.randint(0,len(s_actual)-2)
        pos2 = random.randint(0,len(s_actual)-2)
        while pos2 == pos1:
            pos2 = random.randint(0,len(s_actual)-2)
        aux = vecino[pos1]
        vecino[pos1] = vecino[pos2]
        vecino[pos2] = aux
        # El nodo inicial ha de ser igual al final
        vecino.append(vecino[0])
        lista_vecinos.append(vecino)
    return lista_vecinos

# Generar un plot de un individuo
def generar_plot_individuo(individuo,datos):
    # Plot de la solucion obtenida
    # datos[0] = nodo; datos[1] = x; datos[2] = y
    x = []
    y = []
    for i in individuo:
        x.append(datos[i][1])
        y.append(datos[i][2])
    plt.plot(x,y,'g-d')
    plt.show()
```

Imagen 34: Funciones necesarias para implementar el algoritmo por enfriamiento simulado.

- f_fitness(individuo,distancias): Calcula la función fitness (distancia) dado un individuo y una lista de distancias que corresponde a la segunda EDA que hemos generado, que guarda para cada nodo la distancia euclidea con respecto a los demás nodos.
- generar_solucion(nodos): Genera una solución de forma aleatoria utilizando la EDA de lista de nodos obtenida mediante el fichero TSPLIB.
- generar_vecinos(s_actual,n_vecinos): Genera n_vecinos a partir de la solución actual aplicando el criterio que se ha explicado en el apartado de diseño correspondiente a "generación de vecinos de la solución actual".
- generar_plot_individuo(individuo,datos): Genera un plot de una solución (individuo) introducida como parámetro a la función.

Empezamos con la implementación del algoritmo por enfriamiento simulado, a continuación se muestra los parámetros iniciales y la condición de parada del algoritmo:

```
# ------------------------------------------------
# ALGORITMO ENFRIAMIENTO SIMULADO
# ------------------------------------------------
s_actual = generar_solucion(nodos)
s_mejor = s_actual
puntuacion_s_mejor = f_fitness(s_mejor,distancias)
I = 0
# Bucle Principal
while I < n_ite:
    I = I+1
    if modo_verbose:
        print "-----------------------------------"
        print "S_ACTUAL:"
        print "-----------------------------------"
        print s_actual
```

Imagen 35: Parámetros iniciales y condición de parada del algoritmo por enfriamiento simulado.

Generamos nuevos vecinos a partir de la solución actual y calculamos el fitness de cada vecino calculado:

```
# s_nuevo a partir de los vecinos de s_actual
vecinos = generar_vecinos(s_actual,n_vecinos)
fitness_vecinos = []
for vecino in vecinos:
    fitness_vecinos.append(f_fitness(vecino,distancias))
fitness_vecinos_ord = sorted(fitness_vecinos) # Ordena de menor a mayor
if modo_verbose:
    print "-----------------------------------"
    print "VECINOS:"
    print "-----------------------------------"
    print vecinos
    print "-----------------------------------"
    print "FITNESS VECINOS:"
    print "-----------------------------------"
    print fitness_vecinos
```

Imagen 36: Generación de nuevos vecinos.

Nos quedamos con un vecino siguiendo el criterio que se ha explicado en la parte de diseño de la memoria:

```
# Nos quedamos con 1/3 de los mejores vecinos (min f_fitness)
particion = int(round(len(vecinos)/3.0))
vecinos_candidatos = []
for j in range(particion):
    vecinos_candidatos.append(vecinos[fitness_vecinos.index(fitness_vecinos_ord[j])])
if modo_verbose:
    print "-----------------------------------"
    print "VECINOS CANDIDATOS:"
    print "-----------------------------------"
    print vecinos_candidatos
# De los vecinos seleccionados (candidatos) escogemos uno aleatorio
indice_aleatorio = random.randint(0,len(vecinos_candidatos)-1)
s_nuevo = vecinos_candidatos[indice_aleatorio]
if modo_verbose:
    print "-----------------------------------"
    print "VECINO SELECCIONADO (S_NUEVO):"
    print "-----------------------------------"
    print "Indice aleatorio: ",indice_aleatorio
    print "vecino seleccionado (s_nuevo): ",s_nuevo
```

Imagen 37: Selección del vecino del conjunto de vecinos candidatos generado.

Obtenemos el incremento de la función fitness y decidimos que hacer con la solución nueva tal y como se ha indicado en la estructura del algoritmo:

```
probabilidad = ""
incr_f =  f_fitness(s_actual,distancias) - f_fitness(s_nuevo,distancias)
if incr_f > 0: # mejora la solucion
    s_actual = s_nuevo
    if f_fitness(s_nuevo,distancias) < puntuacion_s_mejor:
        s_mejor = s_nuevo
        puntuacion_s_mejor = f_fitness(s_nuevo,distancias)
else:
    if (T > 0): # Si T es 0 no se acepta directamente
        # Calculamos la probabilidad = e^(incr_f/T)
        e = math.e
        incr_f = round(incr_f,4) # Nos quedamos con 4 decimales
        incr_f = math.sqrt(incr_f*-1)*-1 # Tratamos los datos
        probabilidad = math.pow(e,(incr_f/T))
        probabilidad = round(probabilidad,4) # Nos quedamos con 4 decimales
        v_aleatorio = random.random(); # Valor entre 0.0 y 1.0
        if probabilidad > v_aleatorio: # nos quedamos con la solucion
            s_actual = s_nuevo
            # Decrementamos la Temperatura
            T = round(k*T,4) # Nos quedamos con 4 decimales
if modo_verbose:
    print "-----------------------------------"
    print "PARAMETROS:"
    print "-----------------------------------"
    print "INCREMENTO_F: ", incr_f
    print "Temperatura: ", T
    print "Probabilidad: ", probabilidad
```

Imagen 38: Se toma la decisión de quedarnos con la nueva solución o no.

Al finalizar el bucle de generaciones nuestro programa devuelve por pantalla la solución que corresponde al mejor individuo que hemos obtenido, mostramos por pantalla el individuo obtenido y su correspondiente función fitness.
También se ha implementado que nos muestre el individuo obtenido mediante un plot para poder comparar nuestra solución con la mejor solución encontrada por las paginas que nos han proporcionado los ficheros TSPLIB.

```
# ------------------------------------------------------
# SALIDA POR PANTALLA
# ------------------------------------------------------
# Mostramos el mejor individuo obtenido en la ejecucion
print "-----------------------------------"
print "MEJOR INDIVIDUO"
print "-----------------------------------"
print "Individuo: ",s_mejor
print "Puntuacion : ",puntuacion_s_mejor

# Generamos el mejor individuo obtenido
generar_plot_individuo(s_mejor,datos)
```

Imagen 39: Salida por pantalla la mejor solución encontrada por enfriamiento simulado.

NOTA: Para poder plotear el resultado se tiene que instalar el modulo matplotlib de python2.7, si no está instalado hay dos opciones, instalarlo mediante el comando pip, o comentar la última línea de código “generar_plot_individuo(mejor_individuo,datos)”.

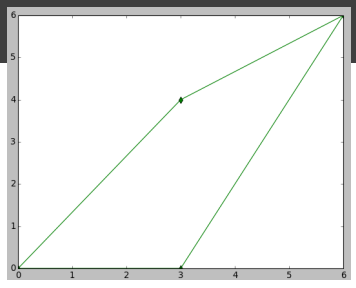
Para ejecutar nuestro programa necesitamos saber que parámetros hemos de introducirle como entrada y en que orden, para ello simplemente ejecutamos el nombre del programa y nos proporciona la información que requiere:

```
pascu@acer ~/Escritorio/TIA/EnfriamientoSimulado $ python2.7 viajante_vecinos.py
-------------------------------------------------------------------------------------------
Uso: python2.7 viajante_vecinos.py <f_tsp> <n_ite> <T> <k> <n_vecinos> <verbose>
-------------------------------------------------------------------------------------------
<f_tsp>: Fichero con las coordenadas del problema TSP (string)
<n_ite>: Iteraciones en el bucle (int)
<T>: Valor de Temperatura (float)
<k>: Factor k para decrementar temperatura [0...1]     (float)
<n_vecinos>: Cantidad de sucesores para la solucion actual (int)
<verbose>: Verbose por pantalla (boolean)
```

Imagen 40: Parámetros de entrada del algoritmo por enfriamiento simulado.

Vamos a realizar dos ejemplos de ejecución para familiarizarnos con el programa, una ejecución va a ser sin modo verbose y la otra con modo verbose. Vamos a utilizar el mismo fichero TSPLIB trivial que hemos visto en este apartado de la implementación del algoritmo genético, vamos a obtener una única iteración, con una temperatura = 100, factor k = 0.8, y se generan 5 vecinos sobre cada solución actual.

```
pascu@acer ~/Escritorio/TIA/EnfriamientoSimulado $ python2.7 viajante_vecinos.py ../data/basico.data.txt 1 1000 0.8 5 False
-----------------------------------
MEJOR INDIVIDUO
-----------------------------------
Individuo:  [3, 0, 1, 2, 3]
Puntuacion :  18.313755208
```



```
pascu@acer ~/Escritorio/TIA/EnfriamientoSimulado $ python2.7 viajante_vecinos.py ../data/basico.data.txt 1 1000 0.8 5 True
------------------------------------
S_ACTUAL:
------------------------------------
[1, 3, 0, 2, 1]
------------------------------------
VECINOS:
------------------------------------
[[1, 2, 0, 3, 1], [0, 3, 1, 2, 0], [3, 2, 1, 0, 3], [0, 3, 1, 2, 0], [0, 1, 2, 3, 0]]
------------------------------------
FITNESS VECINOS:
------------------------------------
[24.19348530673794, 24.19348530673794, 18.31375520796336, 24.19348530673794, 18.31375520796336]
------------------------------------
VECINOS CANDIDATOS:
------------------------------------
[[3, 2, 1, 0, 3], [3, 2, 1, 0, 3]]
------------------------------------
VECINO SELECCIONADO (S_NUEVO):
------------------------------------
Indice aleatorio:  0
vecino seleccionado (s_nuevo):  [3, 2, 1, 0, 3]
------------------------------------
PARAMETROS:
------------------------------------
INCREMENTO_F:  5.87973009877
Temperatura:  1000.0
Probabilidad:
------------------------------------
MEJOR INDIVIDUO
------------------------------------
Individuo:  [3, 2, 1, 0, 3]
Puntuacion :  18.313755208
```

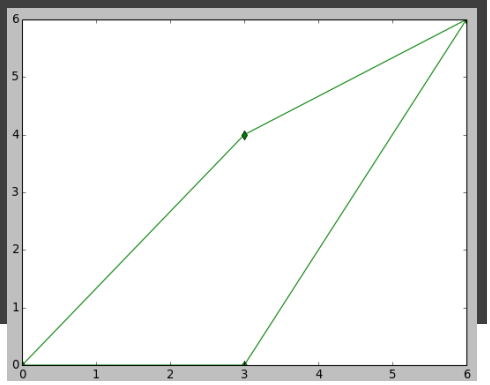


Imagen 41: Ejecución en modo no verbose y verbose del programa “viajante_vecinos.py”.

3. Evaluación

Para evaluar el algoritmo por enfriamiento simulado se han realizado pruebas con diferentes valores de temperatura y mediante un conjunto de soluciones se ha escogido el valor de la temperatura que mejores resultados proporcionaba. Se ha realizado un barrido de iteraciones especificadas por columnas y un barrido de número de vecinos generados especificado por filas, cada valor en las celdas representa la solución mediana respecto a 5 ejecuciones con los mismos parámetros con la finalidad de quedarnos con resultas mas reales. Se ha empleado una temperatura = 1000 y un factor de decremento k = 0.8, estos valores los hemos obtenido de forma empírica realizando un conjunto de pruebas.

PointSet Western Sahara (wi29.data.txt):

Temperatura = 1000				
Decremento de temperatura = α(i, T) = k*Ti, con k = 0.8				
Nº Iteraciones	100	1000	10000	30000
5	55167.18	35261.99	35285.11	36147.77
10	48572.25	34624.81	35326.93	35292.76
20	53747.09	36254.18	37123.74	36362.19
30	50957.93	37546.24	32985.35	37986.70

Tabla 7: Resultados de algoritmo enfriamiento simulado para Western Sahara.

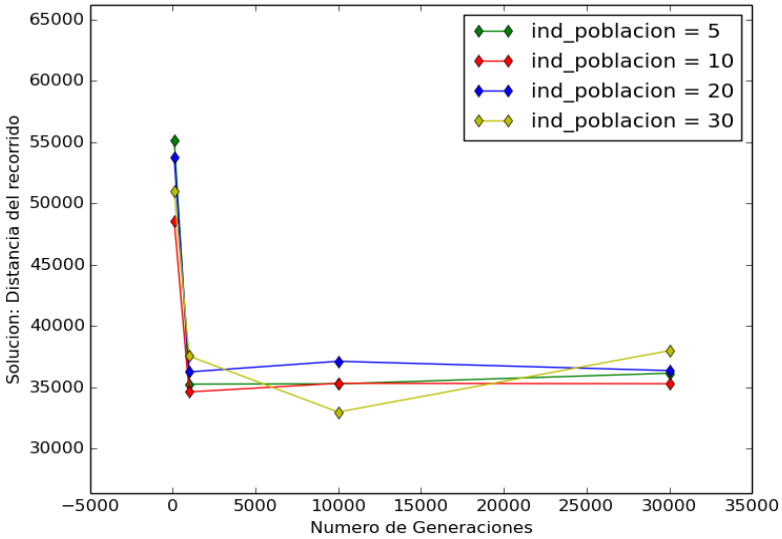


Imagen 42: Gráfica de los datos de la Tabla 7.

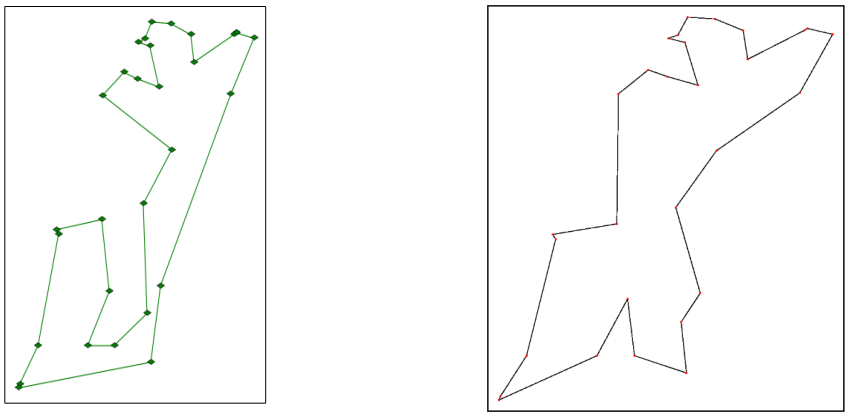


Imagen 43: Izquierda, mejor solución obtenida (32985.35). Derecha, solución optima (27603)

PointSet Qatar (Qa194.data.txt):

Temperatura = 1000				
Decremento de temperatura = α(i, T) = k*Ti				
Nº Iteraciones	100	1000	10000	30000
5	67940.70	38855.30	24627.27	20846.49
10	63667.17	35915.64	23448.33	20405.30
20	65068.63	39507.99	23351.12	20387.03
30	64270.63	36338.74	23125.35	21543.60

Tabla 8: Resultados de algoritmo enfriamiento simulado para Qatar.

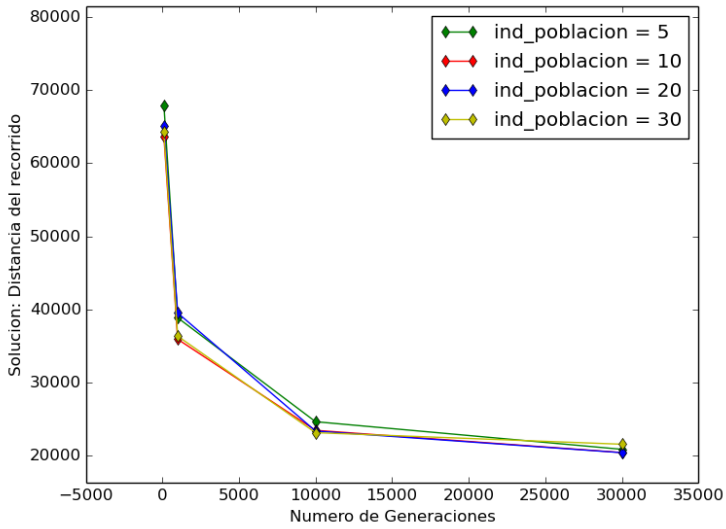


Imagen 44: Gráfico de los datos de la Tabla 8.

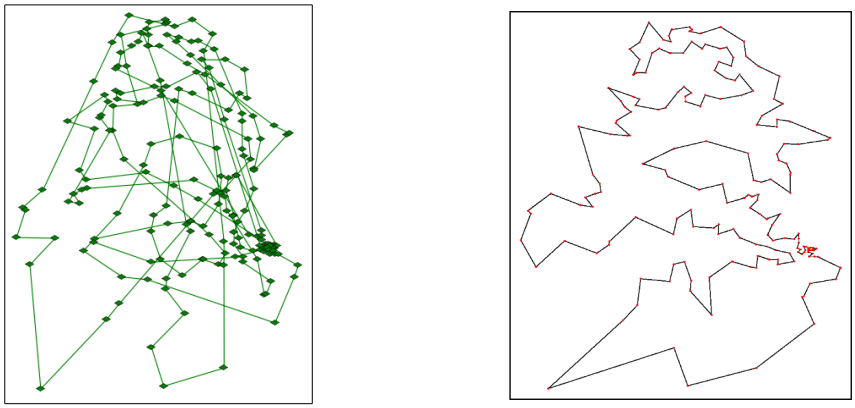


Imagen 45: Izquierda, mejor solución obtenida (20387.03). Derecha, solución optima (9352)

PointSet Uruguay (uy734.data.txt):

Temperatura = 1000				
Decremento de temperatura = α(i, T) = k*Ti				
Nº Iteraciones	100	1000	10000	30000
5	1445875.73	969411.16	558142.24	432948.95
10	1443772.48753	908390.92	513217.57	410613.51
20	1439161.35436	930820.23	528336.21	413190.22
30	1426086.94741	935928.01	530564.70	407707.82

Tabla 9: Resultados de algoritmo enfriamiento simulado para Uruguay.

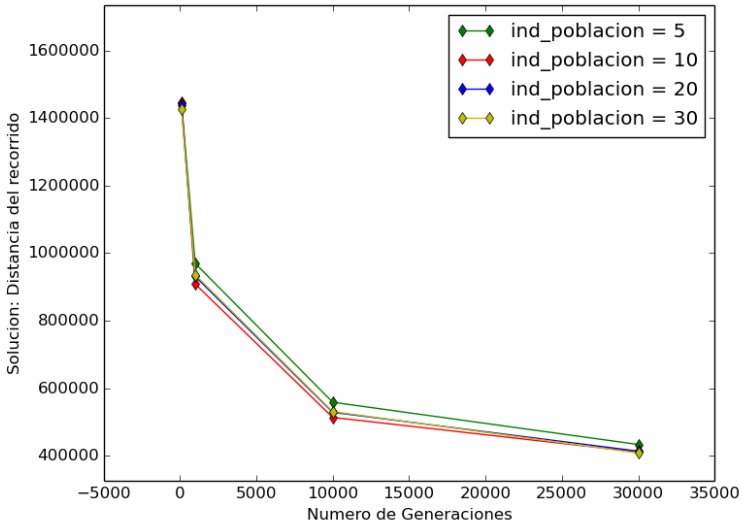


Imagen 46: Gráfico de los datos de la Tabla 9.

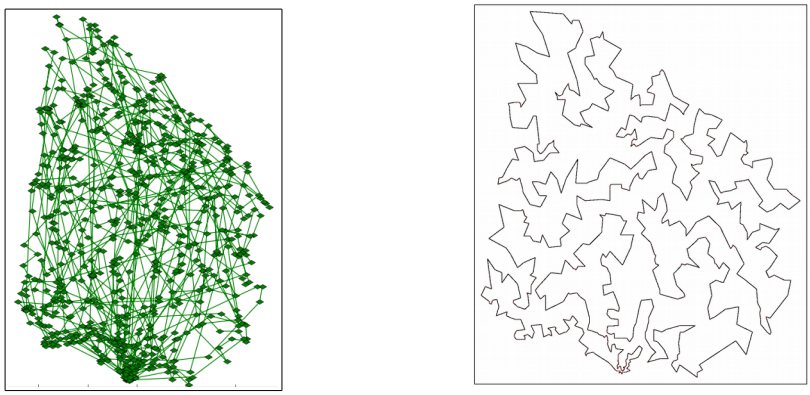


Imagen 47: Izquierda, mejor solución obtenida (407707.82). Derecha, solución optima (79114)

4. Conclusiones

En la evaluación al utilizar un valor de temperatura de 1000 (la temperatura se ha elegido empíricamente realizando pruebas) y un factor de decremento de temperatura de 0.8 (parámetro también obtenido de forma empírica) hemos obtenido ejecuciones donde el algoritmo de enfriamiento simulado realiza un nivel similar de exploración y explotación sobre el conjunto de soluciones de cada instancia de problema. Nuestro diseño del algoritmo de enfriamiento simulado tiene una búsqueda inteligente que se obtiene con la generación de vecinos y la selección estocástica de un conjunto acotado de los mejores vecinos obtenidos (vecinos candidatos). El diseño es similar al diseño del algoritmo genético con cruce por corte de un punto pero al generar los vecinos a partir de una única solución evitamos la corrección de nodos repetidos. Una característica muy importante que tiene el algoritmo por enfriamiento simulado es que mediante los parámetros de temperatura y el factor de decremento, para un valor determinado de iteraciones, podemos indicar si queremos que la ejecución tenga un mayor grado de exploración, un mayor grado de explotación o que la exploración y la explotación tengan un grado similar.
Se han realizado pruebas en las cuales se utilizaba un mayor grado de exploración que de explotación y como era de esperar las soluciones eran cada vez diferentes, por lo tanto no se podía realizar una comparativa. Se han realizado pruebas aplicando un mayor grado de explotación que de exploración y los resultados han sido peores soluciones porque la búsqueda es mucho mas lenta, por estos motivos en la evaluación hemos utilizado una temperatura de 1000 y un factor de escala k = 0.8 que nos ha proporcionado los mejores resultados por el motivo de que tenemos en cuenta la exploración como la explotación en el espacio de soluciones.
Como hemos comentado en el algoritmo genético nos interesa trabajar con nodos evaluados en vez de con tiempo de computo, en enfriamiento simulado los nodos_evaluados se calculan:

nodos_evaluados = n_vecinos * iteraciones

En las tablas observamos que a mayor número de nodos_evaluados la solución tiende a ser mejor, es lo esperado, ya que exploramos y explotamos un conjunto de soluciones mucho mas amplias.
Es importante estudiar la talla del problema (número de ciudades), observamos que cuando la talla del problema es mas pequeña el algoritmo por enfriamiento simulado obtiene soluciones mas cercanas a la solución optima (wi29), esto se debe a que trabajamos en un espacio de búsqueda mas pequeño. En el caso de tener un problema con una talla muy elevada (uy734) el espacio de búsqueda explota y observamos que las mejores soluciones obtenidas se alejan en mayor grado a la solución optima.

CONCLUSIONES FINALES

En este apartado vamos a comparar las metaheurísticas que hemos diseñado observando cual nos proporciona mejor resultado mediante las evaluaciones que hemos realizado en la memoria.
Como hemos visto la parte del algoritmo genético, el cruce uniforme nos proporciona mejores resultados que el cruce por corte de un punto, esto se debe a que al tener nodos repetidos y al no garantizar que cogemos un subconjunto de solución buena de cada padres los hijos resultantes en el cruce por corte de un punto empeoran y la búsqueda que utiliza es mas lenta (explotación).
Vemos la comparativa de ambos cruces con las mejores soluciones para cada problema:

Problema	Mejor solución cruce uniforme	Mejor solución cruce corte (1 punto)	Solución optima
wi29	33345.62	34809.36	27603
qa194	15810.08	23044.88	9352
uy734	338058.88	509460.05	79114

Tabla 10: AG (cruce uniforme) VS AG (cruce corte 1 punto).

A continuación vamos a comparar el ganador en el algoritmo genético (cruce uniforme) y el algoritmo por enfriamiento simulado:

Problema	Mejor solución AG (cruce uniforme)	Mejor solución Enfriamiento S.	Solución optima
wi29	33345.62	35292.76	27603
qa194	15810.08	20387.03	9352
uy734	338058.88	407707.82	79114

Tabla 11: AG (cruce uniforme) VS Algoritmo por enfriamiento simulado.

Observamos que los mejores resultado nos los proporciona el algoritmo genético utilizando cruce uniforme, es la metaheurísitca con mejores resultados para las evaluaciones que hemos realizado en todo el trabajo.
Es interesante compara también el algoritmo genético con cruce por corte de un punto y el algoritmo de enfriamiento simulado, para ver que algoritmo es el peor de los tres implementados.

Problema	Mejor solución cruce corte (1 punto)	Mejor solución Enfriamiento S.	Solución optima
wi29	34809.36	35292.76	27603
qa194	23044.88	20387.03	9352
uy734	509460.05	407707.82	79114

Tabla 12: AG (cruce corte 1 punto) VS Algoritmo por enfriamiento simulado.

Para wi29 ha ganado el algoritmo genético utilizando cruce por corte de un punto, para qa194 y uy734 ha ganado el algoritmo por enfriamiento simulado, consideramos que es mejor el algoritmo por enfriamiento simulado ya que nos proporciona mejores soluciones en problemas de talla mayor.
Posiciones de mejor a peor metaheurística en este trabajo:

Posición	Metaheurística
1	Algoritmo Genético (cruce uniforme)
2	Algoritmo Enfriamiento Simulado
3	Algoritmo Genético (cruce corte 1 punto)

Tabla 13: Posición de las metaheurísitcas estudiadas según resultados de evaluación.

Es importante darnos cuenta que no existe un criterio para decidir que metaheurística funciona mejor cuando vamos a resolver un problema, es importante conocerlas y compararlas ya que nos permiten tener una visión de como funcionan y nos pueden ayudar a decidir cual utilizar sobre problemas que tengan características similares. También es importante comentar dos aspectos importantes, el primero es que las metaheurísticas se pueden combinar para intentar obtener soluciones mejores, en este trabajo no se ha tenido en cuenta porque se ha estudiado como afecta el factor de explotación y exploración dentro del espacio de soluciones de la instancia del problema, el segundo aspecto viene definido por el modelado de los problemas, es un punto muy importante ya que nos facilita mucho la implementación, de igual modo también es importante el apartado de “tuneado” de los algoritmos que nos permitirá ser mas precioso o menos variando el factor de aleatoriedad por una búsqueda mas inteligente.

AUTOCRÍTICA

La asignatura esta muy bien organizada y he disfrutado mucho realizando este trabajo, por poner cosas negativas, me hubiera gustado disponer de mayor tiempo para realizar implementaciones para mejorar los problemas que han ido apareciendo y que he expuesto en la memoria, pero el tiempo era limitado por el motivo de acabar la asignatura en prácticamente un mes. Creo que el criterio de aprendizaje de la asignatura es muy bueno porque se implementa de forma real lo que vemos en teoría.

ANEXO

Aparte de los algoritmos implementados, se han implementado herramientas que nos permiten hacer las evaluaciones de forma automática obteniendo las tablas que hemos expuesto en las evaluaciones del trabajo, al ser un trabajo que se ha realizado de forma adicional, me ha llevado tiempo implementarlo, considero que es importante incorporarlo como anexo en la memoria.

Herramientas adicionales en el algoritmo genético:

- generar_resultados.sh: Genera resultados realizando 5 ejecuciones por celda de la tabla. (Hay que cambiar el ejecutable dentro del programa para variar de versión de cruce del AG).
- obtener_tablas.py: Genera tablas a partir del fichero creado por generar_resultados.sh
- graficas.py: Genera una gráfica a partir de la tabla obtenida por obtener_tablas.py.

Herramientas adicionales en el algoritmo genético:

- generar_resultados.sh: Genera resultados realizando 5 ejecuciones por celda de la tabla.
- obtener_tablas.py: Genera tablas a partir del fichero creado por generar_resultados.sh
- graficas.py: Genera una gráfica a partir de la tabla obtenida por obtener_tablas.py.

NOTA: Todos los scripts y programas python2.7 implementados muestran su forma de uso dentro del programa o simplemente ejecutándolos sin ningún parámetro desde el terminal.