Algoritmo genético aplicado al problema del agente viajero con ventanas de tiempo (TSPTW)

Ian Martínez
Escuela Colombiana de Ingeniería
Email: ian.martinez@mail.escuelaing.edu.co

Resumen - El problema del agente viajero con ventanas de tiempo (Traveling salesman problem with time windows -TSPTW) es considerado un problema de optimización, el cual consiste en encontrar la ruta óptima para recorrer (*n)* ciudades sin repetirlas, respetando las ventanas de tiempo establecidas para cada visita y volviendo siempre a la ciudad de orgen.

Abstract - El problema del agente viajero con ventanas de tiempo (Traveling salesman problem with time windows -TSPTW) es considerado un problema de optimización, el cual consiste en encontrar la ruta óptima para recorrer (*n)* ciudades sin repetirlas, respetando las ventanas de tiempo establecidas para cada visita y volviendo siempre a la ciudad de origen.

Palabras Clave: TSP, heurística, Algoritmo genético

I. INTRODUCCIÓN

Algunos problemas como el del agente viajero que se encuentra dentro del grupo de problemas combinatoriales, no pueden ser solucionados por medio de métodos exactos, esto debido a que cuando crece el número de variables de decisión del problema y el número de decisiones factibles, el esfuerzo computacional crece en forma exponencial.

II. MARCO TEÓRICO

A. *Problema del agente viajero (TSP)*

El TSP es uno de los problemas de optimización combinatoria más populares en la literatura científica. Propuesta inicialmente por G. Dantzig, R. Fulkerson and S. Johnson *[1]*, consistente en determinar el circuito óptimo que tiene que hacer un Viajante que partiendo desde un origen debe visitar n ciudades una y solo una vez, volviendo al punto de partida. *[2] [3]*

B. *Problema del agente viajero con ventadas de tiempo (TSPTW)*

Es una variante del problema TSP, donde TSPTW busca como objetivo minimizar el coste de la ruta necesaria para visitar a todos los clientes, pero su gran diferencia con el resto de los problemas es que este incluye un periodo de tiempo fijado para cada algoritmo. De esta forma se puede considerar un grafo completo donde los vértices representan a los clientes a visitar y las aristas representan el coste de ir de un punto a otro. Adicional cada vértice posee asociado un intervalo de tiempo el cual debe respetar el vehículo, este intervalo de tiempo es representado por [*ai*,*bi*] donde *ai* representa la hora de apertura de la ventana y *bi* representa la hora de clausura de la ventana. [4]

C. *Algoritmos Genéticos*

Este concepto fue introducido por John Holland en 1962, el cual se define como un método de búsqueda que imita la teoría de la evolución biológica de Darwin para la resolución de problemas. Para ello, se parte de una población inicial de la cual se seleccionan los individuos más capacitados para luego reproducirlos y mutarlos para finalmente obtener la siguiente generación de individuos que estarán más adaptados que la anterior generación. [5] [6]

III. ALGORITMO GENÉTICO

A. *Definición*

Este concepto fue introducido por John Holland en 1962, el cual se define como un método de búsqueda que imita la teoría de la evolución biológica de Darwin para la resolución de problemas. Para ello, se parte de una población inicial de la cual se seleccionan los individuos más capacitados para luego reproducirlos y mutarlos para finalmente obtener la siguiente generación de individuos que estarán más adaptados que la anterior generación. [1]

B. Funcionamiento

Una premisa es conseguir que el tamaño de la población sea lo suficientemente grande para garantizar la diversidad de soluciones. Se aconseja que la población sea generada de forma aleatoria para obtener dicha diversidad. En caso de que la población no sea generada de forma aleatoria sede debe garantizar una cierta diversidad en la población generada. Los pasos básicos de un algoritmo genético son: [5]

- Evaluar la puntuación de cada uno de los cromosomas generados.
- Permitir la reproducción de los cromosomas siendo los más aptos los que tengan más probabilidad de reproducirse.
- Con cierta probabilidad de mutación, mutar un gen del nuevo individuo generado.
- Organizar la nueva población.

1) Codificación de la Variables

Los cromosomas deben contener información acerca de la solución que representan. La codificación se puede realizar de varias formas.

- **Codificación Binaria:** Cada cromosoma representa una cadena de bits (0 o 1), este tipo de codificación usualmente es empleado en el problema de la mochila.

- **Codificación Numérica:** Se utilizan cadenas de números que representan un numero en una secuencia, usualmente se utiliza en problemas que requieran ordenar algo, donde este resulta muy útil. Comúnmente utilizado en el problema del agente viajero.

  Codificación por valor directo: esta codificación es usada en el caso de problemas donde se requiere el uso de valores de cifrado complicado como podría ser en el uso de números reales, cuya codificación con números binarios sería muy complejo. Para este caso cada cromosoma es una cadena de valores relacionados con el problema a estudiar, pudiendo ser desde números decimales, cadenas de caracteres o incluso una combinación de varios de ellos.

- **Codificación en árbol:** Se utiliza principalmente en el desarrollo de programas o expresiones para programación genética. Cada cromosoma será en este caso un árbol con ciertos objetos. [5]

2) Selección de la población inicial

De acuerdo con la teoría de Darwin los individuos más capacitados son los que deben sobrevivir y crear una nueva descendencia más facultada. De acuerdo con esto se deben seleccionar estos individuos para que éstos sean los que se reproduzcan con más probabilidad. Por lo tanto, una vez evaluado cada cromosoma y obtenida su puntuación, se tiene que crear la nueva población teniendo en cuenta que los buenos rasgos de los mejores se transmitan a ésta. [5] Esta selección se puede realizar de varias formascomo se verá a continuación:

- **Selección por ruleta:** Propuesto por DeJong, es posiblemente el método más utilizado desde los orígenes de los Algoritmos Genéticos. A cada uno de los individuos de la población se le asigna una parte proporcional a su medida en una ruleta, de tal forma que la suma de todos los porcentajes sea la unidad. Los mejores individuos reciben una porción de la ruleta mayor que la recibida por los peores. Es un método muy sencillo, pero ineficiente a medida que aumenta el tamaño de la población (su complejidad es O(n2)). [7]

- **Selección por torneo:** Consiste en realizar la selección con base a comparaciones directas entre individuos. Para este caso existen dos métodos: El determinístico, se selecciona al azar un número (n) de individuos. De entre estos se selecciona el más apto para pasarlo a la siguiente generación. En cuanto al método probabilístico, en vez de escoger siempre el mejor se genera un número aleatorio del intervalo [0..1], si es mayor que un parámetro (n) se escoge el individuo más alto y en caso contrario el menos apto. [7]

- **Selección por rango:** En este método a cada individuo se le asigna un rango numérico basado en su aptitud.

- **Selección Jerárquica: L**os individuos atraviesan múltiples rondas de selección en cada generación. Las evaluaciones de los primeros niveles son más rápidas y menos discriminatorias, mientras que los que sobreviven hasta niveles más altos son evaluados más rigurosamente. La ventaja de este método es que reduce el tiempo total de cálculo al utilizar una evaluación más rápida y menos selectiva para eliminar a la mayoría de los individuos que se muestran poco o nada prometedores, y sometiendo a una evaluación de aptitud más rigurosa y computacionalmente más costosa sólo a los que sobreviven a esta prueba inicial.

3) Cruce o Crossover

Se considera el operador de búsqueda más importante, su función es intercambiar el material genético de un de un par de padres produciendo hijos que usualmente difieren de sus padres. El objetivo del cruce es conseguir que el descendiente mejore la aptitud de sus padres. Existen varios tipos de Crossover:

- **Crossover de 1 Punto:** Se seleccionan dos individuos y se cortan sus cromosomas por un punto seleccionado aleatoriamente, con esto se generan dos segmentos diferentes de cada uno de

ellos: la cabeza y la cola. Se intercambian las colas entre los dos individuos para generar los nuevos descendientes.

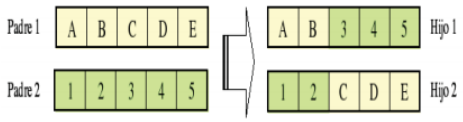

Figura 1. Cruce de 1 punto [7]

- **Crossover de 2 Punto:** Se asemeja a la operación anterior solo que esta vez se realiza el corte en (2) puntos.

- **Crossover Uniforme:** Esta técnica implica la generación de una máscara de cruce con valores binarios. Si en una de las posiciones de la máscara hay un 1, el gen situado en esa posición en uno de los descendientes se copia del primer padre. Si por el contrario hay un 0 el gen se copia del segundo padre. Para producir el segundo descendiente se intercambian los papeles de los padres. [7]

- **Crossover Aritmético:** Los progenitores se recombinan según algún operador aritmético para generar su descendiente.

4) *Mutación*

Esta operación intercambia el valor de un gene de un cromosoma en una población en forma aleatoria, se elige un cromosoma como candidato y se genera un número aleatorio, si el resultado es menor que la tasa de mutación ($p < p_m$) entonces se realiza la mutación. La tasa de mutación se elige del rango [0.001, 0.05]. *[8]*. Usualmente la probabilidad de mutación es muy baja, generalmente menor al 1 %. Esto debido a que los individuos suelen tener un ajuste menor después de mutados. Sin embargo, se realizan mutaciones para garantizar que ningún punto del espacio de búsqueda tenga una probabilidad nula de ser examinado.

5) *Evaluación*

La función de evaluación generalmente es la función objetivo, es decir, es lo que se quiere llegar a optimizar (ej: número de aciertos, número de movimientos, etc. Básicamente corresponde a un método el cual evalúa si los individuos de la población representan o no buenas soluciones al problema planteado. De acuerdo con el tipo de problema se presentan 2 tipos de fitness:

- Maximización: el individuo tiene mayor fitness cuanto mayor es el valor de la función objetivo $f(individuo)$.

- Minimización: el individuo tiene mayor fitness cuanto menor es el valor de la función objetivo $f(individuo)$., o lo que es lo mismo, cuanto mayor es el valor de la función objetivo, menor el fitness. $\frac{1}{1+f(individuo)}$

IV. Solución problema TSPTW por medio de un algoritmo Genético

A. *Contexto*

Se debe diseñar e implementar una heurística o meta-heurística que resuelva el problema del agente viajero con restricciones de ventanas de tiempo, en ingles travelling salesman problem with time windows TSPTW en que el objetivo consiste en reducir el tiempo de viaje (travel time objective). Todas las instancias del problema cumplirán la desigualdad triangular.

El nombre de los casos de prueba se encuentra en la sección 'Benchmark Instances' que se encuentran publicados en http://lopez-ibanez.eu/tsptw-instances.

B. *Pasos para aplicación del algoritmo genético*

Para explicar la solución dada al problema del agente viajero con ventanas de tiempo, se toma como base la definición dada por J. Arraz [5], el cual explica en la siguiente figura los pasos a seguir para solucionar un problema por medio de un algoritmo genético.

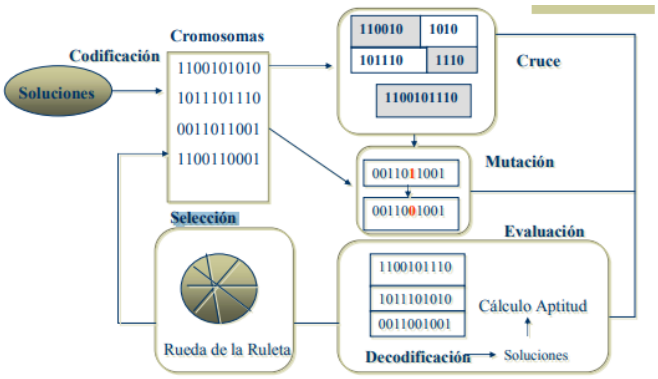

Figura 2. Cruce de 1 punto [7]

1) *Definición de clases*

- 'City': Esta clase esta encargada de suministrar la información acerca de las ciudades y distancia entre las mismas.
- 'Fitness': esta clase cumple 2 funciones, la primera es calcular la distancia final de una ruta ingresada teniendo en cuenta las restricciones de tiempo y distancias entre ciudades, por otra parte, la segunda función calcula el valor fitness de la ruta creada de acuerdo con la distancia.

2) *Codificación*

Para este caso se implementó la *codificación numérica* ya que es útil en problemas de ordenación como este donde se requiere indicar el orden de las ciudades a visita.

3) *Selección*

Generación Inicial: Para este caso se toma una solución aleatoria del problema de acuerdo con las ciudades ingresadas con las siguientes funciones:

```python
#Crear poblacion inicial
def initialPopulation(popSize, cityList):
    population = []

    for i in range(0, popSize):
        population.append(createRoute(cityList))
    return population

#Creacion de ruta aleatoria
def createRoute(cityList):
    route = random.sample(cityList, len(cityList))
    return route
```

Figura 3. Generación Inicial

Posterior a este proceso se calcula la aptitud o (fitness) de esta y cada generación por medio de la función fitness.

```python
def routeDistance(self):
    if self.distance ==0:
        pathDistance = 0

        for i in range(0, len(self.route)):
            fromCity = self.route[i]
            toCity = None
            if i + 1 < len(self.route):
                toCity = self.route[i + 1]
            else:
                toCity = self.route[0]

            if(ordervini[i][0] > pathDistance or  pathDistance > ordervini[i][1]):
                penality = 50
            else:
                penality = 0

            pathDistance += fromCity.distance(toCity) + penality
            #print(fromCity,'->',toCity,' = ',pathDistance,' wt: ',ordervini[i][0],
        self.distance = pathDistance
        #print(pathDistance, penality)
        #print(self.distance,'\n\n')
    return self.distance

#Se asigna el mejor fitness a la ruta que tenga la menor distnacia incluyendo las p
def routeFitness(self):
    if self.fitness == 0:
        self.fitness = 1 / float(self.routeDistance())
    return self.fitness
```

Figura 4. Función Fitness

Ventanas de tiempo: Con el fin de cumplir estas restricciones del problema, se establecieron penalidades para cada ventana de tiempo incumplida, estas penalidades suman 500 cada vez que se incumplen a la distancia total del recorrido. Este proceso lo que hace es bajar la aptitud o fitness de la ruta en mención con el fin de que no se escogida para las siguientes generaciones

```python
if(ordervini[i][0] > pathDistance or  pathDistance > ordervini[i][1]):
    penality = 500
else:
    penality = 0
```

Siguiente generación: Para las siguientes generaciones se realiza la ***selección por rango*** ya que a cada individuo se le asigna un ranking de acuerdo son su valor fitness, estos datos son almacenados en una tabla ordenándolos de mayor al menor, luego de esto se seleccionan los n primeros registros con mayor valor fitness para que sean los padres de la siguiente generación, donde n es el parámetro 'EliteSize' definido al inicio de la función, para este caso se tomó el valor de 10.

```python
def selection(popRanked, eliteSize):
    selectionResults = []
    df = pd.DataFrame(np.array(popRanked), columns=["Index","Fitness"])
    df['cum_sum'] = df.Fitness.cumsum()
    df['cum_perc'] = 100*df.cum_sum/df.Fitness.sum()

    for i in range(0, eliteSize):
        selectionResults.append(popRanked[i][0])
    for i in range(0, len(popRanked) - eliteSize):
        pick = 100*random.random()
        for i in range(0, len(popRanked)):
            if pick <= df.iat[i,3]:
                selectionResults.append(popRanked[i][0])
                break
    return selectionResults
```

Figura 5. Selección

Crossover: Con la función representada en la *figura 6 y figura 7*, se realiza el cruce del material genético de los padres para así proceder con la creación de los hijos.

```python
#Funcion para hacer crossover sobre el pool
def breedPopulation(matingpool, eliteSize):
    children = []
    length = len(matingpool) - eliteSize
    pool = random.sample(matingpool, len(matingpool))

    for i in range(0,eliteSize):
        children.append(matingpool[i])

    for i in range(0, length):
        child = breed(pool[i], pool[len(matingpool)-i-1])
        children.append(child)
    return children
```

Figura 6. Crossover

```python
#Funcion crossover para que dos padres creen un hijo
def breed(parent1, parent2):
    child = []
    childP1 = []
    childP2 = []

    geneA = int(random.random() * len(parent1))
    geneB = int(random.random() * len(parent1))

    startGene = min(geneA, geneB)
    endGene = max(geneA, geneB)

    for i in range(startGene, endGene):
        childP1.append(parent1[i])


    childP2 = [item for item in parent2 if item not in childP1]
    child = childP1 + childP2
    #print(child)
    return child
```

Figura 7. Creación de hijo

Mutación: Una vez los hijos han sido creados se procede a realizar su mutación de acuerdo con el valor 'mutationRate' definido al inicio de la función con '0.01', este proceso consiste en intercambiar elementos del arreglo (ciudades) de forma aleatoria de acuerdo con el rango de mutación definido.

```python
#Funcion que muta una ruta
def mutate(individual, mutationRate):
    for swapped in range(len(individual)):
        if(random.random() < mutationRate):
            swapWith = int(random.random() * len(individual))

            city1 = individual[swapped]
            city2 = individual[swapWith]

            individual[swapped] = city2
            individual[swapWith] = city1
    return individual
```

Figura 8. Mutación

Generaciones: En la *figura 9*, se muestra la función principal la cual es la encarga de orquestar cada uno de los anteriores pasos mencionados

```python
#Funcion para crear el algoritmo genetico
def geneticAlgorithm(population, popSize, eliteSize, mutationRate, generations):
    global progress
    pop = initialPopulation(popSize, population)
    progress = [1 / rankRoutes(pop)[0][1]]
    print("Distancia Inicial: " + str(progress[0]))

    for i in range(1, generations+1):
        pop = nextGeneration(pop, eliteSize, mutationRate)
        progress.append(1 / rankRoutes(pop)[0][1])
        #if i%50==0:
        print('Generacion '+str(i),"Distancia: ",progress[i])

    bestRouteIndex = rankRoutes(pop)[0][0]
    bestRoute = pop[bestRouteIndex]

    return bestRoute
```

Figura 9. Función algoritmo genético

Como se puede observar en la anterior imagen cada uno de los procesos es ejecutado en orden. Por otra parte, se observa que el número generaciones a crear esta definida por el parámetro 'generations' la cual para este caso se definió en **50**, esto debido a que usualmente después de este número la curva de la gráfica 'fitness vs generación' se aplanaba y con esto reducir el tiempo de ejecución de la aplicación, ver *figura 10*.

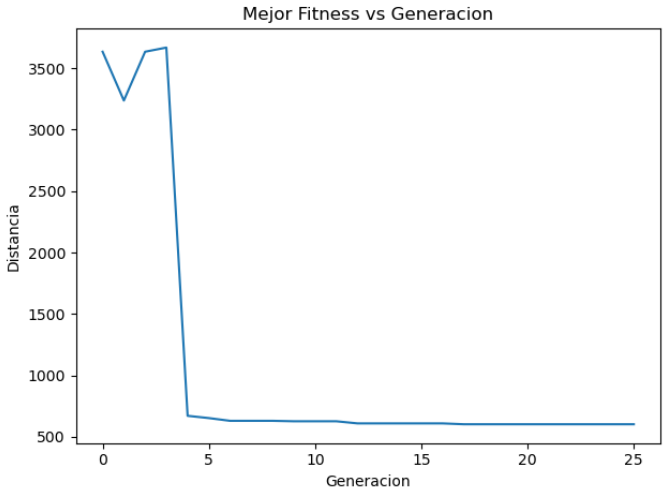

Figura 10. Fitness vs Generación

Resultado: En la *figura 11*, se observa el costo o distancia total recorrida por cada generación para resolver el problema.

```
Por favor ingrese el nombre de uno de los casos de prueba contenidos en 'Benchmark Instances TSPTW': rc_201.1.txt
Por favor ingrese el tiempo máximo de ejecución en segundos: 10
Distancia Inicial: 1072.0643
Generacion 1 Distancia:  1072.0643
Generacion 2 Distancia:  976.3304999999999
Generacion 3 Distancia:  921.4852999999998
Generacion 4 Distancia:  921.4852999999998
Generacion 5 Distancia:  886.5856999999999
Generacion 6 Distancia:  885.3609
Generacion 7 Distancia:  832.2511999999999
Generacion 8 Distancia:  619.7968999999998
Generacion 9 Distancia:  615.4467999999999
Generacion 10 Distancia:  594.9789
Generacion 11 Distancia:  594.9789
Generacion 12 Distancia:  594.9789
Generacion 13 Distancia:  594.9789
Generacion 14 Distancia:  594.9789
Generacion 15 Distancia:  594.9789
Generacion 16 Distancia:  594.9789
Generacion 17 Distancia:  594.9789
Generacion 18 Distancia:  594.9789
Generacion 19 Distancia:  594.9789
Generacion 20 Distancia:  589.0423999999999
Generacion 21 Distancia:  594.9789
Generacion 22 Distancia:  594.9789
Generacion 23 Distancia:  594.9789
Generacion 24 Distancia:  594.9789
Generacion 25 Distancia:  594.9789
Mejor Ruta:
[(2), (11), (18), (19), (12), (15), (0), (14), (13), (9), (16), (7), (5), (4), (8), (17), (6), (1), (10), (3)]
```

Figura 11. Resultado por generación

De igual manera se ofrece un modo 'debug' con el cual se evidencia el recorrido de la ruta seleccionada

```
Detalle:
| # | Desde | Hasta | Dist | Acum | Twi | Twf | Penality
| 0 | (10) | (0) | 48.0526 | 48.0526 | 0 | 960 | penality: 0
| 1 | (0) | (14) | 11.1803 | 59.2329 | 11 | 131 | penality: 0
| 2 | (14) | (12) | 25.8114 | 85.0443 | 39 | 159 | penality: 0
| 3 | (12) | (9) | 65.0 | 150.0443 | 80 | 200 | penality: 0
| 4 | (9) | (6) | 21.1803 | 171.2246 | 105 | 225 | penality: 0
| 5 | (6) | (5) | 15.3852 | 186.6098 | 146 | 266 | penality: 0
| 6 | (5) | (4) | 15.831 | 202.4408 | 149 | 269 | penality: 0
| 7 | (4) | (7) | 18.6023 | 221.0431 | 165 | 285 | penality: 0
| 8 | (7) | (17) | 43.2866 | 264.3297 | 194 | 314 | penality: 0
| 9 | (17) | (13) | 19.434 | 283.7637 | 246 | 366 | penality: 0
| 10 | (13) | (16) | 32.0227 | 315.7864 | 268 | 388 | penality: 0
| 11 | (16) | (8) | 35.0 | 350.7864 | 268 | 388 | penality: 0
| 12 | (8) | (15) | 57.0425 | 407.8289 | 326 | 446 | penality: 0
| 13 | (15) | (18) | 17.2111 | 425.04 | 335 | 455 | penality: 0
| 14 | (18) | (3) | 26.2788 | 451.3188 | 344 | 464 | penality: 0
| 15 | (3) | (19) | 34.3516 | 485.6704 | 375 | 495 | penality: 0
| 16 | (19) | (2) | 32.4722 | 518.1426 | 397 | 517 | penality: 50
| 17 | (2) | (11) | 23.9284 | 542.071 | 440 | 560 | penality: 0
| 18 | (11) | (1) | 28.6011 | 570.6721 | 537 | 657 | penality: 0
| 19 | (1) | (10) | 17.2801 | 587.9522 | 566 | 686 | penality: 0
```

Figura 12. Debug

V. Conclusiones

Ha y una clara ventaja de los algoritmos genéticos en comparación con las técnicas 'ingenuas' las cuales tienen tiempo de ejecución es exponencial y aumentan respecto al número de elementos a calcular. Por otra parte, es interesante observar como con este algoritmo opera en paralelo varias soluciones, en caso una solución no sea optima simplemente la descarta y continua con las demás soluciones, mientras que en los algoritmos tradicionales, si una solución no resulta optima se debe desechar todo o parte del proceso para iniciar de nuevo

También es importante ver como este tipo de métodos heurísticos y metaheurísticos ofrecen resultados muy aproximados, con bajos costes de procesamiento y de tiempo de ejecución. La aplicación de estas metodologías en algunos escenarios puede ser suficiente un resultado 'aproximado' para la toma de decisiones y/o acciones derivadas.

VI. Bibliografía

[1] R. F. S. J. G. Dantzig, «Solution of a large-scale traveling-salesman problem,» *Journal of the operations research society of America,* 1954.

[2] J. J. M. Casanova, «TRAVELING SALESMAN PROBLEM (TSP) Diseño de Algoritmos Heurísticos y Metaheurísticos eficientes para resolver el Problema del Agente Viajero,» *Universidad Nacional Autonoma de Nicaragua,* 2017.

[3] J. I. P. Rave, «TETRAHEURÍSTICA SISTÉMICA (THS) PARA EL TSP,» *Revista chilena de ingeniería, vol. 18 N,* 209.

[4] S. T. Muriel y D. Utrera Jaén, «Problema del viajante con ventanas de tiempo».

[5] J. Arranz de la Peña, «ALGORITMOS GENÉTICOS,» *Universidad Carlos III.*

[6] J. D. Marius M. Solomon, «Time Window Constrained Routing and Scheduling Problems,» *Transportation Science,* 1988.

[7] M. G. Pose, «Introducción a los Algoritmos Genéticos,» *Universidade da Coru˜na,* 2010.

[8] R. A. HINCAPIÉ, C. A. RÍOS PORRAS y R. GALLEGO, «TÉCNICAS HEURÍSTICAS APLICADAS AL PROBLEMA DEL CARTERO VIAJANTE (TSP),» *Scientia Et Technica, vol. X,* 2004.