Documentación Metaheurísticas

Carlos Requena Doña(79042656M) y Yessin Mohamed(45318800Z)

Identificación del guión: Práctica 2

Identificación del grupo de los alumnos: Grupo 11

Algoritmos implementados en la práctica: Algoritmo generacional diferencial, Algoritmo generacional evolutivo

Universidad de Jaén

Contents

1 Problema a tratar, algoritmos y consideraciones

1.1 Algoritmos y definición del problema

El problema a tratar es del Algoritmo del viajero, en el que buscamos la mejor ruta. Para obtener la distancia entre puntos en el problema utilizamos la siguiente fórmula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Partiendo de esta fórmula y los datos, obtenemos la matriz de distancias a partir de las distintas ciudades.

Este proyecto se basa en el uso del algoritmo genético, el cuál se basa en una población que evoluciona simulando el sistema genético. Tenemos el algoritmo genético y el evolutivo diferencial.

1.2 Estructura general del código

En cuanto al código se refiere, la estructura general se basa en el main, de modo que este es el que inicializa la clase "FileLoader" para cargar las coordenadas de las ciudades e inicializa la instancia de ambos algoritmos para proceder a ejecutarlos. Junto a esto, tenemos un archivo log generado tras la ejecución del programa que permite ver como se comportan los algoritmos durante su ejecución facilitando el estudio de los mismos. Tenemos una clase llamada "FileLoader" encargada de inicializar una serie de datos a través de un archivos de texto, dicha clase está implementada mediante un Singleton permitiendo así el acceso a dichos datos desde cualquier parte del código. FileLoader permite evitar no solo el uso de "números mágicos", sino que proporciona una flexibilidad absoluta donde el usuario puede simplemente modificar los parámetros de ejecución sin tocar una sola línea de código.

1.3 Representación de soluciones

Dado que estamos tratando con un algoritmo genético, tenemos una población de individuos, donde el élite representa al mejor individuo de la solución actual en el problema. Dicho individuo tiene su vector que representa la ruta de ciudades.

1.4 Restricciones y consideraciones

Debido a que la generación de poblaciones presentan un factor aleatorio para evitar que el programa se quede colgado intentando generar el número aleatorio correcto se establece un límite de intentos. Por ejemplo, si se está generando un individuo de manera aleatoria y solo faltan 4 ciudades sin asignar, se establece un número de intentos 'x' para obtener alguna de esas ciudades, si tras ese número de intentos no se logra obtener una ciudad válida, se procede a buscar una ciudad sin asignar de manera manual.

1.5 Función objetivo y Coste

El coste de la evaluación, o mejor dicho en este contexto, "fitness", se calcula con la siguiente fórmula:

$$\mathrm{C(S)} = \sum_{i=0}^{n-2} \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$$

Como se ve, se itera calculando las distancias representadas por los distintos desplazamientos representados en el vector solución del individuo.

1.6 Clases utilizadas para el algoritmo genético

Representamos la población del algoritmo mediante una clase llamada "Población" que contiene a los individuos que forman parte de la misma. La Población representa una relación por composición de los individuos de esta, siendo que los crea y destruye en última instancia (realmente se van reemplazando por generaciones, siendo que el destructor si elimina memoria de los individuos al finalizar la aplicación).

La población se compone de individuos representados por la clase "Individuo". Dicha clase almacena el vector mencionado anteriormente al hablar de la representación de soluciones, además de distintos atributos tales como evaluado (variable booleana que indica si el individuo ha sido evaluado) y costeAsociado (coste o fitness del individuo).

2 Algoritmo Genético

2.1 Pseudocódigo

2.1.1 Método principal

En primer lugar vamos a ver el código que actúa de eje principal del algoritmo. Los atributos más importantes del método executeEvolutivo() son:

- numEvaluaciones: variable que contiene el número de evaluaciones dadas en la ejecución del algoritmo.
- individuosSeleccionados: vector de individuos que contiene aquellos individuos seleccionados mediante torneo.
- elites: vector de individuos calificados como élite, es decir, aquellos mejores individuos de la población actual.

Algorithm 1: executeEvolutivo()

```
1 while numEvaluaciones < loader->getNumEvaluaciones() and elapsed_seconds.count() <
    60 do
2   while individuosSeleccionados.size() < loader->getTamPoblacion() do
3     individuosSeleccionados.push_back(poblacion->getIndividuos()[torneo()])
4   end
5   cruce(individuosSeleccionados)
6   mutacion(individuosSeleccionados)
7   EN CASO DE NO MANTENERSE ÉLITES HACEMOS TORNEO
      PERDEDORES
8   poblacion->setIndividuos(individuosSeleccionados)
9 end
```

Este método se ejecuta mientras se respeten los límites de ejecución que son límite de evaluaciones y de tiempo. Se van introduciendo individuos en el vector de individuos seleccionados conforme se va realizando el torneo hasta llenar el límite de la población.

La siguiente fase es el cruce y la mutación, usando el operador de cruce OX2 y MOC para el cruce y mutando mediante permutación en el vector del individuo, en caso de que algún individuo sea modificado será reflejado estableciendo el atributo 'evaluado' de dicho individuo a falso.

Hay que tener en cuenta la existencia de los individuos "Élite". Estos individuos son aquellos que se preservan para la siguiente generación sin ningún tipo de cambio, siendo aquellos que mejor "fitness" aportan. Para comprobar que los individuos Élite se mantengan en la nueva población tenemos un método llamado "comprobarElitismo" que comprueba que el número de Élites que deben estar presentes en la población se corresponde con los realmente existen en la población. En caso de no corresponderse, se devolverá un booleano falso y aquellos élites que no se han mantenido en la población. Por ejemplo, si estamos trabajando con dos élites y en la nueva población solo se mantiene uno se devolverá un bool a falso y el élite que falta. Por tanto, en caso de no mantenerse los élites en la población, se debe asegurar su preservación, para ello, hacemos un torneo de perdedores para echar aquel que tenga el peor "fitness", siendo sustituido por un individuo élite.

Hemos hablado de torneo repetidamente, pero ahora vamos a explicar su funcionamiento. El torneo consiste en elegir un número de individuos en base al parámetro "k", de modo que compitan en un torneo en el que la victoria se asegura a aquel que tenga mayor "fitness". En caso de ser un torneo de perdedores, la selección de los participantes del torneo se hace de la misma forma, pero ganando aquel con peor "fitness".

Una vez tenemos todos los individuos listos, llamamos al método "setIndividuos" que cambia la población con los individuos de ambos vectores previamente mencionados. Dicho método no solo cambia los individuos, sino que calcula el élite, además de calcular el coste nuevamente para aquellos individuos modificados(los que tienen el atributo 'evaluado' a falso).

3 Algoritmo Evolutivo Diferencial

3.1 Pseudocódigo

3.1.1 Método principal

En primer lugar vamos a ver el código que actúa de eje principal del algoritmo. Los atributos más importantes del método executeDiferencial() son:

- padre, aleatorio1, aleatorio2, objetivo: individuos necesarios para la combinación ternaria.
- individuoGenerado: variable que almacena el individuo obtenido tras la recombinación ternario.
- poblacion: variable que representa la población a someterse a reemplazamiento que almacena a los individuos.
- secuenciaPadre: variable que indica que padre se toma, se actualiza de forma secuencial.

```
1  while individuosSeleccionados.size() < loader->getTamPoblacion() do
2  |  padre ← new Individuo(*poblacion->getIndividuos()[secuenciaPadre]);
   |   seleccionIndividuos(aleatorio1, aleatorio2, objetivo);
3  |  IndividuoGenerado ← recombinacionTernaria(padre, aleatorio1, aleatorio2, objetivo);
   |     individuosSeleccionados.push_back(IndividuoGenerado); // Almaceno el nuevo
   |     individuo generado
4  |  secuenciaPadre++; // Actualizo el índice que indica el padre a tomar
5  end
6  // Reemplazamiento
7  if individuosSeleccionados[i]→getCosteAsociado() <
     poblacion→getIndividuos()[i]→getCosteAsociado() then
8  |  poblacion→getIndividuos()[i] ← individuosSeleccionados[i]; // Sustituyo al padre
   |     con el hijo
9  end
10
```

Este método, al igual que aquel en el generacional, tenemos una ejecución bajo el margen de un límite tanto de evaluaciones de individuos como de tiempo de ejecución transcurrido.

En primer lugar, necesitamos generar el padre de forma secuencial, obteniéndolo de la población. Tras ello tomamos los individuos aleatorios mediante el método "seleccionIndividuos", en el que los aleatorios son generados usando EDA o EDB. Finalmente, se genera por torneo el individuo objetivo

Procedemos a continuación con la recombinación ternaria. Con el operador se cruzan los distintos individuos anteriormente seleccionados(padre y aleatorios) dando como resultado un individuo nuevo. Este nuevo individuo sufre un cruce con el individuo objetivo dando lugar finalmente a un hijo.

3.1.2 Método de recombinación

En primer lugar vamos a ver el código que actúa de eje principal del algoritmo. Los atributos más importantes del método executeTSP() son:

- posicionSwap: variable que almacena representa un número aleatorio con rango de población para el método.
- elemento1, elemento2: variables del swap a ocurrir en la recombinación.
- centinela, indiceAleatorio1, indiceAleatorio2, elementoAleatorio2: índices utilizados a la hora de llevar a cabo la recombinación ternaria.

```
1  while individuosSeleccionados.size() < loader->getTamPoblacion() do
2  |  padre ← new Individuo (*aleatorio1); elemento1 ← posicionSwap; elemento2 ←
   |   posicionSwap + 1; (padre →getVIndividuo()[posicionSwap], padre
   |   →getVIndividuo()[posicionSwap + 1]);
3  |  while centinela ≤ 1 and i < aleatorio1→getVIndividuo().size() do
4  |  |  if aleatorio1→getVIndividuo()[i] == padre →getVIndividuo()[elemento1 ] then
5  |  |  |  indiceAleatorio1 ← i; centinela ++;
6  |  |  end
7  |  |  if aleatorio2→getVIndividuo()[i] == padre →getVIndividuo()[elemento2 ] then
8  |  |  |  indiceAleatorio2 ← i; elementoAleatorio2 ← aleatorio2→getVIndividuo()[i];
   |  |  |   centinela ++;
9  |  |  end
10 |  |  i++;
11 |  end
12 |  swap (new Individuo →getVIndividuo()[indiceAleatorio1 ], new Individuo
   |   →getVIndividuo()[indiceAleatorio1 ]); (new Individuo
   |   →getVIndividuo()[indiceAleatorio2 ], new Individuo →getVIndividuo()[indiceAleatorio2
   |   ]);
13 |  hijo1 ← new Individuo (*new Individuo); cruceOX2 (new Individuo, , hijo1);
14 |  delete new Individuo;
15 |  return hijo1;
16 end
```

Este método se encarga como se ha dicho antes de realizar unos cruces y cambios entre individuos, de modo que se simula el proceso de recombinación planteado en clase.

4 Experimentos y análisis de resultados

4.1 Semillas utilizadas

Las semillas utilizadas en las distintas ejecuciones son:

Table 1: Semillas utilizadas

Ejecución 1	Ejecución 2	Ejecución 3	Ejecución 4	Ejecución 5
79042656	90426567	04265679	42656790	26567904

Los resultados de cada semilla se encuentran en el siguiente enlace: Enlace Logs

4.2 Algoritmo Genético

4.2.1 Parámetros

Los valores establecidos en los parámetros para el algoritmo genético son los siguientes:

- tamPoblacion
- porcentajeInicializacion
- numEvaluaciones
- kBest
- numDeElites
- tamGreedyAleatorio
- tipoCruce
- probabilidadCruce
- kWorst
- tiempoParada

4.3 Algoritmo Evolutivo Diferencial

4.3.1 Parámetros

Los valores establecidos en los parámetros para el algoritmo evolutivo diferencial con selección "EDB" o "EDA" son los siguientes:

- tamPoblacion: 50
- porcentajeInicializacion: 80
- numEvaluaciones: 50000

- kBest: 2
- numDeElites: 1
- tamGreedyAleatorio: 5
- tipoCruce: MOC
- probabilidadCruce: 70
- kWorst: 3
- tiempoParada: 60
- tipoSeleccion: EDB/EDA

4.4 Tablas de resultados

Las tablas de resultados son muchas, por lo que se guardan en un enlace a un documento Excel donde se puede ver cada una de las ejecuciones con las diferentes configuraciones. Las configuraciones son las siguientes:

- 1. GenOX2: M=50, E=1, kBest=2, kWorst=3.
- 2. GenOX2: M=50, E=1, kBest=3, kWorst=3.
- 3. GenOX2: M=50, E=2, kBest=2, kWorst=3.
- 4. GenOX2: M=50, E=2, kBest=3, kWorst=3.
- 5. GenOX2: M=100, E=1, kBest=2, kWorst=3.
- 6. GenOX2: M=100, E=1, kBest=3, kWorst=3.
- 7. GenOX2: M=100, E=2, kBest=2, kWorst=3.
- 8. GenOX2: M=100, E=2, kBest=3, kWorst=3.
- EDA (Diferencial)
- EDB (Diferencial)

El enlace al documento es el siguiente: Enlace Tablas

4.5 Análisis de resultados

4.5.1 Mejor OX2

En cuanto a la mejor configuración para el OX2 en líneas generales es la configuración 6 en cuanto a relación media/desviación típica. En cuanto a la influencia de una configuración u otra no se observa ninguna relación a destacar. Para poder hacer un análisis más extenso en el que podamos afirmar como afecta una configuración u otra, habría que tener en cuenta más muestras.

4.5.2 Mejor MOC

En cuanto a la mejor configuración para el MOC en líneas generales es la configuración 5 en cuanto a relación media/desviación típica, aunque si solo se tiene en cuenta la desviación típica la configuración 6 da mejores resultados. Por otro lado, podemos observar que en líneas generales la configuración de un solo élite y 2 kbest favorece al algoritmo MOC, ya que tanto para una población de tamaño 50 como para una de tamaño 100 son las mejores o prácticamente. Además, tener una población de mayor tamaño también favorece a los resultados

4.5.3 MOC vs OX2

Comparando el mejor del MOC con el mejor del OX2 se observa que el MOC da mejores resultados que el OX2, sobretodo en cuanto a la media del 'ch130'. Sin embargo, si comparamos como se comportan en general ambos algoritmos para las diferentes configuraciones, observamos que en general se comportan igual y ambos rondan el baremo alrededor del 50-70 de media para el 'ch130' y rondando el 50-60 de media para el 'pr144'.

4.5.4 EDA vs EDB

Se observa que el EDA es mejor que el EDB tanto en media como en desviación típica.

4.5.5 MOC vs EDA

El MOC es mucho mejor que el EDA teniendo el MOC una media de 49.46% y 53.52% y el EDA una media 122.76% y 63.07%. En cuanto a la desviación, se observa también que el MOC es mejor que EDA, teniendo 11.88%-4.76% y 26.96%-3.94%. De modo

4.6 Gráficos

En primer lugar tenemos una gráfica que representa la ejecución del OX2 Generacional.

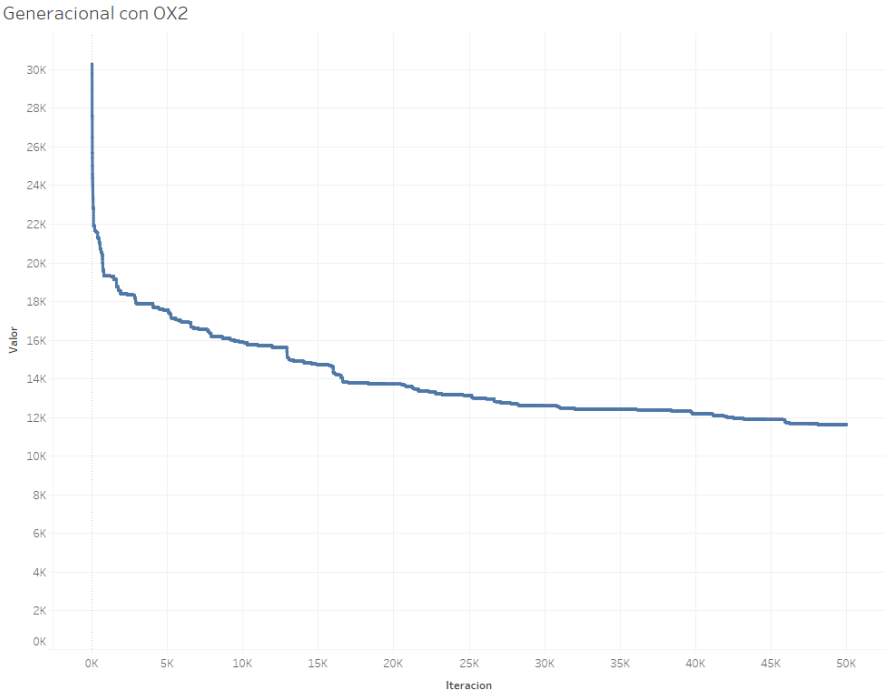

Figure 1: OX2

Esta siguiente gráfica representa el generacional con MOC

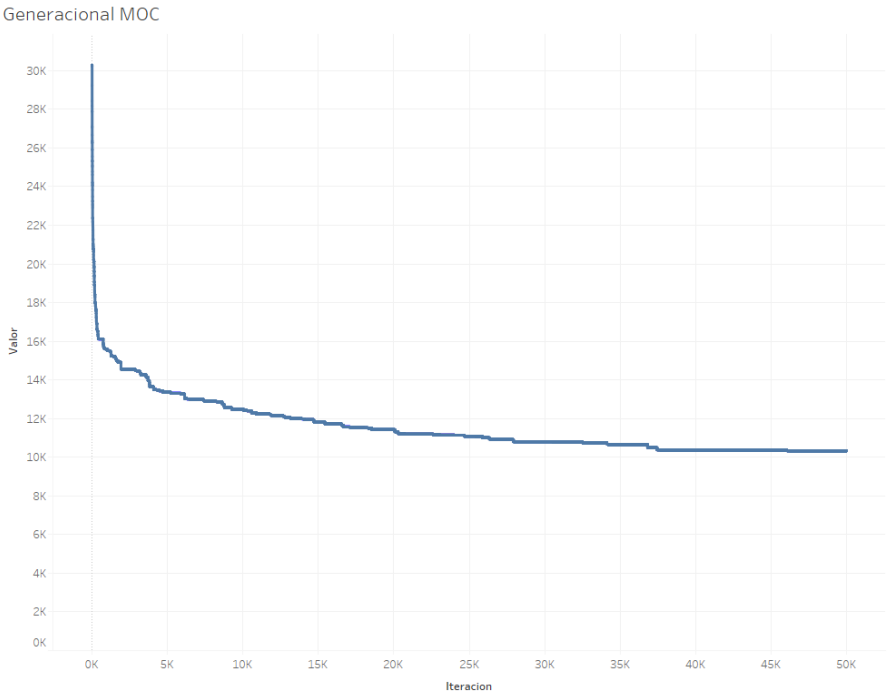

Figure 2: MOC

4.6.1 Análisis gráficas

Se observa que ambos algoritmos acaban obteniendo costes similares, la única diferencia sustancial entre ellas es cómo se comportan durante la ejecución. Tenemos que el MOC mejora los valores muy rápido en un inicio y que después se estabiliza y mejora el coste es saltos más pequeños, por otro lado, el OX2 mejora los valores de manera similar a lo largo de las iteraciones del algoritmo, sin estancarse mucho en ningún punto.

Debido a estos comportamientos podemos deducir que el MOC se comportará mejor en aquellos algoritmos en los que se ejecuten pocas iteraciones y que el OX2 se comportará mejor en aquellos que ejecuten muchas iteraciones. Sin embargo, esta es una deducción sesgada a esta gráfica, para otras configuraciones y semillas se puede comportar de otra forma, por lo que no deja de ser una suposición.