ALGORÍTMICA – II
Grado en Ingeniería Informática en Sistemas de Información
Universidad Pablo de Olavide
EPD03: Algoritmos Evolutivos I

Objetivos

1. Aprender diferentes codificaciones para resolver problemas con algoritmos evolutivos.
2. Codificar correctamente los diferentes operadores genéticos.
3. Resolución de problemas reales.

Introducción

En esta práctica se van a aplicar los conceptos aprendidos en las clases de EB (Tema 4) relativos a los algoritmos genéticos. El esquema general de un algoritmo genético es el siguiente:

```
Inicio (1)
   t = 0;
   inicializar P(t);
   evaluar P(t);
   Mientras (no se cumpla la condición de parada) hacer
   Inicio(2)
        t = t + 1
        seleccionar P' desde P(t-1)
        recombinar P'
        mutar P'
        reemplazar P(t) a partir de P(t-1) y P'
        evaluar P(t)
   Final(2)
Final(1)
```

Antes de desarrollar la práctica es muy recomendable que el estudiante dedique unos minutos a recordar los conceptos básicos que se vieron en teoría. En concreto, es fundamental recordar:

1. La representación de un problema.
2. Inicialización de una población.
3. Correspondencia entre genotipo y fenotipo.
4. Evaluación de un individuo (fitness).
5. Operador de mutación.
6. Operador de cruce.
7. Selección de individuos.
8. Reemplazo de individuos.
9. Condición de parada del algoritmo.

Ejercicios

Ejercicio 1 (100 minutos). Se trata de implementar una solución al problema del viajante de comercio *(TSP)* usando un algoritmo evolutivo. Un viajante de comercio quiere visitar *n* ciudades empezando y terminando en la misma ciudad, por lo que debe determinar la ruta a seguir para que la distancia total recorrida sea mínima.

Para resolver este problema se le facilita un proyecto con parte del código ya implementado que puede descargar del Campus Virtual, debiendo desarrollar los métodos especificados en este ejercicio que son los necesarios para implementar el algoritmo evolutivo que resuelve el problema. Junto al proyecto se facilitan tres ficheros de texto con ejemplos de ciudades y sus ubicaciones para probar el funcionamiento de las soluciones que se implementen, una con 100 ciudades, otra con 500 y la última con 1000.

El proyecto está compuesto por tres clases:

- **TSP**: clase principal necesaria para ejecutar el programa donde habrá que configurar una serie de parámetros del algoritmo evolutivo, entre ellos, el nombre del fichero de ejemplo que se quiera usar. Además tiene un atributo llamado **`matrix`** que es una tabla bidimensional donde se almacena la distancia entre dos ciudades cualesquiera, por ejemplo, si quisiéramos averiguar la distancia que separa la ciudad 23 y la 59, habría que consultar el valor de **`matrix[23][59]`**.
- **Individual**: clase que representa a cada individuo de la población del algoritmo evolutivo. En este caso cada individuo será un vector de enteros que representará el orden en el que el viajante debe recorrer las distintas ciudades.
- **Population**: clase que representa la población de individuos del algoritmo evolutivo, en nuestro caso un vector de objetos de la clase *Individual*. En esta clase se implementan todos los métodos necesarios para el proceso de evolución de unas generaciones a otras.

Para resolver el ejercicio deberá implementar los siguientes métodos:

- **Clase Individual**:
  - **calculateCost**: que deberá calcular la distancia total recorrida al hacer la ruta marcada por la secuencia de ciudades representada por dicho objeto de la clase Individual. Por definición del TSP, a la distancia total acumulada de recorrer todas las ciudades hay que sumar también la distancia que separa la última ciudad y la primera, ya que se supone que el viajante debe volver a la ciudad de origen.
  - **generateRandomTour**: que debe generar una secuencia de ciudades de forma aleatoria y calcular la distancia total al recorrerlas en ese orden, es decir, generará un nuevo individuo. Este método también será el responsable de almacenar ese nuevo individuo en el atributo **`tour`** de la clase. La secuencia generada para el nuevo individuo tendrá un número de ciudades determinado por el parámetro de entrada del método. Hay que recordar que, por definición del propio problema TSP, no se puede pasar dos veces por la misma ciudad.
- **Clase Population**:
  - **initializePopulationRandomly**: que debe inicializar aleatoriamente la población de individuos cargándolos en el atributo **`individuals`**.
  - **getBestIndividual**: recibe como parámetro un vector de individuos y devuelve el individuo cuyo coste (distancia total recorrida) sea menor.
  - **crossover**: operador de cruce del algoritmo genético. A partir de los dos padres pasados como parámetros se genera un nuevo individuo hijo que será el resultante de cruzar ambos padres. Habrá que tener en cuenta que en la clase TSP existe un parámetro configurable llamado **`CLONE_RATE`** que implicará que, según ese porcentaje de clonación, el individuo hijo que se genera no será un cruce de los padres sino un clon del padre cuyo coste sea menor.
  - **mutate**: operador de mutación del algoritmo genético. Hay que tener en cuenta que el nuevo individuo mutado deberá tener una permutación entre dos genes al azar del individuo original, ya que por definición del TSP el viajante no debe visitar dos veces una misma ciudad.
  - **evolve**: genera una nueva población a partir de la actual. La operatividad del método se divide en cuatro grandes bloques:

- Ordenación: ordena ascendentemente la población actual según su coste y la guarda en el atributo **`sorted`**.
- Elitismo puro: el mejor individuo de la población actual pasa a la población siguiente.
- Elitismo por porcentaje: la clase TSP tiene un parámetro configurable llamado **`ELITE_PERCENT`** que determina el porcentaje de individuos de la población anterior con menor coste que pasan a la generación siguiente aplicándole el operador de mutación según el parámetro **`MUTATION_RATE`**.
- Reemplazo: el resto de la nueva población se genera cruzando dos padres seleccionados por torneo usando el método **`selectParentViaTournament`**. Al hijo resultante, que formará parte de la siguiente generación, se le aplicará el operador de mutación en función del parámetro configurable **`MUTATION_RATE`**. Habrá que controlar que todos los nuevos individuos que se generen deben de tener calculado su coste.

Problemas

Problema 1 (2 horas). Diseñar e implementar una solución al siguiente problema usando un algoritmo genético. Se dispone de una balanza con dos platillos y de *n* objetos, cada uno de los cuales tiene un peso positivo. El objetivo es encontrar un reparto de los objetos entre los dos platillos de la balanza de forma que la diferencia entre los pesos de los objetos situados en cada platillo sea mínima. Se sugiere usar una representación binaria para determinar el platillo de la balanza al que va cada objeto, es decir, un vector de **`n`** elementos donde $\mathbf{x_i}$ valdrá cero si el objeto **`i`** es colocado en el platillo 1, o bien uno, si es colocado en el platillo 2. Si se usa esta representación es mucho más sencillo implementar los operadores genéticos de cruce y mutación.

Problema 2 (3 horas). Diseñar e implementar una solución al siguiente problema usando un algoritmo genético. Dado un grafo no dirigido se ha de encontrar el coloreado mínimo de sus vértices, es decir, se ha de utilizar el menor número de colores cumpliéndose que dos vértices adyacentes no pueden tener el mismo color. Se sugiere usar una representación donde cada individuo sea un vector que contenga una permutación de los vértices, el grafo se coloreará recorriendo en orden al individuo y asignándole el primer color legal que puede tener cumpliendo las restricciones.