ALGORÍTMICA – II
Grado en Ingeniería Informática en Sistemas de Información
Universidad Pablo de Olavide
EPD04: Algoritmos Evolutivos II

Objetivos

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

Ejercicios

Ejercicio 1 (120 minutos). Se trata de implementar una solución al conocido problema de la mochila *(Knapsack Problem)* usando un algoritmo genético. El problema responde a la siguiente situación: imagínese que vamos a hacer una excursión a la que podemos llevar una mochila que tiene una capacidad limitada. Cada objeto que introducimos en ella tiene un peso y en contrapartida durante el viaje nos proporcionará un beneficio o utilidad (ejemplo: una cantimplora). El problema surge cuando debemos seleccionar los objetos que se meterán en la mochila de forma que nuestro beneficio sea máximo sin exceder su capacidad.

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. Dentro del proyecto se facilita una carpeta llamada **`Datasets`**, donde hay 8 ejemplos para probar la solución que se implemente. Dichos ejemplos se componen de tres ficheros, más uno adicional, cuyo nombre tiene un prefijo (**`p01`**, **`p02`**, ….) y un sufijo que determina su contenido:

- "**`_c`**": capacidad total de la mochila. Ejemplo: **`p01_c.txt`**
- "**`_p`**": valor de cada objeto que podemos meter en la mochila. Ejemplo: **`p01_p.txt`**
- "**`_w`**": peso de cada objeto que podemos meter en la mochila. Ejemplo: **`p01_w.txt`**
- "**`_s`**": Fichero opcional. Representa la solución óptima conocida para ese ejemplo. El programa presentará los resultados del algoritmo genético comparados con la solución óptima. Si no se facilita este fichero la presentación de resultados no contemplará esa comparación. Ejemplo: **`p01_s.txt`**

El proyecto facilitado contiene tres clases:

- **Ejercicio1**: clase principal necesaria para ejecutar el programa donde no habrá que configurar nada.
- **KnapsackGenerator**: clase principal opcional que solo sirve para generar nuevos ejemplos de problemas de la mochila con el formato descrito anteriormente. Este generador no facilita la solución óptima por lo que solo devolverá tres ficheros, los terminados en **`c`**, **`p`** y **`w`**.
- **Knapsack**: clase que representa un solucionador de problemas de la mochila usando un algoritmo genético. Los atributos fundamentales para entender la representación escogida para el problema son los siguientes:
    - **`capacity`**: entero que representa capacidad total de la mochila.
    - **`numItems`**: entero que representa el total de objetos disponibles para ser seleccionados.
    - **`values`**: vector de enteros donde cada posición representa el valor que aporta un determinado objeto. Ejemplo: **`values[i]`** → valor del objeto *i*-ésimo.
    - **`sizes`**: vector de enteros donde cada posición representa el peso de un determinado objeto. Ejemplo: **`sizes[i]`** → peso del objeto *i*-ésimo.

- **`PopulationSize`**: entero que representa el número de individuos (posibles soluciones) que formarán parte de la población del algoritmo genético.
- **`numGenerations`**: entero que representa el número de generaciones que evolucionará el algoritmo genético para encontrar una solución.
- **`mutationRate`**: tasa de aplicación del operador genético de mutación.
- **`crossoverRate`**: tasa de aplicación del operador genético de cruce.
- **`population`**: matriz de booleanos donde cada fila representará una posible solución al problema, en la que cada posición indicará si un objeto forma parte de la selección que llenará la mochila. Ejemplo: **`population[i][j]=true`** → La solución *i*-ésima de la población incluye el objeto *j*-ésimo dentro de la mochila.
- **`vectorFitness`**: vector de números reales que almacenará la bondad de cada una de las soluciones de la población actual. Ejemplo: **`vectorFitness[i]`** → representará el fitness de la solución *i*-ésima de la población.
- **`vectorSizes`**: vector de enteros que almacenará el peso total de cada una de las soluciones de la población actual. Ejemplo: **`vectorSizes[i]`** → representará el peso total de la solución *i*-ésima de la población.

Para resolver el ejercicio deberá implementar los siguientes métodos de la clase **Knapsack**:

- **fitness**: hay que implementar parte del código de este método para que calcule el beneficio total de la solución que se pasa como parámetro junto al peso de la misma.
- **GenerateInitialPopulation**: que genera una población inicial de forma aleatoria.
- **rouletteSelection**: que devuelve un vector de enteros que contiene los índices de las soluciones que forman parte de la población actual que han sido seleccionadas mediante el método de la ruleta.
- **tournamentSelection**: que devuelve un vector de enteros que contiene los índices de las soluciones que forman parte de la población actual que han sido seleccionadas mediante el método del torneo.
- **nSliceCrossover**: que recibe un vector de enteros con los índices de las soluciones seleccionadas para ser padres, y devuelve una matriz booleana que contiene las dos soluciones hijas generadas al aplicar el operador genético de cruce en un punto.
- **uniformCrossover**: que recibe un vector de enteros con los índices de las soluciones seleccionadas para ser padres, y devuelve una matriz booleana que contiene las dos soluciones hijas generadas al aplicar el operador genético de cruce uniforme.
- **nPointMutation**: que recibe una solución como parámetro y devuelve otra resultante de mutar *n* genes seleccionados al azar. El valor de *n* vendrá determinado por el atributo **`numGensMutate`**.
- **invertMutation**: que recibe una solución como parámetro y devuelve otra resultante de mutar todos sus genes.
- **elitism**: hay que implementar parte del código de este método que realiza el elitismo del algoritmo genético. Selecciona las dos mejores soluciones de la población actual y las inserta directamente en dos posiciones al azar de la población de la generación siguiente.

Problemas

Problema 1 (2 horas). Diseñar e implementar una solución al problema de la asignación de tareas usando un algoritmo genético. Existen *n* tareas y *n* trabajadores. La realización de la tarea *i* por parte del trabajador *j* implica un coste **`c[i][j]`**. El objetivo es encontrar una asignación de tareas a trabajadores (una tarea a cada trabajador; cada trabajador sólo puede realizar una tarea) de forma que se minimice el coste total.

Problema 2 (3 horas). Diseñar e implementar una solución al Problema de Asignación Cuadrática (QAP) usando un algoritmo genético. Se desea realizar una asignación de *n* unidades a *n* localizaciones diferentes. Se conocen las distancias entre las localizaciones y el flujo que existe entre las diferentes unidades. Los datos con los que se va a trabajar estarán almacenados en dos matrices cuadradas (NxN), la primera es la matriz de flujo en la que cada posición `f[i][j]` contiene el flujo existente entre las unidades *i* y *j*. La segunda matriz contendrá en `d[i][j]` la distancia entre la localización *i* y la localización *j*. Un ejemplo muy ilustrativo de este problema sería el de imaginar un hospital con 4 localizaciones y 4 unidades. Tendría una gran repercusión la organización de estas unidades en las diferentes localizaciones, ya que no tendría sentido colocar dos unidades muy relacionadas entre sí en zonas alejadas del hospital. Buscamos obtener la organización óptima para este problema. ¿Cómo ubicaríamos en el hospital las diferentes unidades (maternidad, urgencias, cirugía, traumatología) para que las distancias recorridas por las personas que se desplazan entre las distintas unidades sea mínima?