Diseño de un algoritmo genético para el problema de máxima diversidad

Ricardo Ignacio Shepstone Aramburu

1. Problema de máxima diversidad (MDP)

consiste en determinar un subconjunto M con cardinalidad dada a partir de un conjunto N de elementos, de tal manera que la suma de las distancias entre los elementos de M, tomados en parejas, sea la máxima posible. La medida de distancia entre los elementos dependerá de la naturaleza del problema, para esta aplicación, estas distancias vienen dadas en los ficheros de texto como números reales o enteros.

Si $N = \{e_1, ..., e_n\}$ es un conjunto de elementos y $d(e_i, e_j)$ la distancia entre los elementos e_i y e_j (también expresado como d_{ij}), el problema de máxima diversidad consiste en encontrar un subconjunto $M \subset N$ de cardinalidad dada m con $m < n$, tal que se maximice la suma de las distancias entre los elementos de M. Este problema se puede formular como:

$$\text{Maximizar} \quad z = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij} \cdot x_i \cdot x_j,$$

$$\text{Sujeto a} \sum_{i=1}^{n} x_i = m, \quad x_i \in \{0, 1\}, \quad i = 1, ..., n$$

Donde x_i es una variable binaria que indica si el elemento e_i se ha seleccionado para el subconjunto M.

2. Metodología

Para la implementación del algoritmo genéticose ha utilizado el lenguaje de programación Python, se han ido probando las distintas funciones encontradas en el archivo "Funciones.py", a su vez, estas funciones se han probado junto a pequeños experimentos en el archivo "Pruebas.py". Tras probar el correcto funcionamiento de las funciones, se ha generado la clase **MDP_int** del archivo "GA.py" que recoge en una clase el algoritmo. Esta clase se ha probado en el mismo archivo de pruebas y se han realizado pequeños experimentos para determinar los mejores parámetros. Finalmente con los parámetros encontrados, se ha realizado la experimentación sobre todos los casos, registrando los resultados y el tiempo de ejecución para cada caso. El código de esto último se encuentra en el archivo "Experimentos.py".

3. Descripción del algoritmo

En este apartado se describirán las principales partes y funciones de la clase en la que se encuentra el algoritmo genético implementado. Esta clase utiliza un método constructor para establecer ciertos parámetros del algoritmo, como el tamaño de la población o el número de generaciones que se quieren simular.

3.1. Representación del problema

A la hora de construir un algoritmo genético es importante establecer la manera en la que las soluciones del problema se van a representar, ya que puede condicionar la manera en que los operadores genéticos actúan y la lógica del resto del algoritmo.

Para este problema se ideó utilizar una codificación binaria, donde cada solución estaría compuesta por una lista de valores binarios. En esta representación, las listas tendrían una longitud igual a la cantidad de elementos del conjunto N, y el valor 1 indicaría que el elemento que se encuentra en esa posición se ha seleccionado. Para esta representación habría que tener cuidado con la generación de soluciones no válidas a la hora de aplicar el operador de cruce, y además habría que tener en cuenta que cada vez que se mute cambiando un valor a 0 o 1, habría que mutar otro valor a 1 o 0 respectivamente. Esto se debe a que las soluciones tienen que tener un determinada cantidad de valores 1, puesto que las soluciones deben tener m elementos.

En su lugar se optó por utilizar una codificación de números enteros, en la que cada solución viene representada por una lista de números enteros de tamaño m. Cada número de la lista representa el índice del elemento escogido del conjunto N original. Los problemas que se pueden encontrar al aplicar los operadores genéticos en esta representación están relacionados con la repetición de elementos en una misma solución. En las secciones correspondientes se explicarán estas limitaciones en detalle, y se explicará la lógica propuesta para solventarlas.

En cuanto a los valores de las distancias entre los elementos, se han obtenido con el método *__read_distances*, a partir de los archivos de texto proporcionados. Las distancias entre los elementos se han almacenado en un diccionario de Python para agilizar la lectura y su acceso, en el que las claves del diccionario corresponden con los índices de los elementos de los que se quiere consultar la distancia.

3.2. Inicialización de la población

La población inicial se ha generado mediante el método *__generate_population*, que genera una lista del tamaño especificado mediante su parámetro correspondiente de soluciones. Esto se hace llamando al método encargado de generar una solución, que devuelve una lista de longitud m con valores aleatorios entre 0 y $n - 1$. Por lo que la generación de la población inicial es completamente aleatoria.

3.3. Evaluación

La evaluación de cada población se ha realizado mediante el método *__evaluate_population*, que lo único que hace es llamar a la función *__calculate_fitness* para obtener el valor de la función objetivo para cada solución de la población.

El cálculo de la función objetivo se ha implementado de forma que solamente se tenga en cuenta la distancia de un elemento e_i al elemento e_j (d_{ij}) y no la distancia de elemento e_j al elemento e_i (d_{ji}) puesto que estas distancias son equivalentes, así en vez de sumar las distancias en ambos sentidos y dividir el resultado entre dos, se ha calculado la distancia en un sentido solamente, por lo que se agilizan los cálculos.

3.4. Operador de selección

El operador de selección, como su nombre indica, es el encargado de seleccionar de una población dada las soluciones que se van a utilizar para generar la población de la siguiente generación. Para este algoritmo se han implementado dos operadores diferentes. El primero está basado en una selección por ruleta, donde cada solución tiene una probabilidad de ser seleccionada en función de su coste, después se genera un número aleatorio y se escoge una de las soluciones. En este caso, cuanto mayor sea la función objetivo asociada a una solución, más posibilidades tendrá esa solución de ser escogida.

El otro operador escogido es el de selección por torneo, en el que mediante un parámetro especificado se escoge el número de soluciones que participan en el torneo. Entre los posibles candidatos que participan, se escoge aquel cuyo valor de la función objetivo sea mayor. Para los experimentos finales se escogía este operador sobre el otro porque proporcionaba mejores resultados en las pruebas realizadas.

3.5. Operador de cruce

Una vez se han seleccionado dos soluciones con el operador de selección, se deben combinar para producir dos nuevas soluciones descendientes que hereden propiedades de las soluciones escogidas. Para esta representación, el problema principal radica en que dos soluciones pueden contener el mismo elemento y realizar una operación de cruce puede producir soluciones con elementos repetidos. Para solventarlo, se comparan ambos padres y se extraen los elementos en común que pasan directamente a los hijos. Después se aplica una operación de cruce de punto único sobre las listas que contienen las características de ambos padres sin los elementos en común. Se establece un punto aleatorio y se cortan ambas listas, a cada hijo se le asigna el primer segmento del primer o segundo padre, después se concatenan los elementos que ambos padres tenían en común, y por último se añade el segundo segmento del primer o segundo padre, forzando a que cada hijo contenga un segmento de cada padre.

3.6. Operador de mutación

Para el operador de mutación, el problema de esta representación radica en que al cambiar uno de los elementos, puede ser que el elemento nuevo tenga un valor que ya estaba en la lista, lo que quiere decir que el elemento estaría repetido dos veces. Para ello se comprueba que el valor generado no esté en la lista antes de hacer el cambio, y si lo está se genera un número nuevo.

El operador planteado itera a lo largo de los elementos de las soluciones generadas, generando números aleatorios. Si uno de estos números es menor a un umbral especificado por el usuario, el número sobre el que se estaba iterando se sustituye por otro elemento del conjunto N, asegurandose como se ha mencionado, que este elemento no se encotraba en la lista. De esta forma se introduce cierta componente de aleatoriedad para ir explorando el espacio de soluciones. Podría darse el caso en que ningún elemento de la solución se modificase, pero también que se modifiquen más de uno.

3.7. Estructura general del algoritmo

En este apartado se plantea la estructura del método *run_evol* que es el que se encarga de realizar todo el proceso evolutivo llamando a cada una de las funciones explicadas y utilizando los operadores detallados. Cabe destacar que se ha implementado una estrategia elitista para conservar las dos mejores soluciones de cada generación. El criterio de parada se da cuando se alcanza el número de generaciones especificadas por el usuario. El pseudo código de este método podría ser el que se encuentra a continuación. Con esto ya quedaría explicado el algoritmo implementado.

Algorithm 1 MDP_int

Object atributes:
NP: Population size ▷ Tamaño de la población
P_m: Mutation probability ▷ Probabilidad de que ocurra una mutación
NG: Number of generations ▷ Número de generaciones
NT: Tournament size ▷ Tamaño del torneo en el proceso de selección
Inputs:
$Path$: Path to text file ▷ ruta y nombre del archivo de texto
Outputs:
P_f: Final population ▷ Población final obtenida
F_f: Final fitness ▷ Coste de la población final
// Read distances and generate initial population
$Dist\_dic \leftarrow read\_distances(Path)$
$P \leftarrow generate\_population()$
// Evolution process
for $i = 1$ to NG **do**
  $F \leftarrow evaluate\_population(P, Dist\_dic)$
  $P, F \leftarrow sort(P, F, key = F)$ ▷ Se ordena la población y los costes según el coste
  $P_0 \leftarrow \emptyset$
  $P_0 \leftarrow P[:2]$ ▷ Se conservan las dos mejores soluciones
  while $|P_0| < NP$ **do**
    if NT **then** ▷ Si el tamaño de torneo tiene un valor, se realiza selección por torneo
      $Parents \leftarrow tournament\_selection()$
    else ▷ Si tiene valor nulo, se realiza selección por ruleta
      $Parents \leftarrow roulette\_selection()$
    end if
    if $Parents_1 \neq Parents_2$ **then** ▷ Si los padres son iguales, seleccionar otros padres
      $offspring \leftarrow crossover\_operation(parents)$
      $offspring \leftarrow mutation\_operation(offspring)$
      $P_0 \leftarrow P_0 \cup offspring$ ▷ Se añaden los descendientes generados a la nueva generación
    end if
  end while
  $P \leftarrow P_0$
end for
$P_f \leftarrow P$
$F_f \leftarrow evaluate\_population(P_f, Dist\_dic)$

4. Resultados

Los resultados de cada caso se pueden encontrar en el archivo Excell adjunto llamado “Resultados”. También se proporcionan las poblaciones finales obtenidas para cada caso, así como el valor de la función objetivo para estas poblaciones, en dos archivos de texto llamados “Poblaciones obtenidas” y “Costes obtenidos”. Se ha fijado la semilla durante los experimentos para que estos resultados sean reproducibles.