ALGORITMOS GENÉTICOS: PROBLEMA DE LA ASIGNACIÓN DE TURNOS

David Mallasén Quintana e Iván Prada Cazalla
Doble grado en Ingeniería Informática y Matemáticas

TABLA DE CONTENIDO

1 Descripción de la representación de los individuos

Para la representación de los individuos se ha utilizado un array de enteros. Este array tenía tantas posiciones como **turnos** pueda tener nuestro horario (inicialmente nos dicen 16), y en cada casilla se le podrá asignar un número entre el **0** y el **número de profesores** disponibles (ambos inclusive). Utilizaremos el cero para denotar que no se ha asignado ningún profesor a ese turno. Por tanto, a cada uno de los profesores dados por la entrada se les asignará un número, en orden creciente y desde el uno, según se vayan leyendo de la entrada.

Tendremos la clase *TurnosCFIHorario* que encapsulará las acciones que queramos hacer sobre el horario como puede ser obtener información filtrada de los turnos. Mediante el constructor podemos pasar fácilmente de la representación interna de AIMA usando *Individual<Integer>* a un objeto de nuestra clase.

1 Descripción de la generación de la población inicial

Para la generación de la población inicial, utilizaremos el método *TurnosCFIGenAlgoUtil.generateRandomIndividual(int numProfs, int numTurnos)*, al que llamaremos tantas veces como individuos queramos añadir a nuestra población inicial, y que devuelve un individuo válido.

Este método se encargará de generar un individuo de la siguiente forma: inicializará a cero nuestro individuo, para no tener así profesores asignados inicialmente. Después, irá seleccionando turno y profesor (seleccionando la casilla en la que se va a poner, así como el profesor, de forma aleatoria) y, justo antes de asignárselo, comprobará si el profesor tiene ese turno como restricción. Si puede asignarlo, lo hará y añadirá uno al contador de asignados. En caso de que el turno sea una restricción, simplemente probará otra opción hasta que finalmente se asignen todos los turnos.

2 Descripción de la implementación del operador de cruce

El operador de cruce lo hemos implementado en el método *AlgoritmoGenetico.reproduce(Individuo x, Individuo y)* que devuelve el hijo producido por el cruce. Este viene dado mediante la técnica del "***corte por un punto***", en la que cogemos el primer trozo del primer padre (del inicio hasta el punto de corte, elegido aleatoriamente), el segundo trozo del segundo padre y los unimos en lo que va a dar el hijo. Como lo que esto genera no siempre es válido (ya que puede que queden más o menos turnos de los necesarios asignados), mientras que el número de turnos asignados no coincida con los que necesitamos, se irán quitando o generando aleatoriamente asignaciones profesor-turno que cumplan las restricciones.

3 Descripción de la implementación del operador de mutación

Para el operador de mutación, desarrollado en *AlgoritmoGenetico.mutate(Individuo child)* se desarrollará la mutación uno a uno, que consistirá en ir mutando uno a uno cada uno de los turnos del horario. Una vez hecho esto, utilizaremos el mismo código usado en el algoritmo de cruce que se encargaba de hacer que el individuo fuera válido cuando tiene más o menos turnos asignados de los correspondientes.

Para realizar la mutación de cada turno, seleccionamos un número aleatorio entre 0 y 1 y, si es menor que la probabilidad de mutación, lo mutamos. Para mutar se seleccionará un número aleatorio en el rango de profesores y, si no tiene ese turno como restricción, se asignará este nuevo valor sustituyendo al anterior (incluyendo el caso del cero, que denota la ausencia de profesor).

4 Descripción de la función de fitness empleada

La función de fitness la definimos en la clase *TurnosCFIFitnessFunction* que implementará la interfaz *FitnessFunction<Integer>*.

Calculamos primero el número de profesores que tienen asignado un número de turnos que está en la media de turnos óptima. Cuantos más profesores estén en esta media, mejor distribuidos estarán los turnos. A continuación, calculamos el número de turnos cuyo profesor asignado tiene a dicho turno en su lista de preferencias. De esta manera tenemos en cuenta que se cumplan el mayor número posible de preferencias de los profesores.

Posteriormente, normalizamos al intervalo [0, 1] los dos valores que hemos calculado para poder asignar un peso a cada parámetro. Para ello tenemos definidas dos constantes en *TurnosCFIDemo*, *PESO_TURNOS_EQUIL* y *PESO_PREFERENCIAS*, que hemos inicializado a 50 y 50.

Finalmente, para que nunca se devuelva 0, devolvemos el máximo entre lo que hemos calculado y un valor muy próximo a 0. De esta manera comprobamos que nuestra función de fitness cumple todos los requisitos: Siempre devuelve valores estrictamente positivos, penaliza a las peores soluciones y favorece a las que cumplen lo que queremos optimizar.

5 Descripción de la función objetivo empleada

En la función objetivo comprobaremos que los profesores asignados a todos los turnos tengan a dicho turno en su lista de preferencias, si es posible, y que el número de turnos asignados a cada profesor esté perfectamente equilibrado. Nos basta comprobar que el profesor que tiene más turnos asignados está en la media de turnos ya que un individuo válido tiene todos los turnos asignados entre los profesores disponibles.

A la hora de ver las preferencias de los profesores no siempre se podrá obtener un valor óptimo. Esto lo podemos ver en el caso en que un profesor tenga todos los turnos en su lista de preferencias y el resto de profesores ningún turno. Evidentemente, para que los turnos estén bien repartidos, habrá profesores cuyo turno asignado no esté en su lista de preferencias. Esto también ocurrirá en el caso en que el número de preferencias distintas de todos los profesores sea menor que el número de turnos a asignar. La solución que hemos adoptado ha sido calcular el número de preferencias distintas de todos los profesores cuando introducimos los datos y comparar con este valor. Si hay suficientes profesores con preferencias se alcanzará el estado objetivo, pero como ya hemos comentado, habrá casos en los que no sea alcanzable.

6 Resultados obtenidos por el algoritmo desarrollado para las implementaciones dadas

Para obtener resultados hemos realizado una batería de pruebas para cada uno de los tres ficheros. En ellos se ha realizado lo siguiente:

- configuracionConvocatoria1.txt:
  Realizadas pruebas de tamaño 50, con tope de tiempo de 1000ms.
- configuracionConvocatoria2.txt:
  Realizadas pruebas de tamaño 100, con tope de tiempo de 1000ms.
  Realizadas pruebas de tamaño 50, con tope de tiempo de 5000ms.
- configuracionConvocatoria3.txt:
  Realizadas pruebas de tamaño 100, con tope de tiempo de 1000ms.
  Realizadas pruebas de tamaño 50, con tope de tiempo de 5000ms.

Cada una de estas pruebas nos ha dado el mejor fitness obtenido, el número de veces que se ha alcanzado el objetivo, la media, la varianza y la desviación típica de los valores de fitness obtenidos. Se ha realizado para las combinaciones de:

- Probabilidad de cruce: 0.7, 0.8, 0.9, ó 1.
- Número de hijos devueltos por el cruce: 1 ó 2.
- Cruce destructivo: activado (true), desactivado (false).

Los datos obtenidos los vamos a presentar en una tabla, para facilitar su lectura.

Adjuntamos los datos como tabla en el archivo **Estadistica.pdf**, que serán usados para la comparación de los siguientes apartados de esta memoria.

7 Influencia de la probabilidad de cruce en la aplicación

Para el resto de apartados vamos a centrarnos en los archivos de las convocatorias 2 y 3, ya que el 1 al ser mucho más simple no nos da casi información (debido al bajo número de turnos que son necesarios asignar y que prácticamente siempre obtiene solución óptima enseguida).

Podemos observar tanto el mayor valor de fitness obtenido, así como la media de los fitness y los valores de dispersión son prácticamente los mismos con las distintas probabilidades de cruce. Destacar que hay configuraciones como la que tiene probabilidad 1, devuelve dos hijos y el cruce no es destructivo que producen además de los mejores fitness la menor dispersión.

8 Influencia de obtener dos individuos en el cruce en lugar de uno

Por lo general observamos que se obtienen valores parecidos en todas las medidas que hemos calculado si sólo modificamos el número de hijos de 1 a 2. Sin embargo, obtenemos mejores medias de valores objetivos así como una menor dispersión de los valores obtenidos en los casos en los que combinamos el cruce con dos hijos y el reemplazo es no destructivo.

9 Influencia de la estrategia no destructiva frente a la estrategia destructiva en la aplicación

Detectamos que se obtienen medias de funciones objetivo más altas con el cruce no destructivo, aunque en la configuración 3 se obtienen valores de dispersión más grandes. Sin embargo, en la configuración 2 también disminuye la varianza, luego no podemos saber con certeza cómo influirá en estos valores el incluir la estrategia no destructiva.

Lo que sí se observa claramente es una ligera mejora en la media de los fitness obtenidos, como hemos comentado anteriormente, al combinar la estrategia no destructiva con devolver 2 hijos en el cruce.

10 Parte opcional. Descripción en detalle de la implementación de las partes opcionales realizadas, resultados...

Como parte opcional hemos realizado cruce por dos puntos en lugar del cruce por un punto que implementa el algoritmo de AIMA. Para implementarlo elegimos dos puntos aleatorios e intercalamos la cabeza y la cola de uno de los progenitores con el cuerpo del otro para cada hijo.

Por lo general conseguimos mejorar levemente los resultados, aunque sin ninguna mejora notable. Observamos que conseguimos “pleno” en los resultados para la primera convocatoria (se generan soluciones objetivo para todas las configuraciones de parámetros). Además, para las demás convocatorias se llega a alcanzar en varias configuraciones el máximo fitness que se puede generar para esa convocatoria, aunque por lo general nos aporta resultados muy parecidos.