Universidad de Burgos

Computación Neuronal y Evolutiva

P5: AirTafficController v2

Analizar y desarrollar un algoritmo genético que realice el cálculo automático de la distribución más conveniente de vuelos que solicitan aterrizar en un determinado aeropuerto sujetos a varias restricciones.

Estudiantes:

David Miguel Lozano
Javier Martínez Riberas

Profesor de la asignatura:

Bruno Baruque Zanón

1º semestre 2016

Índice

A. Introducción

El objetivo de esta práctica es desarrollar un programa que implemente un algoritmo genético que permita al usuario realizar el cálculo automático de la distribución más conveniente de vuelos que solicitan aterrizar en un determinado aeropuerto sujetos a varias restricciones. Se experimentará modelando el problema con restricciones débiles y empleando optimización multi-objetivo. Así mismo, se analizará la conveniencia de la solución propuesta y se reflexionará sobre los resultados obtenidos.

La aplicación tiene que ser capaz de adaptarse a las siguientes condiciones:

- El número de pistas del aeropuerto debe ser un parámetro configurable.
- Cada pista acepta un conjunto determinado de tipos de aviones.
- El número de aviones en cada situación puede variar.
- Los aviones tienen un programa de vuelo que incluye:
  - ETA (*Estimated Time of Arrival*): tiempo estimado de llegada calculado en el momento del despegue.
  - Tipo de avión: heavy / big / small. Condiciona el tiempo necesario para su aterrizaje.
  - Vuelo asociado (opcional): vuelo que se desea que aterrice con la menor diferencia de tiempo del vuelo actual.

El programa tiene que conseguir obtener de forma automática la mejor asignación posible de vuelos a aterrizar en cada pista, de forma que se cumplan los siguientes objetivos:

- El tiempo de espera de los vuelos en su conjunto sea el menor posible.
- Solo los tipos de aviones autorizados aterricen en las pistas.
- Que el tiempo transcurrido entre aterrizajes de vuelos asociados sea el menor posible.

Para su resolución se hará uso de la librería para Java JCLEC (Java Class Library for Evolutionary Computation) [5]. La cual, proporciona un framework para programación evolutiva que da soporte, entre otras cosas, a los algoritmos genéticos.

B. Solución propuesta

A continuación se detalla la codificación y configuración del problema para ser resuelto con un algoritmo genético.

B.1. Representación de los individuos

Para representar los individuos se ha utilizado un array de enteros ordenado (`OrderArrayIndividual`). Cada posición del array representaba un avión y el array estaba ordenado por orden de llegada. De tal manera, que la primera posición se correspondía con el primer avión en llegar y el valor de cada posición indicaba el identificador del avión.

Ejemplo de genotipo:

$$\begin{bmatrix} 2 & 3 & 1 & 4 \end{bmatrix}$$

Representa que el primer vuelo en aterrizar fue el 2, seguido del 3, 1 y 4.

B.2. Esquema evolutivo

Se ha utilizado el algoritmo SGE (*Simple Generational and Elitist*). Se trata de un algoritmo elitista que asegura que, en cualquier momento, sólo los mejores individuos pasen a la siguiente generación [4].

B.3. Función de fitness

Para evaluar los individuos, como el genotipo se encontraba ordenado por orden de llegada, se iba iterando sobre él y planificando cada vuelo. La asignación de la pista se realizaba minimizando el ATA, de tal forma, que se asignaba la primera pista que quedase libre. Al finalizar la planificación, se comprobaban las restricciones incumplidas.

Por último, el cálculo del fitness se realizó teniendo en cuenta los siguientes parámetros:

1. Minimizando el retraso acumulado. Es decir, el sumatorio de la diferencia entre el ATA y el mínimo ETA de cada avión.
2. El número de restricciones de pistas violadas.
3. Minimizando el tiempo entre aterrizajes de vuelos asociados. Es decir, la diferencia en valor absoluto de los dos ATAs.

En la primera parte de la práctica, los dos últimos parámetros se implementaron como restricciones débiles. Dando más importancia a la restricción de pistas.

En la segunda parte de la práctica, se ha implementado mediante optimización multi-objetivo, por los que los tres parámetros se han tenido en cuenta por igual.

B.4. Inicialización

La población inicial se inicializa de forma aleatoria. Se ha utilizado el generador `Ranecu`, un generador lineal congruencial avanzado con un periodo aproximado de 10^{18} [2].

B.5. Criterio de parada

El criterio de parada se ha establecido en 1.000 generaciones por defecto.

B.6. Criterio de selección

Para seleccionar un subconjunto de la población se han analizado los siguientes algoritmos:

1. `RouletteSelector`: selección por ruleta [3].
2. `RandomSelector`: selección aleatoria [1].

B.7. Operador de cruce

Para obtener un nuevo individuo basado en el genotipo de sus padres se han analizado los siguientes algoritmos:

1. `OrderOXCrossover`: OX Crossover.
2. `OrderPMXCrossover`: PMX Crossover.

La probabilidad de cruce se estableció en un 75 % por defecto.

B.8. Operador de mutación

Cada gen del genotipo de un individuo tiene, por defecto, un 3 % de probabilidad de mutar. Se han analizado los siguientes algoritmos de mutación:

1. `Order2OptMutator`: mutación 2-opt del genotipo.
2. `OrderSublistMutator`: mutación de una sublista del genotipo aleatoriamente.

B.9. Criterio de reemplazo

Se ha utilizado `OrderArrayCreator`, mediante el cual, los hijos reemplazan directamente a los padres. Para preservar el elitismo, si la mejor solución de la generación anterior no sobrevive, la peor solución se reemplaza por una nueva.

No se han analizado más algoritmos de reemplazo ya que la librería sólo proporciona este para trabajar con `OrderArrayIndividual`.

B.10. Implementación

Para la importación de la librería JCLEC se ha creado una dependencia Maven de esta. Se ha publicado en el siguiente repositorio: JCLEC Maven Repository.

*Se ha añadido el paquete `orderarray` a la versión base de la librería. Ya que, en la versión original sólo se incluye con los ejemplos. Además, se han sustituido las clases con errores por las versiones corregidas proporcionadas con los materiales de la práctica.

La aplicación cuenta con las siguientes clases:

- `Run`: permite lanzar la aplicación seleccionando por parámetro el archivo de vuelos deseado.
- `AirTrafficController`: implementación del algoritmo genético.
- `Airport`: clase que modela un aeropuerto. Posee la lógica para seleccionar la mejor pista para un determinado avión. Además, permite conocer el retraso acumulado, el momento en el que aterrizó el último avión, el total de restricciones de pista violadas y el acumulado del retraso de aviones asociados.
- `Runway`: clase que modela una pista del aeropuerto. Posee la lógica para calcular cuando estará libre para que aterrice un determinado tipo de avión y conocer cuántas veces se ha violado la restricción de pista.
- `Flight`: clase que modela una vuelo. Posee la lógica para calcular el retraso que tuvo y el del vuelo asociado si existiese.

C. Resultados

Los resultados de los distintos experimentos realizados con la primera implementación se encuentran disponibles en el repositorio: Data.

No fuimos capaces de ejecutar nuestros experimentos con VisualJCLEC. Después de configurar todo correctamente (o al menos cómo lo hicimos en la prática ante-

rior), al ejecutar el experimento obteníamos siempre un `NullPointerException` proveniente de su clase `RunExperiment`. Tras perder unas cuantas horas intentado localizar la fuente del error, no fuimos capaces de encontrarlo.

El tiempo perdido con VisualJCLEC unido a las malas fechas en las que nos encontramos, nos impidieron recopilar los resultados de la segunda implementación (que sí que se ha implementado), así como el análisis de estos para completar así la práctica.

D. Análisis

Desafortunadamente no se tuvo tiempo para analizar a detalle los resultados obtenidos. Sí que se observó que la varianza seguía siendo muy elevada y que el peor caso seguía sin reducirse según avanzaban las generaciones.

E. Conclusiones

Sin analizar profundamente los resultados no podemos sacar grandes conclusiones de las implementaciones realizadas. Pero a grandes rasgos, nos ha parecido que la forma en que hemos codificado el problema (utilizando un `OrderArrayIndividual`) no ha sido la óptima para ser resuelta con algoritmos genéticos.

Referencias

[1] JCLEC. Class randomselector, 2008. URL http://jclec.sourceforge.net/data/jclec4-classification-doc/net/sf/jclec/selector/RandomSelector.html.

[2] JCLEC. Class ranecu, 2008. URL http://jclec.sourceforge.net/data/jclec4-classification-doc/net/sf/jclec/util/random/Ranecu.html.

[3] JCLEC. Class rouletteselector, 2008. URL http://jclec.sourceforge.net/data/jclec4-classification-doc/net/sf/jclec/selector/RouletteSelector.html.

[4] JCLEC. Class sge, 2008. URL http://jclec.sourceforge.net/data/jclec4-classification-doc/net/sf/jclec/algorithm/classic/SGE.html.

[5] Sebastián Ventura, Cristóbal Romero, Amelia Zafra, Jose Antonio Delgado, and César Hervás. JCLEC - java class library for evolutionary computation, 2008. URL http://jclec.sourceforge.net/.