Universidad Politécnica de Cartagena

PLANIFICACIÓN Y GESTIÓN DE REDES

Grado en Ingeniería Telemática
Curso 2015-2016

Práctica 4. Algoritmos evolutivos para el diseño de encaminamiento OSPF de mínima congestión ante fallo simple de SRG

Autor:

Pablo Pavón Mariño

1 Objetivos

Los objetivos de esta práctica son:

1. Desarrollar en `net2plan` un algoritmo para el diseño de encaminamiento OSPF (fijación de pesos OSPF), minimizando el promedio de la congestión de red en dos situaciones: (i) cuando no hay ningún fallo, (ii) cuando hay fallo en un y sólo un SRG (Shared Risk Group) definido en la red. El algoritmo se basará en un heurístico de tipo *algoritmo evolutivo*.
2. Analizar los resultados y calidad de la solución. Desarrollar posibles variaciones del algoritmo.

2 Duración

Esta práctica tiene una duración de 1 sesión, cumpliendo un total de 3 horas de laboratorio.

3 Evaluación

Los alumnos no tienen que entregar ningún material al finalizar esta práctica. Este boletín es para el estudio del alumno. En él, el alumno deberá resolver los problemas planteados y anotar las aclaraciones que estime oportunas para su posterior repaso en casa.

4 Documentación empleada

La información necesaria para resolver esta práctica se encuentra en:

- Ayuda de la herramienta `net2plan` (`http://www.net2plan.com/`).
- Instrucciones básicas presentes en este enunciado.
- Apuntes de la asignatura. Sección 12.9 del libro.

5 Problema de determinación de pesos para red IP/OSPF, considerando recuperación IP ante fallos

Partimos de una red IP donde el encaminamiento vendrá determinado por un protocolo como OSPF o ISIS. Cada enlace de la red e tiene asociado un peso w_e, que será un número entero mayor o igual a uno. En la red hay definidos un conjunto F de grupos de fallo o SRGs (*Shared Risk Groups*). Cada SRG $f \in F$ representa un riesgo de fallo en la red que, si se materializa, provoca que un conjunto dado de nodos y/o enlaces de la red, fallen simultáneamente.

El objetivo de este problema es encontrar para cada enlace e, su peso w_e asociado tal que se minimice la peor congestión de red encontrada entre el estado de red de no-fallo, y los estados de red cuando un y sólo un SRG falla (es decir, observando no-fallo y fallo simple de SRG).

6 Algoritmo evolutivo para el problema

El problema de determinación de los pesos OSPF es un problema $\mathcal{NP}$-completo, y por tanto no se han encontrado algoritmos de complejidad polinomial que lo resuelvan óptimamente. El objetivo de este apartado de la práctica es desarrollar en `net2plan` un algoritmo heurístico de tipo evolutivo para este problema.

Las características del algoritmo pedido son:

- El algoritmo se ejecuta a través de la clase `Offline_fa_eaMinCongestionSingleFailureOSPF.java`. Recibirá como entrada una topología con los nodos y los enlaces de la red, con sus capacidades conocidas, un conjunto de demandas con el tráfico ofrecido, y los SRGs con los riesgos de fallo considerados para la red. El algoritmo devolverá un diseño en el que:
  - El tráfico se enruta según los pesos OSPF proporcionados por el algoritmo.
  - Se añade a cada enlace el atributo `linkWeight`, con el valor del peso asociado al mismo.
- Los parámetros de entrada definidos por el usuario son:
  - `maxLinkWeight`: Valor máximo permitido para el peso OSPF de un enlace. El valor por defecto es `maxLinkWeight = 10`
  - `maxExecTimeSecs`: Tiempo máximo de ejecución del algoritmo permitido. El algoritmo debe finalizar tras este tiempo, devolviendo la mejor solución encontrada. El valor por defecto es `maxExecTimeSecs = 10`.
  - `ea_populationSize`: Número de soluciones en la población. En cada iteración, el número de elementos permanecerá constante. El valor por defecto para este parámetro será `ea_populationSize=100`.
  - `ea_offSpringSize`: Número de soluciones "hijo" que se producen en cada generación. Este número no podrá ser mayor al tamaño de la población partida por dos. El valor por defecto para este parámetro será `ea_offSpringSize=50`.
  - `ea_fractionParentsChosenRandomly`: En el proceso de selección de padres, este parámetro indica la fracción del grupo de padres a crear que se elige aleatoriamente. El valor por defecto para este parámetro será `ea_fractionParentsChosenRandomly=0.5`.
- **Codificación de la solución:** La solución se codificará como un array de `double`, que asocia a cada enlace su peso OSPF (que será un número entero).

En las siguientes secciones se detallan algunos aspectos de implementación de los operadores evolutivos.

6.1 Generación de la población inicial

Cada elemento de la población inicial se generará aleatoriamente, eligiendo para cada enlace e aleatoriamente un peso entre uno y `maxLinkWeight`.

6.2 Operador selección de padres

A partir de la población de partida, se eligen $n =$ `ea_offSpringSize` $\times$ `2` elementos que actuarán como padres. De esos n elementos:

- Un número de soluciones igual a $n \times (1 - \texttt{ea\_fractionParentsChosenRandomly})$ (redondeado hacia arriba) se eligen cogiendo los mejores elementos de la población actual.
- El resto de soluciones se eligen aleatoriamente entre toda la población (pudiendo repetirse entre ellos, y con el conjunto escogido anteriormente).

6.3 Operador recombinación

Partimos de un conjunto de soluciones "padre" seleccionado. A partir de él, repartimos aleatoriamente los padres en parejas, tal que cada uno de los `ea_offSpringSize` padres seleccionados aparecen en una y sólo una pareja[1].

Cada pareja de padres seleccionada genera una solución "hijo", cogiendo aleatoriamente para cada enlace e, o bien el peso OSPF proveniente de una solución padre o de la otra (con igual probabilidad).

6.4 Operación mutación

Para cada hijo generado, se aplica un operador de mutación, que consiste en que se elige aleatoriamente un enlace e, y para ese enlace se elige aleatoriamente un peso entre uno y `maxLinkWeight`.

6.5 Operador selección

A partir de la población original, unida a la descendencia generada, se genera la población para la siguiente generación, eligiendo las `ea_populationSize` mejores soluciones entre ellas.

6.6 Ayudas para la realización del algoritmo

Se proporciona al alumno la clase `Offline_fa_eaMinCongestionSingleFailureOSPF.java` ya implementada, por lo que no deberá modificar esta clase en absoluto. Esta clase es un algoritmo ejecutable desde `net2plan` que:

- Define y recibe los parámetros de entrada del algoritmo.
- Crea un objeto del tipo `EvolutionaryAlgorithmCore`, y ejecuta el algoritmo evolutivo llamando al método `evolve` de ese objeto.
- Recibe los resultados del algoritmo, que se supone deben estar disponibles al terminar el método `evolve`, y los graba en la estructura `net2plan`.

Se proporciona al alumno una plantilla esqueleto para la clase `EvolutionaryAlgorithmCore`, en el fichero `EvolutionaryAlgorithmCore_template.java`. El alumno tendrá que modificar este esqueleto, implementando parte de la funcionalidad del algoritmo, tal y como se detalla a continuación:

- Debe cambiar el nombre de la clase a `EvolutionaryAlgorithmCore.java`.

[1] Una forma sencilla de implementar esto es barajar de manera aleatoria las soluciones padre (p.e. usando el método `shuffle` de la clase `java.utils.Collections`). A continuación, se cogen en orden los padres de dos en dos.

- El método constructor ya se encuentra implementado. Este método recibe los parámetros de entrada del algoritmo, y define una serie de variables que serán necesarias para la implementación del algoritmo. Destacamos las variables:
  - `List<double []> population`: Se trata de una lista, con un elemento para cada solución de la población actual.
  - `double[] costs`: Se trata de un array con un elemento para cada solución de la población, indicando el coste (congestión media) de esa solución.
- El método `evolve` ya se encuentra implementado. Define el bucle principal que gobierna el algoritmo evolutivo y llama a las siguientes funciones **que deben ser implementadas por el alumno**:
  - `generateInitialSolutions ()`: Debe generar la solución inicial y almacenarla en las variables `population` y `costs`.
  - `operator_parentSelection ()`: Devuelve la lista con los indicadores de las soluciones de la población elegidas como padres.
  - `operator_crossover (parents)`: Recibe la lista de padres generada por el método anterior, y devuelve una lista con la descendencia (*offspring*) generada.
  - `operator_mutate (offspring)`: Recibe la descendencia generada por el método anterior, y produce una mutación en cada elemento.
  - `operator_select (offspring)`: Recibe la descendencia (tras el proceso de mutación), lo añade a la población actual (almacenada en las variables `population` y `costs`), y genera la nueva población (actualizando las variables `population` y `costs`).
- El método `computeCostSolution`, que computa el coste (congestión) de una solución, **debe ser implementado por el alumno**. Como ayuda, se sugiere que la función `computeCostSolution` llame a la función `computeCongestion`, que calcula la congestión de una red OSPF concreta, dados sus pesos `solution_e`. En el caso es que se considere que un enlace está caído, debe indicarse asignando el valor `Double.MAX_VALUE` a ese enlace en `solution_e`.
- Se proporciona como ayuda al alumno el método `sortAscending`, que recibe un `List<double []>` con una población y un `double []` con sus costes asociados, y modifica estas variables reordenando sus elementos, de tal manera que las soluciones de menor coste estén primero. Este método será útil para implementar los algoritmos de selección de padres y selección de supervivientes para la siguiente generación.

7 Posibles variaciones (opcional)

Se sugieren algunas variaciones al algoritmo que pueden ser intentadas:

- Modificar la selección de padres, de manera que ninguna solución pueda ser elegida dos veces como padre en la misma iteración.
- Modificar la selección de padres, de manera que se elija según el criterio de la rueda de la ruleta (*roulette wheel selection*): en cada elección de padre, un padre se elige con una probabilidad inversamente proporcional a su valor de congestión media. Una misma solución puede ser elegida varias veces como padre.
- Modificar la funcion de coste, para que sea igual a la media de las congestiones entre todos los estados de la red (sin fallo, fallo en enlace 0, fallo en enlace 1...)

- Modificar el operador de recombinación, para que se realice un *crossover* en $\frac{|E|}{10}$ puntos elegidos aleatoriamente dentro del vector solución. Para cada uno de los $1 + \frac{|E|}{10}$ subarrays, se elige con igual probabilidad el heredar de un padre o del otro.
- Modificar el operador mutación de tal manera que, *para cada enlace*, el peso se modifique aleatoriamente, con un probabilidad igual a $\frac{1}{|E|}$.
- Modificar el operación selección para que se realice según el esquema de la rueda de la ruleta, repetido las veces necesarias hasta que `ea_populationSize` soluciones *distintas* se hayan seleccionado.