Universidad Politécnica de Cartagena

PLANIFICACIÓN Y GESTIÓN DE REDES

Grado en Ingeniería Telemática
Curso 2013-2014

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.

5. Algoritmos evolutivos

Los algoritmos evolutivos son algoritmos metaheurísticos que intentan resolver problemas de optimización, replicando el comportamiento de los sistemas biológicos en cuanto a cómo se produce una evolución de las especies favoreciendo la supervivencia del más apto. Estos algoritmos fueron utilizados por primera vez en la década de 1960, y han sido desde entonces aplicados con éxito en numerosos problemas de optimización.

Los algoritmos evolutivos se basan en la generación inicial de una *población* de soluciones posibles para el problema. A continuación, se suceden una serie de iteraciones, emulando cada una de ellas una generación. En cada generación, se parte de una población de soluciones, y se genera una nueva población que sustituye a la actual. Este proceso se realiza aplicando una serie de *operadores evolutivos*:

- Operador selección de padres (*parent selection*): algunas soluciones de la población son elegidas para *reproducirse*. En general, este proceso se diseña de tal manera que las mejores soluciones (p.e. aquellas con menor coste) consiguen reproducirse con más facilidad.

- Reproducción o recombinación (*crossover*): es un proceso por el cual dos soluciones "padre" se combinan para formar una nueva solución "hijo". Este operador debe favorecer que la solución "hijo" contenga características de ambas soluciones "padre", incluyendo una componente aleatoria en la forma en la que estas características se combinan en la nueva solución.
- Mutación (*mutation*): Durante el proceso de reproducción o recombinación, puede producirse una *mutación*, por la cual la solución "hijo" generada puede sufrir un pequeño cambio aleatorio respecto a la recombinación inicial.
- Selección natural (*selection*): La selección de los padres, seguida de la recombinación, y la mutación, producen una serie de soluciones "hijo" (descendencia, o *offspring*). A partir de la población inicial, unida a la nueva descendencia creada, se produce un proceso de *selección natural* por el cual sólo una parte de esta nueva población agregada sobrevive hasta la siguiente generación. El proceso de selección estará también sesgado para que las mejores soluciones sobrevivan hasta la siguiente iteración, siendo lo más habitual que en cada iteración el tamaño de la población no varíe (por lo que el número de soluciones que desaparecen es igual al tamaño de la descendencia en cada iteración).

El siguiente pseudocódigo ilustra el esquema general de los algoritmos evolutivos.

Algorithm 5.1: EVOLUTIONARY ALGORITHM()

main
- P = initial population **comment:** initialize the population of solutions (i.e. randomly)
- x_{best} = best solution in P **comment:** best solution so far, equal to the best solution in P

while stopping criteria does not hold
do
- $Parents$ = operator_parent_selection(P) **comment:** select the parents
- $Offspring$ = operator_crossover($Parents$) **comment:** from parents → offspring
- $Offspring$ = operator_mutate($Offspring$) **comment:** random mutations during crossover
- P = operator_selection($P, Offspring$) **comment:** best solutions survive
- x_{best} = update the best solution so far

return (x_{best})

Naturalmente, los operadores mencionados establecen únicamente reglas generales que se pueden aplicar de múltiples maneras. La manera concreta depende del problema, de cómo codifiquemos la solución, y de las múltiples opciones al aplicar las ideas evolutivas mencionadas. En esta práctica veremos un ejemplo concreto de algoritmo evolutivo, aplicado al problema de encaminamiento OSPF.

Nota: En general, cuando los algoritmos evolutivos se aplican a soluciones codificadas en arrays binarios (cadenas de 0s y 1s), se les denomina *algoritmos genéticos*.

6. 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 media de estos dos valores:

- La congestión de red (máxima utilización en los enlaces), cuando la red no tiene fallos.
- La peor congestión de red que sucede, cuando un y sólo un SRG, entre los definidos para la red, se ha producido, por lo que todos sus nodos/enlaces asociados están caídos.

7. Algoritmo evolutivo para el problema

El problema de determinación de los pesos OSPF es un problema 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 `FA_EA_minCongestionSingleFailureOSPF.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 permtido. 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`, con un elemento para cada enlace, indicando el peso OSPF del mismo. A pesar de estar contenido en un `double`, los pesos de los enlaces deben ser números enteros.

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

7.1. 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`.

7.2. 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 $n \times (1 -$ `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).

7.3. 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).

7.4. 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`.

7.5. 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.

[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.

7.6. Ayudas para la realización del algoritmo

Se proporciona al alumno la clase `FA_EA_minCongestionSingleFailureOSPF.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`.
- 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:
  - `population`: Se trata de un `ArrayList<double []>` con un elemento para cada solución de la población actual.
  - `costs`: Se trata de un `double []`, con un elemento para cada solución de la población, indicando el coste (congestión media) de esa solución.
  - `srgLinks_f`: Para cada grupo de fallo f, contiene una lista con los enlaces que fallan si el riesgo asociado al SRG se materializa. En el caso de que el SRG esté asociado a algún nodo, todos los enlaces entrantes y salientes del nodo se añaden a `srgLinks_f`.
- 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 un `ArrayList<double []>` 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 media) 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 `ArrayList<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.

8. Análisis: ajuste de los parámetros del algoritmo

El comportamiento de los algoritmos evolutivos es muy dependiente del problema. Un correcto ajuste de sus parámetros es a menudo difícil, y requiere varias pruebas tentativas. En general:

- La selección de padres con poca componente aleatoria favorece la intensificación en contra de la diversificación.
- El operador mutación es necesario para garantizar estadísticamente la diversificación (al menos teóricamente de manera asintótica). En general, la mutación genera pequeñas variaciones en la solución (no grandes cambios). Un alto grado de mutación, hace más aleatorio el resultado.
- La selección de supervivientes puede o no tener una componente aleatoria. De nuevo, existe un compromiso entre intensificación y diversificación.

Realizando varias ejecuciones con distintos parámetros, podremos llegar a soluciones distintas, y (con paciencia) observar las distintas tendencias que se producen al variar estos parámetros. El alumno puede observar esto p.e. ejecutando el algoritmo sobre la red NSFNET, utilizando el fichero `NSFNet_N14_E42_complete.n2p`, que incluye la topología y el tráfico.

9. Estimación de la disponibilidad

El alumno debe realizar una evaluación de prestaciones en términos de disponibilidad, para el caso de la red `example7nodes.n2p` con la matriz de tráfico `tm7nodes.n2p`. Para ello:

- Ejecute el algoritmo desarrollado sobre esta red y tráfico, con los parámetros de entrada por defecto. Grabe este diseño como un fichero `n2p` de nombre `p4.n2p`.
- Utilice el *report* proporcionado por `net2plan` para estimar medidas de disponibilidad de red contemplando fallos simples y dobles en enlaces bidireccionales. El tiempo medio hasta el fallo (MTTF) será de 1 año, y el tiempo de reparación 12 horas. ¿Cuál es el algoritmo de provisionamiento que debe utilizar para que se apliquen los segmentos de preotección definidos en el diseño? ¿Cuál es la disponibilidad caso peor entre las demandas de la red? ¿Cuál es el grado de supervivencia de la red? (*weighted network availability* en `net2plan`).

  > *Respuesta*: Debe utilizar el algoritmo de provisionamiento `NRSIM_AA_OSPF_pathRerouting`. La disponibilidad caso peor de las demanda calculada es: 0,999989. El grado de supervivencia de la red es 0,999983.

- Repita los cálculos de disponibilidad utilizando la herramienta *Resilience simulation*, para tiempos MTTF y MTTR en SRGs estadísticamente independientes con distribución exponencial

negativa. ¿Cual es el algoritmo de generación de eventos de fallo y reparación que debe utilizar? ¿Cuál es el algoritmo de provisionamiento que debe utilizar para que se apliquen los segmentos de preotección definidos en el diseño?

> *Respuesta*: Debe utilizar el algoritmo de generación de fallos y reparaciones `NRSim_EG_exponentialSRGFailureGenerator`, y el algoritmo de provisionamiento `NRSIM_AA_OSPF_pathRerouting`.

10. Posibles variaciones (opcional)

Se sugieren algunas variaciones al algoritmo que pueden ser intentadas:

- Modificar la selección de padres, de manera que ninguna solucíón 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 enlece 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.