Universidad Politécnica de Cartagena

PLANIFICACIÓN Y GESTIÓN DE REDES

Grado en Ingeniería Telemática
Curso 2015-2016

Práctica 1. Algoritmos de búsqueda local (*local search algorithms*) y algoritmos avariciosos (*greedy algorithms*)

Autor:

Pablo Pavón Mariño

1 Objetivos

Los objetivos de esta práctica son:

1. Desarrollar en `net2plan` un algoritmo de localización de nodos basado en un heurístico de búsqueda local. Analizar los resultados y calidad de la solución. Desarrollar posibles variaciones del algoritmo.
2. Desarrollar en `net2plan` un algoritmo de diseño de topologías de anillo basado en un heurístico de tipo *avaricioso* (*greedy algorithm*). 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 de búsqueda local: problema de localización de nodos

5.1 Algoritmos de búsqueda local

Los algoritmos de búsqueda local son algoritmos iterativos, con el siguiente esquema:

- Los algoritmos parten de una solución inicial x_0 elegida por cualquier método.
- En cada iteración i, a partir de la solución actual x_i:
  - Calculan el conjunto de soluciones $N(x_i)$, formado por todas las soluciones factibles del problema que sean "vecinas" de la solución actual x_i. La definición de qué es una solución vecina depende del problema.

- Para cada solución vecina $x \in N(x_i)$, evalúan cuál sería su coste. Hay dos formas principales de realizar esta búsqueda en el vecindario:
  * Búsqueda de la mejor solución (*best-fit*). Se evalúan *todas* las soluciones del vecindario. Si ninguna tiene un coste estrictamente mejor que la solución actual, el algoritmo se detiene (x_i es la solución devuelta). En caso contrario, el algoritmo continúa siendo $x_{i+1} \in N(x_i)$ la solución vecina de mejor coste.
  * Búsqueda "primera solución" (*first-fit*). Se evalúan según algún orden definido las soluciones del vecindario. Cuando se encuentre una solución que tenga un coste estrictamente mejor que la solución actual, el algoritmo continúa en la siguiente iteración siendo $x_{i+1} \in N(x_i)$ esta primera solución de mejora encontrada. Es decir, al encontrar una solución de mejora no se evalúan el resto de soluciones vecinas. Si ninguna solución vecina tiene un coste estrictamente mejor que la solución actual, el algoritmo se detiene (x_i es la solución devuelta).

5.2 Problema de localización de nodos

Sea $i = 1, \ldots, N$ un conjunto de ubicaciones. En cada ubicación i se encuentra un nodo de acceso, y uno o ningún nodo troncal. Cada nodo de acceso i se debe conectar a un y sólo un nodo troncal. El coste de adquisición de un nodo troncal se denota como c, y se asume constante (no dependiente de la ubicación donde se vaya a situar). El coste de un enlace que conecta un nodo de acceso en la ubicación i con un nodo troncal en la ubicación j, lo denotamos como $c_{ij} \geq 0$. Cuando el nodo de acceso y el troncal están en la misma ubicación, el coste del enlace será cero ($c_{ii} = 0$).

El objetivo del problema es encontrar (i) cuántos nodos troncales se adquieren y en qué ubicaciones se localizan, y (ii) cómo se conectan los nodos de acceso a los nodos troncales, tal que se minimice el coste total (sumando el coste de los nodos troncales, y los enlaces acceso-troncal).

Es posible formular el problema de la siguiente manera:

Variables de decisión:

$z_j, j = 1, \ldots, N = \{1$ si la ubicación j tiene un nodo troncal, 0 en caso contrario$\}$

$e_{ij}, i \in N, j \in N = \{1$ si el nodo de acceso en ubic. i se conecta con un nodo troncal en ubic. $j\}$

$$\min \sum_j c z_j + \sum_{ij} c_{ij} e_{ij} \tag{1a}$$

$$\sum_j e_{ij} = 1 \quad \forall i = 1, \ldots, N \tag{1b}$$

$$\sum_i e_{ij} \leq N z_j \quad \forall j = 1, \ldots, N \tag{1c}$$

La función objetivo (1a) suma el coste de los nodos troncales ($\sum_j c z_j$) y el coste de los enlaces ($\sum_{ij} c_{ij} e_{ij}$). Las restricciones (1b), una para cada nodo de acceso i, obligan que un nodo de acceso se conecte exactamente a un nodo troncal. Las restricciones (1c), una para cada posible ubicación de nodo troncal j, hacen que si una ubicación j no tiene nodo troncal ($z_j = 0$), entonces ningún nodo troncal está conectado a él ($\sum_i e_{ij} = 0 \Rightarrow e_{ij} = 0 \quad \forall i$).

5.3 Algoritmo de búsqueda local para el problema de localización de nodos

El problema de localización de nodos descrito es un problema NP-completo, y por tanto no existen algoritmos de complejidad polinomial que lo resuelvan óptimamente. El objetivo de este apartado de la práctica es desarrollar en `net2plan` un algoritmo heurístico de búsqueda local para este problema.

Las características del algoritmo pedido son:

- El algoritmo debe implementarse en una clase de nombre `TCA_LS_nodeLocation.java`. Recibirá como entrada una topología con los nodos de la red, asumiendo que cada nodo corresponde a una ubicación donde hay un nodo de acceso, y es posible además ubicar un nodo troncal. El algoritmo devolverá una topología con los mismos nodos, y un enlace unidireccional $i \rightarrow j$ entre aquellos nodos $i \neq j$ cuando el nodo de acceso en la ubicación i está conectado con un nodo troncal en la ubicación j.
- Los parámetros de entrada definidos por el usuario serán:
  - `linkCapacities`: Capacidad constante que se asignará a todos los enlaces acceso-troncal (entre ubicaciones distintas) que produzca el diseño. El valor por defecto de este parámetro será `linkCapacities = 100`.
  - `initialNodeIndex`: La solución inicial de la que partirá el algoritmo estará formada por único nodo troncal en la ubicación de índice indicado por `initialNodeIndex`, y todo el resto de ubicaciones conectadas a él. El valor por defecto de este parámetro será `initialNodeIndex = 0`.
  - `linkCostPerKm`: Coste por km de los enlaces acceso-troncal. La distancia en km entre dos nodos se asume que es la proporcionada por el método `getNodePairPhysicalDistance` de la clase `NetPlan`. El valor por defecto del parámetro `linkCostPerKm` será 1. El coste de un nodo troncal se asume que es siempre igual a 1 ($c = 1$ en la formulación (1)).
- El algoritmo deberá implementar una política de búsqueda local con las siguientes características:
  - La solución inicial será la formada por un único nodo troncal en la ubicación `initialNodeIndex`.
  - Dada una solución x, las soluciones vecinas serán aquellas que *añaden* un nodo troncal a la solución x, en alguna de las ubicaciones que no lo tiene.
  - Se debe implementar una política *best-fit*: se elegirá como siguiente solución la mejor entre las soluciones vecinas, en caso de mejorar la solución actual.

5.4 Ayudas para la realización del algoritmo

Se sugiere que el alumno mantenga en varias variables la información sobre la solución actual consistente en:

- Coste de esa solución.
- Ubicaciones de los nodos troncales de esa solución.
- Para cada nodo de acceso, ubicación del nodo troncal al que está conectado.

Además, se sugiere que el alumno implemente una función que reciba la solución actual, una posible ubicación donde añadir un nuevo nodo troncal, y devuelva el coste de la nueva solución. Esta función de ayuda podrá ser llamada desde el bucle principal del algoritmo para cada solución vecina de la solución actual.

Se recuerda al alumno que antes de añadir los enlaces al objeto `netPlan`, debe eliminar todos los enlaces que el objeto pudiera tener anteriormente, p.e. llamando al método `removeAllLinks`.

5.5 Análisis: compromiso coste de nodos vs. coste de enlaces

Utilice el algoritmo que acaba de desarrollar, para rellenar la siguiente tabla, en la que se muestra cómo varía el diseño de red, para distintos valores del parámetro `linkCostPerKm`.

La tabla debe rellenarse para la red de topología `ATTWorldNet_N90_E274.n2p`. En las ejecuciones, fije el parámetro `linkCapacities = 100` y el parámetro `initialNodeIndex = 0`. Se incluyen algunos valores de la tabla para que el alumno pueda comprobar su implementación.

Tabla topología ATT

`linkCostPerKm`	Núm. nodos troncales	Coste final
0		
0.0001		
0.001	11	30.11
0.01		
0.1		

Responda a las siguientes preguntas.

1. ¿Cómo varía el diseño al aumentar el parámetro `linkCostPerKm`? ¿por qué?

2. Ejecute el algoritmo para el caso `linkCostPerKm = 0.001`, para distintas soluciones iniciales `initialNodeIndex` (p.e. 1, 2, 3...) ¿Varía la solución obtenida y su coste? ¿por qué? ¿Es la variación de coste significativa (p.e. mayor al 10%)? ¿cree que a partir de estos resultados para esta topología puede deducir que el algoritmo se comportará de manera similar para otras topologías? ¿hay alguna garantía de que alguna solución para algún parámetro `initialNodeIndex` sea óptima?

5.6 Posibles variaciones (opcional)

Se sugieren algunas variaciones al algoritmo que pueden ser intentadas:

- Crear un algoritmo que ejecute el algoritmo anterior N veces, una para cada nodo de inicio, y devuelva la mejor solución.

- Implementar la política *first-fit* en lugar de *best-fit*.
- Ampliar la definición de vecindario, de tal manera que se pueda añadir un nuevo nodo troncal, y también eliminar un nodo troncal.

6 Algoritmos tipo *greedy* para el problema TSP

6.1 Algoritmos avariciosos (*greedy*)

Los algoritmos de tipo avaricioso (*greedy*), son algoritmos iterativos con las siguientes características:

- Son algoritmos *constructivos*: comienzan con una fracción de la solución (p.e. fijando valores en algunas variables de decisión), y van añadiendo partes de la solución en cada iteración. No existe una solución completa (y factible) hasta terminar el algoritmo.
- En cada iteración, la decisión que toman fija una parte de la solución intentando mejorar lo más posible alguna estimación del coste esperado. Las partes de la solución fijadas en una iteración *no son modificadas en sucesivas iteraciones.*

6.2 Problema del viajante (*Travelling Salesman Problem*)

Dado un conjunto de nodos N, y un coste c_{ij} de viajar desde el nodo i al nodo j, el problema del viajante consiste en encontrar el recorrido de coste más pequeño que empieza y termina en el mismo nodo, y pasa una vez por el resto de nodos[1].

El problema TSP aparece en numerosas disciplinas, también en el diseño de redes de comunicaciones. Traducido a este ámbito, toma la forma de, dado un conjunto de nodos N, y un conjunto de costes c_{ij} de posibles enlaces entre ellos, encontrar la topología en anillo de coste mínimo que conecta todos los nodos.

6.3 Algoritmo del vecino más cercano para el problema TSP

El problema TSP descrito es un problema NP-completo, y por tanto no existen problemas de complejidad polinomial que lo resuelvan óptimamente. El objetivo de este apartado de la práctica es desarrollar en `net2plan` un algoritmo heurístico de tipo *greedy* que lo resuelva.

Las características del algoritmo pedido son:

- El algoritmo debe implementarse en una clase de nombre `TCA_nearestNeighborTSP.java`. Recibirá como entrada una topología de nodos. El coste entre cada par de nodos se asume que es proporcional a la distancia entre ellos, tal y como la proporciona el método `getNodePairEuclideanDistance`. El algoritmo devolverá una topología con los mismos nodos, y enlaces bidireccionales (dos enlaces unidireccionales en sentidos opuestos) formando el anillo calculado.
- Los parámetros de entrada definidos por el usuario son:
  - `initialNodeIndex`: Nodo inicial del que parte el algoritmo. El valor por defecto de este parámetro será `initailNode = 0`.

[1] El coste de un recorrido es igual a la suma de los costes de los viajes realizados

  - `linkCapacities`: Capacidad constante que se asignará a todos los enlaces de la red. El valor por defecto de este parámetro será `linkCapacities = 100`.

- El algoritmo definirá la topología en anillo, a través de encontrar un orden en el que recorrer los nodos de la red. El nodo inicial será el dado en el parámetro `initialNodeIndex`. En cada iteración, se añade un enlace bidireccional al anillo, que empieza en el último nodo visitado n, y termina en el nodo no visitado que se encuentre más cerca de n. Cuando haya visitado todos los nodos, se añade un último enlace bidireccional entre `initialNodeIndex` y el último nodo visitado para cerrar el anillo. Nota: A este algoritmo se le conoce como el *algoritmo del vecino más cercano* para el problema TSP.

Nota: Se recuerda al alumno que antes de añadir los enlaces al objeto `netPlan`, debe eliminar todos los enlaces que el objeto pudiera tener anteriormente, llamando al método `removeAllLinks`.

6.4 Análisis

Utilice el algoritmo que acaba de desarrollar, para rellenar la siguiente tabla, en la que se muestra cómo varía el diseño de red, para distintos valores del parámetro `initialNodeIndex`.

La tabla debe rellenarse para la red de topología `ATTWorldNet_N90_E274.n2p`. En las ejecuciones, fije el parámetro `linkCapacities = 100`. Se incluyen algunos valores de la tabla para que el alumno pueda comprobar su implementación. La columna `km solución` suma los km de los enlaces bidireccionales (es decir, **no** suma cada enlace dos veces, una por cada sentido).

Tabla topología ATT

`initialNodeIndex`	km solución
0	20834.09
4	
12	
50	
70	

Responda a las siguientes preguntas.

1. ¿Varía mucho (p.e. más del 20%) el diseño al variar el parámetro `initialNodeIndex`? En caso afirmativo, encuentre una explicación de por qué el algoritmo devuelve en ocasiones soluciones significativamente peores.

2. Visualmente, observe que algunas soluciones tienen "cruces de enlaces". ¿Se le ocurre la forma de variar una solución tal que se "descruzasen" y se mejorase el coste resultante?.

6.5 Posibles variaciones (opcional)

El algoritmo del vecino más cercano tiene como factor positivo un coste computacional muy bajo, aunque a cambio puede ofrecer soluciones de baja calidad. El alumno podría implementar un algoritmo que ejecutase el esquema del vecino más cercano tantas veces como nodos tenga la red, utilizando en cada ejecución un nodo distinto como nodo inicial. Al finalizar se devuelve la mejor solución encontrada.