Universidad Politécnica de Cartagena

PLANIFICACIÓN Y GESTIÓN DE REDES

Grado en Ingeniería Telemática
Curso 2015-2016

Práctica 5. Algoritmos de GRASP (*Greedy randomized adaptive search procedure*) y ACO (*Ant colony optimization*) para la resolución del problema TSP

Autor:

Pablo Pavón Mariño

1 Objetivos

Los objetivos de esta práctica son:

1. Desarrollar en `net2plan` un algoritmo GRASP (*Greedy randomized adaptive search procedure*) para la resolución del problema del viajante (TSP, *Travelling Salesman Problem*). Analizar los resultados y calidad de la solución. Desarrollar posibles variaciones del algoritmo.
2. Desarrollar en `net2plan` un algoritmo ACO (*Ant Colony Optimization*) para la resolución del problema del viajante (TSP, *Travelling Salesman Problem*). Analizar los resultados y calidad de la solución. Desarrollar posibles variaciones del algoritmo.

2 Duración

Esta práctica tiene una duración de 2 sesiones, cumpliendo un total de 6 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 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.

El problema TSP descrito es un problema $\mathcal{NP}$-completo, y por tanto no se conocen algoritmos de complejidad polinomial que lo resuelvan óptimamente. El objetivo de este apartado de la práctica es desarrollar en `net2plan` algoritmos heurísticos de tipo GRASP y ACO para este problema.

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

6 Algoritmo GRASP para el problema TSP

6.1 Algoritmos GRASP

Los algoritmos tipo GRASP fueron propuestos a finales de los 1980s por Thomas Feo y Mauricio Resende [1] [2]. Consisten en repetir un número arbitrario de veces lo que se conoce como una iteración GRASP, formada por dos fases:

1. Una fase de construcción, en la que se genera una solución factible con un método *greedy* aleatorizado, tal que se produce una solución distinta en cada iteración GRASP.
2. Una fase de búsqueda local, tomando la solución de la fase anterior como solución inicial.

A lo largo de las iteraciones, la mejor solución encontrada se almacena como resultado. El esquema general se muestra en el siguiente pseudocódigo.

Algorithm 6.1: GRASP-Esquema general()

main
$x_{best} = \emptyset$ **comment:** mejor solución vacía
while no se cumpla la condición de parada de algoritmo
do
$x =$ solución generada por método greedy-aleatorizado
$x' =$ solución de búsqueda local, tomando x como solución inicial
if $c(x') < c(x_{best})$
then $x_{best} = x'$
return (x_{best})

Se sugiere al alumno repasar los apuntes de la asignatura para conocer más sobre los algoritmos GRASP.

6.2 Implementación del algoritmo

Las características del algoritmo pedido son:

- El algoritmo debe implementarse en una clase de nombre `Offline_tca_graspTSP.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:
  - `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`.

  - `alpha`: Factor entre 0 y 1 de aleatorización de la fase *greedy*. Si `alpha = 0`, el método se convierte en el algortimo del vecino más cercano. Si `alpha = 1`, se generan anillos de manera totalmente aleatoria. El valor por defecto de este parámetro será `alpha = 0.5`
  - `linkCapacities`: Capacidad constante que se asignará a todos los enlaces de la red. El valor por defecto de este parámetro será `linkCapacities = 100`.
  - `randomSeed`: Número entero que será la semilla para el generador de números aleatorios. El valor por defecto de este parámetro será `randomSeed = 1`.

- El método *greedy* aleqatorizado es una variación del algoritmo del vecino más cercano. Comenzará por un nodo inicial elegido al azar. En cada iteración, añade un nuevo enlace al anillo, desde el último nodo visitado. Para ello, en cada iteración calcula el enlace de menor y de mayor coste a un nodo todavía no visitado. Sean c_{min} y c_{max} esos costes. A partir de estos valores, crea una lista con los posibles enlaces cuyo coste esté entre c_{min} y $c_{min} + \alpha(c_{max} - c_{min})$. El enlace añadido será uno elegido al azar en esa lista.

- El método *local search* a utilizar será del tipo *best-fit*, y considerará como vecinos de un anillo, todos aquellos anillos iguales a él en todos los enlaces (bidireccionales), salvo en dos.

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`. La capacidad de todos los enlaces añadidos es la dada por `linkCapacities`.

6.3 Ayudas a la implementación

Se proporciona al alumno la clase `Offline_tca_graspTSP_template.java` que servirá como plantilla. Esta clase ya implementa el bucle principal del algoritmo GRASP, y la función que resuelve la fase de búsqueda local a partir de una solución inicial. El alumno deberá únicamente implementar la función `computeGreedySolution`, que implementa la fase *greedy*-aleatorizada. Esta función debe devolver utilizando un objeto `Pair`, con dos objetos: un `ArrayList<Node>` con la secuencia de nodos atravesada por el anillo, y un `Double` con el coste (suma de la distancia de sus enlaces bidireccionales[2]).

7 Algoritmo ACO para el problema TSP

7.1 Algoritmos ACO

La optimización por colonia de hormigas (*Ant Colony Optimization*, ACO), propuesta por Marco Dorigo en su tesis doctoral en 1992, se inspira en el comportamiento de las colonias de hormigas. En su búsqueda de comida, las hormigas inicialmente exploran aleatoriamente un área cercana a su hormiguero. Cuando una hormiga encuentra una fuente de alimento, arranca una parte y lo lleva de vuelta al hormiguero. En su tránsito, deposita en el suelo un rastro químico de feromonas, que otras hormigas pueden oler. La cantidad de feromonas puede depender de la calidad del alimento encontrado. El rastro de feromonas ayuda a otras hormigas a encontrar el camino hacia el alimento. Cuantas más hormigas siguen el mismo camino, mayor es el rastro de feromonas que se produce, lo cual refuerza positivamente el proceso.

El metaheurístico ACO está inspirado en este comportamiento, e intenta replicar sus aspectos principales. El esquema general ACO se muestra en el siguiente pseudocódigo.

[2]Por tanto, para este coste no sume dos veces la distancia de cada enlace.

Algorithm 7.1: ALGORITMO ACO()

main
K = conjunto de elementos que forman las soluciones
H = conjunto de hormigas
τ_k = cantidad (inicial) de feromonas elemento $k \in K$
while condición de parada no se cumpla
do
- **for each** $h \in H$
  - **do** Construir solución greedy-aleatorizada (depende de τ_k)
- x_{best} = actualizar la solucion incumbente
- **for each** $e \in E$
  - **do**
    - Aplicar estrategia de evaporación en τ_k
    - Aplicar estrategia de refuerzo en τ_k

return (x_{best})

Se sugiere al alumno repasar los apuntes de la asignatura para conocer más sobre los algoritmos ACO.

7.2 Implementación del algoritmo

El algoritmo pedido deberá implementar la aplicación del metaheurístico ACO al problema TSP, descrito como *Ant System* en la literatura [3]. Las características del algoritmo pedido son:

- El algoritmo debe implementarse en una clase de nombre `TCA_ACO_TSP.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:
  - `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`.
  - `numAnts`: Número de hormigas en la colonia. El valor por defecto es `numAnts = 10`.
  - `alpha`: Factor que modula la influencia de la cantidad de feromonas en el movimiento de las hormigas. El valor defecto es `alpha = 1`.
  - `beta`: Factor que modula la influencia de la distancia de los enlaces en el movimiento de las hormigas. El valor defecto es `beta = 1`.
  - `evaporationFactor`: Factor entre 0 y 1 que controla el ratio al que se evaporan las feromonas. El valor defecto es `evaporationFactor = 0.5`.
  - `linkCapacities`: Capacidad constante que se asignará a todos los enlaces de la red. El valor por defecto de este parámetro será `linkCapacities = 100`.
  - `randomSeed`: Número entero que será la semilla para el generador de números aleatorios. El valor por defecto de este parámetro será `randomSeed = 1`.
- Inicialmente todos los enlaces ij tienen una cantidad de feromonas igual a $\tau_{ij} = 1$. Cuando construyen una solución, las hormigas eligen un nodo inicial aleatoriamente. En cada iteración,

añaden un nuevo nodo a visitar al anillo. Sea n el último nodo añadido al anillo, N' el conjunto de nodos todavía no visitados. La probabilidad de elegir $n' \in N'$ como siguiente nodo es:

$$p_{n'} = \frac{\tau_{nn'}^{\alpha} b_{nn'}^{\beta}}{\sum_{m \in N'} \tau_{nm}^{\alpha} b_{nm}^{\beta}}$$

Donde denotamos como $b_{ij} = \frac{1}{c_{ij}}$ a una medida del beneficio que se obtiene al incluir el enlace en el anillo. Después de que todas las hormigas hayan construido una solución completa, se recalcula la cantidad de feromonas asociadas a cada enlace. Primero las feromonas se evaporan según la relación:

$$\tau_{ij} = (1 - \rho)\tau_{ij} \quad \forall i, j \in N$$

A continuación, cada hormiga h añade a los enlaces incluidos en su solución, la cantidad $\frac{1}{L_h}$, donde L_h es el coste de la solución creada.

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`. Además, el anillo será bidireccional, con lo que al añadir un enlace, se debe añadir el enlace en sentido contrario. La capacidad de todos los enlaces es la dada por `linkCapacities`.

7.3 Ayudas a la implementación

Se proporciona al alumno la clase `Offline_tca_acoTSP_template.java` que servirá como plantilla. Esta clase ya implementa el bucle principal del algoritmo ACO. El alumno deberá únicamente implementar la función `computeAntSolution`, que implementa el comportamiento de una hormiga. Esta función debe devolver utilizando un objeto `Pair`, dos objetos: un `ArrayList<Node>` con la secuencia de nodos atravesada por el anillo, y un `Double` con el coste (suma de la distancia de sus enlaces bidireccionales[3]).

En la implementación de la función `computeAntSolution`, para el cómputo de la elección de siguiente nodo a visitar, el alumno puede hacer uso de la función `sampleUniformDistribution`. Esta función, recibe un vector `p` con valores no-negativos, y el valor de la suma de todos los elementos de `p`. La función elige aleatoriamente un índice del vector, tal que cada índice `s` tenga una probabilidad de ser elegido proporcional a `p[s]`.

8 Análisis: comparativa de ambos heurísticos

Utilizando el generador de topologías aleatorias `TCA_WaxmanGenerator`, utilizando sus valores por defecto (salvo el número de nodos `N`), genere topologías 5 topologías de 50 nodos, 5 de 100 nodos y 5 de 200 nodos. Ejecute los algoritmos GRASP y ACO con sus valores por defecto para cada red. Complete la siguiente tabla con los valores medios de costes encontrados:

Tabla comparativa

`N`	Distancia media GRASP	Distancia media ACO
50		
100		
200		

[3]Por tanto, para este coste no sume dos veces la distancia de cada enlace.

9 Posibles variaciones y análisis (opcional)

Se sugieren algunas variaciones y análisis utilizando los algoritmos que pueden ser intentados:

- Modifique el algoritmo GRASP, tal que la RCL contenga los k enlaces con los nodos no visitados, de menor distancia. Su hay menos nodos no visitados que k, todos están en la RCL.
- Busque en la literatura variaciones al modelo *Ant system* implementado, que incluya otras estrategias de refuerzo o de evaporación. Por ejemplo: que únicamente la hormiga que mejor solución haya encontrado sea la que deposite feromonas en la fase de refuerzo.
- Supongamos que debe resolver instancias del problema TSP de tamaños típicos de 100 nodos, con un tiempo de ejecución de 5 segundos, utilizando el algoritmo GRASP. Realice una batería de pruebas utilizando varias topologías de 100 nodos creadas con el generador de topologías aleatorias `TCA_WaxmanGenerator`, barriendo distintos valores del parámetro `alpha` = $\{0; 0.1; 0.2; \ldots, 1\}$. Determine el valor del parámetro `alpha` que mejores resultados le proporciona.
- Realice ajustes de parámetros semejantes para el algoritmo ACO implementado.

Bibliography

[1] T. A. Feo and M. G. Resende, "A probabilistic heuristic for a computationally difficult set covering problem," *Operations research letters*, vol. 8, no. 2, pp. 67–71, 1989.

[2] ——, "Greedy randomized adaptive search procedures," *Journal of global optimization*, vol. 6, no. 2, pp. 109–133, 1995.

[3] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: optimization by a colony of cooperating agents," *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 26, no. 1, pp. 29–41, 1996.