Metaheurística
Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Práctica 1
Greedy y Busqueda Local en Problema de Asignación Cuadrática(QAP)

Iván Sevillano García

DNI: 77187364-P

E-mail: ivansevillanogarcia@correo.ugr.es

Grupo del martes, 17:30h-19:30h

30 de marzo de 2018

Índice

1. Descripción del problema QAP

El problema que se nos plantea es el Problema de Asignación Cuadrática(en adelante QAP por las siglas). En él, tenemos una serie de instalaciones las cuales tienen que interactuar entre ellas una cierta çantidad de trabajo", produciendo un coste. Las instalaciones tienen, además, unas determinadas localizaciones en las que se tienen que situar. Dependiendo de la distancia entre cada dos instalaciones, el coste que producen al interactuar es mayor o menor.

Cabe destacar una serie de detalles en el problema:

- **No Euclideo(No Métrico).** Este problema no pone ninguna objeción a que la distancia de una localización de un lugar a si mismo sea mayor que cero. Además, una instalación puede interactuar consigo misma por consiguiente.
- **No simétrico.** Tampoco pone objeción a que la distancia de una localización a otra no sea la misma que de otra a una.

El problema entonces consiste en repartir las instalaciones en las localizaciones de forma que el coste total o trabajo sea mínimo. Esto es fácil de representar con una permutación, que a cada instalación i le asigna una localización $\pi(i)$. Si llamamos d_{ij} a la distancia que hay de la localización i a la j, y f_{ij} la cantidad de trabajo que tiene que mandar la instalación i a la j, la función de coste asociada al problema sería la siguiente:

$$Coste(\pi) = \sum_{i=1}^{N}\sum_{j=1}^{N} f_{ij}d_{\pi(i)\pi(j)}$$

Si consideramos las matrices de distancias y flujos, D y F y las matrices de que representan a cada permutación y su inversa, Π y Π^{-1} respectivamente, se puede representar el coste de una manera más sencilla:

$$Coste(\pi) =< F, \Pi^{-1}D\Pi >$$

Donde la aplicación $<,>$ tiene como argumentos dos matrices de mismas dimensiones y se aplica en la suma de la multiplicación de sus componentes una a una.

2. Breve descripción de los algorítmos utilizados.

En esta sección vamos a explicar brevemente la configuración de una solución concreta del problema, el funcionamiento de la función de coste y de cómo trabajan los dos algoritmos que se describen en el enunciado.

2.1. Solución.

Una solución está perfectamente determinada por un vector de N componentes donde cada una de sus componentes es distinta unas de otras y el rango de valores difiere de 1 hasta N, o lo que es lo mismo, una permutación. Según la misma, la localización i tendrá alojada la instalación con el número que ocupa en el vector la posición i.

2.2. Función de coste.

La función de coste es la descrita anteriormente en la primera sección.

2.3. Algoritmo Greedy.

Este algoritmo es el más simple que podemos pensar. Éste supone que la localización más cercana a todas las demás debe de ser utilizada por la instalación que más trabajo tenga que realizar, así reduce el "sobrecoste por transporte". Este algoritmo entonces se puede reducir a lo siguiente:

- Ordenar las localizaciones de forma ascendente con respecto a cuanto de lejos está esta de todas las demás.
- Ordenar las instalaciones de forma descendente con respecto a la cantidad de trabajo que deben de realizar en total.
- Asignar por orden las instalaciones y las localizaciones.

2.4. Búsqueda local de primer mejor.

Este algoritmo parte de una solución cualquiera, en nuestro caso utilizaremos la obtenida por el algoritmo Greedy anterior, e intentaremos mejorarla. Para ello, debemos definir y explicar distintos conceptos:

- **Espacio de soluciones.** Podemos abstraer el problema a buscar una permutación dentro de S_N que minimice la función de costo. La existencia es trivial al ser un conjunto finito.
- **Entorno de una solución.** Podemos definir que una solución está çerca"de otra cuando la permutación de una se puede obtener como la otra al aplicarle una permutación cíclica de orden dos. Así, cada solución tiene $\frac{N(N-1)}{2}$ soluciones cercanas, o vecinos.

- **Factorización del cálculo de coste.** Entre una solución y otra vecina hay muy pocas diferencias, por tanto la función de coste debe de ser fácil de calcular partiendo de una solución vecina base, pues sólo se han visto modificados algunos factores de la función coste. En concreto, sólo en los que intervienen uno o los dos elementos del ciclo disjunto de orden dos explicado anteriormente.

El algoritmo de búsqueda local se basa en encontrar de entre los elementos vecinos de nuestra solución la que sea mejor y quedarnos con esa. Tras esto, repetir con esta nueva mejor solución hasta que nuestra solución sea la mejor de todo su "vecindario". El algoritmo del primer mejor se queda con la primera solución encontrada que mejore a la anterior.

2.4.1. Busqueda local:Don't look Bits

Para agilizar el proceso de busqueda del primer mejor, vamos a desarrollar una estrategia para ahorrarnos calculos. En esta estrategia, vamos a marcar los emplazamientos que no han producido mejoras en búsquedas anteriores y no los vamos a comprobar. Así, nos ahorramos tiempo de cómputo. No nos da la misma solución que la búsqueda local pero nos puede dar una muy buena también y nos ahorra bastante tiempo.

3. Pseudocódigos y explicaciones.

3.1. Algoritmo Greedy

Este algoritmo hace uso de dos funciones distintas. La primera, ordena las filas de una matriz por el valor de la suma de sus valores. No creo necesaria la inclusión de este código ya que cualquier lenguaje de medio nivel tiene una función genérica para ordenar vectores por alguna característica indeterminada.
La segunda simplemente asigna la primera componente del vector ordenado de instalaciones a la última del vector ordenado de localizaciones. Más simple que la función anterior.

3.2. Algoritmo primer mejor: Don't look bits.

3.2.1. Factorización de la busqueda local

3.2.2. Generador de vecino

3.2.3. Generador de soluciones aleatorias

3.2.4. Exploración del entorno con restricción DLB.

4. Algoritmo de comparación.

5. Manual de uso.

Para la implementación de la práctica hemos utilizado el lenguaje precompilado Python3, por lo que debe de estar instalado en el sistema. También hemos hecho uso de la biblioteca random del mismo lenguaje(Aunque no para esta práctica, ya que no hemos generado ningún factor aleatorio).

La forma de utilizar el programa es la siguiente:

- **Programa.** El programa recibe como argumento el nombre del archivo de entrada con los datos(terminado en ".dat"), que debe de estar en el directorio *./qapdata/*. Tiene que haber también un archivo solución con el mismo nombre pero con terminación ".sln.[en] el directorio *./qapsoln/*. Es posible también introducir una semilla como segundo argumento, aunque para este programa no la utilizamos:

  $./Greedy - QAP\ ./qapdata/nombredatos.dat\ semilla$

  Nos dará como resultado:

  - Solución Greedy. La primera linea será la solución greedy obtenida. La segunda, cuanto tiempo ha tardado el programa en calcularla y cuanto coste total tiene esta solución.
  - Solución BL-DLB. La primera linea será la solución obtenida por busqueda local. La segunda, cuanto tiempo ha tardado el programa en calcularla(incluyendo el cálculo de la primera solución greedy necesaria para comenzar el algoritmo) y cuanto coste total tiene esta solución.
  - Mejor solución. La primera linea es la configuración de la mejor solución y la siguiente es su coste.

- **Experimento total.** Para ejecutar el experimento total, nos hemos ayudado de un script que ejecuta todos los casos de prueba y una makefile que lo llama. Así, la forma de obtener todas las soluciones será tan simple como ejecutar el siguiente comando:

  make greedy

  Todos los resultados nos los encontramos en la carpeta *./solutionGreedy/* con el mismo nombre del archivo de datos pero con terminación ".sol"

6. Experimento y análisis de resultados.