MAX–MEAN DISPERSION

Diseño y Análisis de Algoritmos

INFORME

Óscar Darias Plasencia, alu0100892833

Abril de 2017

Introducción

Los problemas de optimización, a día de hoy, suponen un foco importante para el desarrollo de algoritmos en la informática. Por lo general, se trata de problemas complejos y con un gran número de soluciones posibles, pero siendo estas unas mejores que otras. Una persona, por sí sola, tiene bastante complicado resolver uno de estos problemas en cuanto empiezan a hacerse demasiado grandes, mucho menos encontrar la solución óptima, lo cual hace que esta tarea sea delegada en programas que apliquen los algoritmos adecuados.

Un problema de optimización, a grandes rasgos, consiste en la selección del mejor elemento, con respecto a algún criterio, de un conjunto de elementos disponibles. En el caso que nosotros vamos a estudiar, se trata de maximizar una función real eligiendo sistemáticamente valores de entrada de entre un grupo de candidatos. Esta función real se conoce como función objetivo.

Así pues, para problemas pequeños, se podría tratar de examinar todas las combinaciones de entradas posibles para la función objetivo, y entre ellas, quedarnos con la mejor, lo que llamamos el óptimo global del problema. Sin embargo, para problemas muy grandes, esto es completamente inviable, ya que el coste computacional sería demasiado elevado.

Supongamos que tenemos un grafo completo, es decir, con todos sus nodos conectados entre sí mediante aristas que poseen un determinado valor. En el *Max-mean dispersion problem*, al que nos enfrentamos en esta práctica, se busca encontrar el subconjunto de nodos del grafo que maximicen la dispersión media, dada por la siguiente fórmula:

$$md(S) = \frac{\Sigma d(i,j)}{|S|}$$

En los siguientes apartados, veremos los distintos algoritmos que hemos implementado para tratar de obtener la mejor solución posible para varios ejemplos de los que disponemos. El lenguaje de programación que hemos utilizado es Java.

Estructuras de datos

Como es de suponer, la principal estructura de datos que necesitamos para poder resolver el problema es el grafo. Esto normalmente nos llevaría a tener que crear una nueva clase Grafo, compuesta por una serie de objetos Nodo (otra clase a crear) conectados por una serie de Enlaces (otra clase). En total, habría que implementar tres clases nuevas.

Sin embargo, hay que tener en cuenta de que en este problema se trabaja con grafos completos: al estar todos los nodos conectados con todos, para cada nodo existen conexiones hacia todos los demás. Esto puede implementarse abstrayendo una matriz para ser interpretada como un grafo. Cada posición *(i, j)* de la matriz representa el valor de la arista

que une el vértice o nodo *i* con el vértice *j*. Nótese que la diagonal principal de la matriz está formada por ceros, ya que el coste de un nodo hasta sí mismo es 0.

Por tanto, reutilizaremos una clase Matriz empleada para trabajos anteriores de la asignatura, y crearemos una nueva clase `GraphMatrix` que guardará los datos utilizando una matriz. Además, se añadirán una serie de métodos necesarios para obtener la información del grafo de forma mucho más sencilla. Estos métodos son:

- `getAffinity(int firstNode, int secondNode)`. Este método devuelve simplemente el valor asignado a la posición de la matriz `(firstNode, secondNode)`, el cual es el mismo que `(secondNode, firstNode)`. Es el coste asociado a la arista que une dos nodos cualesquiera.
- `getHighestAffinityLink()`. Devuelve la posición de la matriz con el mayor valor, es decir, la arista de mayor beneficio de todo el grafo. El método de nombre `getLowestAffinityLink()` realiza la función opuesta.
- `getBetterNextNode(solution).` Devuelve el mejor nodo posible para añadir a la solución dada por el parámetro. El método `getWorstNextNode(solution)` lleva a cabo la función opuesta.

También se ha creado una pequeña clase `Position` que representa una posición de la matriz: simplemente guarda el valor de la fila y de la columna. El resto del código implementado, que se corresponde con los algoritmos propiamente dichos, se encuentra en la clase `MaxMeanDispersion`. A continuación, vamos a ver cada uno de ellos.

Algoritmos voraces

Un algoritmo voraz o *Greedy* es aquel que trata de resolver un determinado problema siguiendo una heurística consistente en elegir la opción óptima en cada uno de los pasos que va llevando a cabo, con la esperanza de en algún momento llegar a la solución óptima. Generalmente, son los algoritmos más sencillos de implementar para resolver problemas de optimización, pero muy rara vez son capaces de encontrar la solución óptima para un problema.

Nosotros hemos implementado dos algoritmos voraces diferentes: un algoritmo voraz constructivo, y otro que hemos llamado algoritmo voraz destructivo. El funcionamiento de ambos es bastante similar, pero opuesto; mientras uno va añadiendo elementos a la solución, el otro los va eliminando.

Comenzaremos con el algoritmo voraz constructivo. Este parte de una primera solución voraz: se añaden primero que nada los nodos cuyo enlace tiene la mayor afinidad de todo el grafo. A partir de ahí, el algoritmo aprovecha otro método de la clase `GraphMatrix` descrito en el apartado anterior: `getBetterNextNode(solution)`. Mientras el nodo devuelto por este

método mejore a la solución actual, se van añadiendo nodos. En el momento en que uno de estos nodos no mejore a la solución, el algoritmo se detiene. El resultado de esto siempre será el mismo para un mismo problema dado: en cada iteración, puesto que siempre tendrá disponibles los mismos nodos, elegirá siempre el mejor y lo añadirá a la solución.

Max-mean-div-10	Número de nodos	Valor de dispersión	Operaciones ejecutadas
1	10	10.142857142857142	13
2	10	10.142857142857142	13

Max-mean-div-15	Número de nodos	Valor de dispersión	Operaciones ejecutadas
1	15	5.0	3
2	15	5.0	3

Max-mean-div-20	Número de nodos	Valor de dispersión	Operaciones ejecutadas
1	20	8.75	7
2	20	8.75	7

Max-mean-div-25	Número de nodos	Valor de dispersión	Operaciones ejecutadas
1	25	8.0	5
2	25	8.0	5

Ahora veamos el algoritmo voraz destructivo que hemos ideado. Puesto que en este tipo de problemas no tenemos limitaciones, como podría ser la limitación de peso en el problema de la mochila, podemos considerar una solución factible aquella formada por todos los nodos del grafo. A partir de ella, podemos ir retirando aquellos nodos que estén "perjudicando" a la solución, es decir, aquellos nodo que, al retirarlos de la solución, resulta la dispersión media en el valor más alto posible. En cada iteración, el nodo que cumple estas condiciones es encontrado por un método de la clase `GraphMatrix`, `getWorstNextNode(solution)`.

Max-mean-div-10	Número de nodos	Valor de dispersión	Operaciones ejecutadas
1	10	6.0	3
2	10	6.0	3

Max-mean-div-15	Número de nodos	Valor de dispersión	Operaciones ejecutadas
1	15	6.2	19
2	15	6.2	19

Max-mean-div-20	Número de nodos	Valor de dispersión	Operaciones ejecutadas
1	20	-1.2222222222222223	3
2	20	-1.2222222222222223	3

Max-mean-div-25	Número de nodos	Valor de dispersión	Operaciones ejecutadas
1	25	0.3181818181818182	5
2	25	0.3181818181818182	5

Nótese que en el algoritmo destructivo, al igual que en el constructivo, al igual que en el constructivo, los resultados siempre son exactamente los mismos, por las mismas razones. Sin embargo, podemos ver claramente que, para las instancias del problema que hemos ejecutado, el algoritmo constructivo funciona considerablemente mejor. La única excepción a esta regla es el problema de 15 instancias, cuya solución mejora ligeramente su valor objetivo.

Algoritmo GRASP

El algoritmo GRASP podría verse como una mejora sobre los algoritmos voraces más tradicionales. Mientras los algoritmos voraces son exclusivamente constructivos (ya que construyen la solución y finalizan), el algoritmo GRASP añade una fase de mejora de la solución obtenida, que normalmente se lleva a cabo mediante una búsqueda local. Así pues, un GRASP primero construye la solución y luego examina todas sus soluciones vecinas para quedarse con la mejor.

Además, GRASP introduce un cierto componente de aleatoriedad a las soluciones que proporciona. Un algoritmo voraz, para una misma instancia de un mismo problema, genera siempre la misma solución. El GRASP, en cambio, otorgará soluciones diferentes en diferentes ejecuciones. Esto se debe a la aplicación de la lista restringida de candidatos (RCL). En lugar de seleccionarse únicamente el siguiente mejor nodo, se selecciona un número determinado de ellos y se introducen en la RCL. Acto seguido, se selecciona al azar uno de estos. Hay que tener en cuenta que, aunque las RCL tienen un tamaño definido, estas pueden terminar siendo de un tamaño menor si no hay suficientes soluciones que mejoren a la solución actual como para formar una RCL del tamaño requerido.

Max-mean-div-10	RCL	Número de nodos	Valor de dispersión	Operaciones ejecutadas
1	2	10	12.714285714285714	29
2	2	10	12.714285714285714	29
3	2	10	13.0	25

Max-mean-div-10	RCL	Número de nodos	Valor de dispersión	Operaciones ejecutadas
4	2	10	13.0	41
5	2	10	12.714285714285714	33
6	3	10	13.0	25
7	3	10	12.714285714285714	29
8	3	10	13.0	25
9	3	10	12.714285714285714	29
10	3	10	12.714285714285714	33

Max-mean-div-15	RCL	Número de nodos	Valor de dispersión	Operaciones ejecutadas
1	2	15	9.5	17
2	2	15	9.5	17
3	2	15	7.666666666666667	17
4	2	15	9.5	21
5	2	15	9.833333333333334	29
6	3	15	9.5	17
7	3	15	9.833333333333334	37
8	3	15	9.5	17
9	3	15	7.666666666666667	17
10	3	15	9.5	17

Max-mean-div-20	RCL	Número de nodos	Valor de dispersión	Operaciones ejecutadas
1	2	20	12.857142857142858	29
2	2	20	12.857142857142858	37
3	2	20	12.857142857142858	29
4	2	20	12.857142857142858	29
5	2	20	12.857142857142858	33
6	3	20	12.857142857142858	37
7	3	20	12.0	33
8	3	20	12.857142857142858	29
9	3	20	12.857142857142858	29

Max-mean-div-20	RCL	Número de nodos	Valor de dispersión	Operaciones ejecutadas
10	3	20	12.857142857142858	37

Max-mean-div-25	RCL	Número de nodos	Valor de dispersión	Operaciones ejecutadas
1	2	25	14.125	33
2	2	25	11.714285714285714	29
3	2	25	14.714285714285714	29
4	2	25	14.125	37
5	2	25	14.714285714285714	53
6	3	25	11.714285714285714	33
7	3	25	13.0	29
8	3	25	14.375	37
9	3	25	13.285714285714286	37
10	3	25	14.5	33

Algoritmo Multi-Arranque

Un algoritmo multi-arranque genera un determinado número de soluciones aleatorias y luego mejora cada una de ellas haciendo uso de la búsqueda local. De entre todas las resultantes, se queda con la mejor. En el algoritmo multi-arranque que hemos implementado en nuestro programa, realizamos tantas llamadas al algoritmo GRASP como arranques queramos llevar a cabo. Cabe destacar que es un GRASP "reducido", sin fase de mejora: simplemente genera soluciones de forma voraz con cierta aleatoriedad.

Cuando se tienen todas las soluciones de arranque deseadas, se lleva a cabo la búsqueda local sobre cada una de ellas y se escoge la mejor. Esa última se considera la solución final. El algoritmo funciona de esta manera para tratar de escapar de los óptimos locales, explorando varias zonas aleatorias del espacio de soluciones.

A continuación, veamos los resultados con 2, 4 y 8 arranques para el problema de 25 nodos:

Max-mean-div-25	Arranques	Valor de dispersión	Operaciones ejecutadas
1	2	9.8	40
2	2	11.714285714285714	40

Max-mean-div-25	Arranques	Valor de dispersión	Operaciones ejecutadas
3	2	11.833333333333334	21
4	2	10.909090909090908	22
5	2	11.727272727272727	37
6	4	9.0	82
7	4	9.0	95
8	4	11.181818181818182	91
9	4	9.8	106
10	4	11.714285714285714	85
11	8	9.0	220
12	8	11.833333333333334	204
13	8	9.8	241
14	8	10.75	193
15	8	12.444444444444445	199

A continuación, para el problema de 15 nodos:

Max-mean-div-15	Arranques	Valor de dispersión	Operaciones ejecutadas
1	2	7.166666666666667	22
2	2	9.833333333333334	25
3	2	9.5	25
4	2	9.5	22
5	2	9.833333333333334	21
6	4	9.833333333333334	73
7	4	9.833333333333334	69
8	4	9.833333333333334	66
9	4	9.833333333333334	82
10	4	9.833333333333334	73
11	8	9.833333333333334	172
12	8	9.833333333333334	160
13	8	9.833333333333334	163
14	8	9.5	151

Max-mean-div-15	Arranques	Valor de dispersión	Operaciones ejecutadas
15	8	7.166666666666667	167

Búsqueda por entorno variable

La búsqueda por entorno variable (VNS) trata de escapar de los óptimos locales provocando una perturbación aleatoria dentro de una vecindad y realizando otra búsqueda local a partir de la nueva solución. De forma sucesiva, se va quedando con la mejor solución que va encontrando. Además, para facilitar la mejora de cada solución, se definen un conjunto de estructuras de entorno y no solo una, de tal manera que si para una no se encuentra una solución mejor, se prueba con las demás.

Nuestro algoritmo VNS es lo que se definiría como un VNS básico (BVNS). Las estructuras de entorno están anidadas con otras: la tercera estructura de entorno rodea a la segunda estructura de entorno, esta a la primera, y lo mismo para las demás. Así, el procedimiento `shake` de la clase `MaxMeanDispersion` intercambia un nodo aleatorio de la solución con un nodo aleatorio que no esté en la solución. Para la segunda estructura de entorno, intercambia dos nodos, para la tercera tres, etc. El número de estructuras de entorno establecidas por defecto es de 2, pero probaremos también para cada ejemplo con 3 estructuras de entorno.

Max-mean-div-10	Nº Estructuras de entorno	Valor de dispersión	Operaciones ejecutadas
1	2	12.714285714285714	42
2	2	14.0	38
3	2	14.0	39
4	2	14.0	34
5	2	14.0	34
6	3	13.0	32
7	3	14.0	32
8	3	13.0	32
9	3	12.714285714285714	48
10	3	14.0	36

Max-mean-div-15	Nº Estructuras de entorno	Valor de dispersión	Operaciones ejecutadas
1	2	9.5	22
2	2	9.5	22

Max-mean-div-15	Nº Estructuras de entorno	Valor de dispersión	Operaciones ejecutadas
3	2	9.75	22
4	2	9.75	22
5	2	9.5	22
6	3	9.5	28
7	3	9.5	24
8	3	9.75	24
9	3	9.75	24
10	3	9.833333333333334	32

Max-mean-div-20	Nº Estructuras de entorno	Valor de dispersión	Operaciones ejecutadas
1	2	12.857142857142858	34
2	2	12.857142857142858	38
3	2	12.857142857142858	34
4	2	12.857142857142858	34
5	2	12.857142857142858	38
6	3	12.857142857142858	36
7	3	12.857142857142858	36
8	3	12.857142857142858	36
9	3	12.857142857142858	40
10	3	12.857142857142858	44

Max-mean-div-25	Nº Estructuras de entorno	Valor de dispersión	Operaciones ejecutadas
1	2	14.714285714285714	34
2	2	14.375	42
3	2	13.0	34
4	2	13.0	34
5	2	14.714285714285714	38
6	3	14.714285714285714	48
7	3	14.714285714285714	44
8	3	14.714285714285714	44

Max-mean-div-25	Nº Estructuras de entorno	Valor de dispersión	Operaciones ejecutadas
9	3	14.714285714285714	40
10	3	14.125	40

Conclusiones

Como hemos podido comprobar, para problemas relativamente pequeños, de entre 10 y 25 elementos obtener las mejores soluciones posibles puede resultar todo un reto. Al no poder examinar absolutamente todas las opciones sin que eso suponga un gasto de recursos demasiado alto, se busca utilizar aproximaciones para tratar de acercarse a la solución más óptima posible. Las búsquedas locales son el principal instrumento para esto, así como las mejoras del VNS y el multi-arranque para evitar quedarse atrapados en los óptimos locales.

Otro factor importante que hemos podido observar acerca de estos algoritmos es la progresiva mejora de cada uno de ellos. Sin bien hay excepciones, examinando los resultados obtenidos en las diferentes ejecuciones, podemos ver cómo el algoritmo GRASP puede obtener mejores resultados que los algoritmos voraces. El algoritmo multi-arranque y el BVNS mejoran todavía más en los resultados, pues parten de una solución dada por el GRASP y la mejoran aplicando sus técnicas particulares.