Óscar Darias Plasencia
Maximum Diversity Problem
Diseño y Análisis de Algoritmos
11 May 2017

Problema de diversidad máxima

Para representar un problema de diversidad máxima de forma efectiva, es necesario únicamente manejar vectores. La información de cada problema, al fin y al cabo, se basa en un conjunto de vectores que pueden introducirse o no en una solución. Sin embargo, para facilitar todo posible la programación de los algoritmos, hemos creado una clase *MaximumDiversitySet*. Además, los algoritmos a implementar muchas veces requieren de estructuras de datos adicionales, como es el caso del árbol en el algoritmo de Ramificación y Poda.

En este documento vamos a examinar todas las estructuras de datos utilizadas en la programación de una librería capaz de representar y resolver adecuadamente un problema de diversidad máxima. El lenguaje de programación utilizado ha sido Java y se ha testeado por medio de dos programas ejecutables: uno para los algoritmos Greedy y GRASP, y otro para Ramificación y Acotación.

`MaximumDiversitySet`

Esta clase permite instanciar objetos que representan un problema de diversidad máxima a resolver. También incluye una estructura que define la solución. Así, tiene tres atributos principales:

- La variable `elementSize` almacena el tamaño de cada vector. En los ejemplos de los que disponemos, los vectores son de tamaño 2 o de tamaño 3.
- Un vector de vectores llamado `set` que almacena los datos del problema. Básicamente es un conjunto de vectores asociados cada uno a un índice.
- Un vector de valores booleanos {`true, false`}. Tiene el mismo tamaño que el vector anterior, y el valor de cada posición indica si el elemento con el mismo índice en `set` se encuentra o no en la solución. Almacena la solución del problema.

Su constructor permite leer un fichero con formato de problema y cargarlo directamente en un objeto de este tipo. También posee una serie de métodos fundamentales que nos permiten trabajar mejor con este tipo de objetos, como pueden ser el método que calcula

la diversidad (valor objetivo) de la solución actual, métodos que añaden o retiran elementos de la solución, un método para obtener la distancia euclídea entre dos vectores…

Muy importantes son los métodos *`getFarthest()`* y *`getClosest()`*. El primero permite obtener el elemento más alejado del centro de gravedad de la solución o del centro de gravedad total (en función del valor de su parámetro `general`). El segundo método realiza el procedimiento contrario: encuentra el elemento más cercano al centro de gravedad. Este último procedimiento está pensado para el algoritmo voraz destructivo.

También tenemos algún otro método importante en esta clase, pero está estrictamente relacionado con algún algoritmo, y lo veremos en el apartado correspondiente al mismo.

`GreedyConstructive`

Esta clase contiene los procedimientos necesarios para resolver un problema de diversidad máxima utilizando un algoritmo voraz constructivo. En concreto, solo requiere de un método que recibe como argumentos un objeto `MaximumDiversitySet` y el tamaño de la solución (valor de `m`).

Añade a la solución el elemento más alejado del centro de gravedad del conjunto total de vectores. Luego, mientras el tamaño de la solución no sea el especificado como parámetro, va introduciendo en la solución el elemento más alejado del centro de gravedad de la solución. Para hacer todo esto, aprovecha los métodos explicados en el apartado anterior, de los objetos `MaximumDiversitySet`.

`GreedyDestructive`

El algoritmo voraz propuesto es un algoritmo voraz destructivo, que trabaja a la inversa que el algoritmo voraz constructivo. Está implementado por medio de la clase con el mismo nombre, la cual tiene dos métodos:

- El método `prepareInitialState` hace que todos los vectores del problema pasen a estar en la solución. Esto generaría una solución no factible, pues probablemente el tamaño deseado para la solución sea menor que el número de vectores candidatos.

- El método `solve` toma el objeto `MaximumDiversitySet` en el estado en el que el método anterior lo deja y retira elementos de la solución hasta que esta tiene el tamaño deseado. Para retirar elementos de la solución, selecciona siempre el elemento más cercano al centro de gravedad de la solución actual. Se entiende que este elemento es el que menos está “aportando” a la diversidad de la solución.

LocalSearch

La búsqueda local ha sido implementada por medio de una clase con un único método, que se encarga de mejorar la solución actual del objeto `MaximumDiversitySet` que recibe como argumento. Almacena la diversidad (valor objetivo) de esa solución y luego busca soluciones mejores en su localidad, quedándose siempre con la mejor.

La localidad de una solución determinada viene dada por todas aquellas soluciones que sean resultado de cambiar un elemento que esté en la solución actual por un elemento que no lo esté. El algoritmo implementado comprueba todas estas posibilidades y se queda con la mejor. En total, tiene que revisar `n * m` posibilidades, siendo `n` el número de elementos en la solución actual y `m` el número de elementos que no están en la solución actual.

GRASP

El algoritmo `GRASP` funciona de forma bastante similar al algoritmo voraz constructivo, con el añadido de la lista restringida de candidatos (referida a partir de ahora como RCL), y del hecho de que realiza múltiples intentos y se queda con la mejor solución encontrada.

La clase `GRASP` tiene dos métodos; ambos necesitan un objeto `MaximumDiversitySet`:

- Uno de ellos genera la RCL. Mientras esta no tenga el tamaño deseado, añade a la misma el elemento más alejado del centro de gravedad de la solución actual. En cada llamada a `getFarthest()`, comprueba que el índice devuelto no es -1. Si lo es, significa que ya no hay más candidatos posibles y devuelve la RCL con el tamaño actual.
- El segundo método es el método principal, `solve`, que ejecuta un bucle tantas veces como se indique por parámetro. El número de veces que este bucle se ejecuta equivale al número de intentos realizados por el algoritmo. Empieza añadiendo a la solución el elemento más alejado del centro de gravedad de todos los elementos, y a partir de ahí, mientras la solución no tenga el tamaño deseado, repite la generación de la RCL, escogiendo al azar uno de sus elementos y añadiéndolos a la solución.

BranchAndBound

La ramificación y acotación se implementó por medio de la clase `BranchAndBound`. Esta tiene todos los métodos necesarios para resolver el problema, pero depende de dos clases fundamentales: `BABTree` y `BABNode`. Estas clases permiten representar el árbol necesario para el algoritmo.

- `BABTree` representa un árbol. Es una clase bastante sencilla, con un puntero al primer nodo (de tipo `BABNode`) y un constructor que permite establecer ese primer nodo.

- `BABNode` representa los nodos del árbol. Es una clase algo más compleja que la anterior, pero su funcionamiento sigue siendo el de una clase nodo tradicional. Contiene un objeto `MaximumDiversitySet` asociado, un puntero al nodo padre y una lista de enlaces a los nodos hijos. Esta última está inicialmente vacía, y eso tan solo cambia si se llama al método `generateSons()`. Este genera los nodos hijos, siendo estos todas las posibles variaciones del padre al añadir un nuevo elemento a la solución.

Conociendo el funcionamiento de estas dos clases, podemos pasar a detallar el funcionamiento de la clase `BranchAndBound`, el algoritmo propiamente dicho. Los atributos que almacena son:

- Un objeto `MaximumDiversitySet` con la mejor solución encontrada hasta el momento. Comienza siendo una copia del que se genera con un algoritmo metaheurístico al comenzar el algoritmo, pero cada vez que se encuentra un nodo con una solución mejor, lo sustituye.

- Un objeto `BABTree` para desarrollar el algoritmo.

- El `String exploreStrategy` y el entero `solutionSize`. El primero es una constante que recibe el método solve y que indica la forma de explorar el árbol. Puede tomar cualquier cadena como valor, pero para funcionar correctamente debe tomar un valor dado por las constantes de la clase **`DEPTH_FIRST`** o **`SMALLER_UB`**.

- Un conjunto (emulado mediante un `ArrayList`) de los nodos que ya han sido expandidos. Esto se almacena con el objetivo de no tener que repetir expansiones de nodos. Si un nodo ya ha sido expandido anteriormente, no se vuelve a expandir otro que sea igual.

El método principal para resolver problemas es el método `solve`, al que se le especifican el problema a resolver, el tamaño deseado para la solución, el algoritmo a utilizar para generar la cota inferior inicial y la estrategia de exploración del árbol.

Primero, genera una solución inicial con el algoritmo indicado y lo establece como cota inferior. Luego crea un nuevo objeto `MaximumDiversitySet` sin ningún elemento en la solución, y lo sitúa como raíz del árbol. Luego llama al método `branchOut()`, pasándole como argumento la raíz del árbol.

Este método trata de expandir el nodo que recibe como parámetro. Si el número de elementos en la solución almacenada en el nodo no ha alcanzado el tamaño deseado, genera los nodos hijos y explora utilizando el método correspondiente a la constante especificada para explorar el árbol. Estos métodos, a su vez, realizan una llamada a `branchOut()`, y por tanto, todo el procedimiento es recursivo.

Cabe destacar que los métodos de exploración solo exploran un determinado nodo si este merece la pena ser explorado. Para saber esto, utilizan el método `worthy(BABNode)`, que comprueba que el nodo no ha sido expandido ya anteriormente. Además, también comprueba que no se cumpla la siguiente proposición:

> "Si la distancia máxima desde un elemento A, recién añadido a la solución, hasta cualquier elemento previamente presente en la solución, es menor que la distancia mínima de un elemento B, que no está en la solución, hasta cualquier elemento de la solución, entonces A no puede estar en una solución en la que no esté B."

Por tanto, en ese caso A no merecería la pena añadirlo a la solución, y el nodo no sería explorado.

Por último, si el nodo actual cumple las dos condiciones actuales, aplica una última condición: la cota superior del nodo tiene que ser mayor o igual que la cota inferior menos un margen, que hemos llamado coeficiente de desviación. La cota inferior viene dada por el valor acumulado de la mejor solución encontrada hasta el momento. La cota superior se obtiene aplicando un voraz constructivo sobre el nodo: la diversidad de la solución obtenida es el valor de la cota superior. Esta debe ser mayor que la cota inferior menos el margen.

Este margen debería adaptarse al problema, pero simplemente existe para aportar seguridad de que no se está perdiendo la solución óptima al descartar un nodo, ya que obtener la cota superior utilizando un voraz no asegura precisión absoluta. Por defecto, el margen se establece como el 8% de la cota inferior. Para mejores resultados, el margen tendría que aumentar, pero traería también consigo un aumento considerable del número de nodos generados y, sobre todo, del tiempo necesario para que el algoritmo completara su ejecución. Sin ir más lejos, aumentar mucho el número de nodos implicaría aumentar también mucho la longitud del conjunto de nodos ya expandidos: la complejidad para recorrerlo crece exponencialmente y la simple tarea de comprobar si un nodo ya ha sido expandido podría volverse excesivamente compleja. Hay que tener en cuenta que, con un margen del 8%, el tiempo de ejecución para el algoritmo 30_3 con m = 5 asciende a 37 segundos.

Los resultados obtenidos por los diferentes algoritmos se encuentran en un fichero aparte, el fichero `Resultados.pdf`. Ver para más información.