**INTRODUCCIÓN**

En esta práctica se ha desarrollado un sistema para dividir los productos de una cierta empresa en dos lugares distintos (Java). La complejidad está en que hay que reducir el número de productos separados en las dos empresas que han sido comprados alguna vez juntos, ya que esto puede dar lugar a cancelar la compra de los mismos si están divididos. Para resolver el problema, se ha identificado la división de los productos como un problema de grafos, representando cada producto como un vértice, y siendo las aristas la unión de dos productos si se han comprado alguna vez juntos.

Teniendo ya identificado el problema, se han implementados los algoritmos de “Karger” y “Karger-Stein” que encuentran el mínimo corte en un grafo. El corte mínimo consiste en minimizar el número de aristas que conectan dos vértices de los dos cortes disjuntos generados. Para conseguirlos, aparte de implementar los dos algoritmos nombrados, se han utilizado las estructuras de datos más convenientes para reducir al máximo el coste del algoritmo, generando también numerosas pruebas para comprobar su correcto funcionamiento.

**PRODUCTOS**

En primer lugar, antes de generar el grafo del que hay que calcular el coste mínimo, se debía tener una lista de productos. Esta lista se generó de manera iterativa con productos de valores aleatorios para los campos que se creyó que los definían (nombre, precio, unidades, empresa, descuento…). Para acceder a la información del producto, se necesitaba una estructura de datos rápida capaz de dar con coste constante la información a partir del nombre. Dicha estructura no era otra que una tabla hash que identificara cada elemento por una clave única dentro de la tabla.

Producto: nombre, unidades, precio, descuento y marca

Teniendo ya la estructura decidida, el siguiente paso era definir cuál sería la clave de cada producto. Se pensó que el más adecuado sería el nombre, ya que en teoría no debería haber dos productos con el mismo nombre, pero eso dificultaba mucho las cosas al trabajar luego con el grafo y dar nombres a los vértices. Por ello, se decidió utilizar el nombre como clave en la tabla hash, pero utilizando además otra tabla hash auxiliar, independiente a la anterior, que relacionara el nombre del producto con una clave numérica (el identificador de lo que sería posteriormente el vértice).

|  |  |  |  |
| --- | --- | --- | --- |
| NombreProd1 | Producto1 | Clave1 | NombreProd1 |
| NombreProd2 | Producto2 | Clave2 | NombreProd2 |
| NombreProd3 | Producto3 | Clave3 | NombreProd3 |
| … | … | … | … |
| NombreProdN | ProductoN | ClaveN | NombreProdN |

**EMPAREJAR PRODUCTOS**

Con el listado de productos almacenados en la tabla hash, el siguiente paso era generar los emparejamientos entre los productos. Esta información se supone que se daría al algoritmo por parte de “Amazon” para iniciar el corte en dos partes, pero como no se dispone de ella, se generaron emparejamientos aleatorios entre todos los productos. La idea era emparejarlos, reduciendo al máximo el coste, y manteniendo la consistencia de si un cierto producto había sido comprado junto con otro, que en el segundo también estuviera marcado ese emparejamiento.

Para ello, la información se almacenó en una tabla bidimensional, cuyo tamaño en cada dimensión era el número de productos. La posición fijada por la fila “i” y la columna “j” tomaba el valor true si el producto “i” había sido comprado alguna vez con el producto “j”, y false en caso contrario (se aplica el mismo criterio para el valor de fila y columna intercambiados). Para rellenarla, simplemente se recorrió la mitad de la matriz, obteniendo un número aleatorio. Si el número era par, se marcaba a true su celda y a la que correspondía a los valores invertidos de fila columna, si era false, se ponía el valor false.

**GENERACIÓN GRAFO**

Con los datos de los emparejamientos en la tabla booleana, se procedió a generar el grafo. Para representar un grafo, las dos estructuras de datos más comunes eran matriz de adyacencia o lista de adyacencia. La primera de ellas correspondería a la tabla de emparejamientos nombrada anteriormente, valiendo true si un cierto producto (vértice) ha sido comprado con otro (existe arista entre ambos). La segunda es una lista dinámica con punteros a una lista de vértices con los que está conectado uno dado. Dichas estructuras eran muy eficientes para representar el grafo, pero nosotros no decidimos utilizar ninguna de ellas.

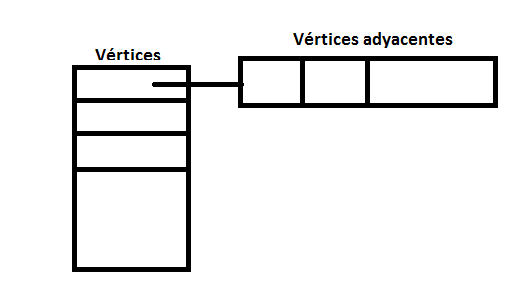
Si analizábamos el algoritmo a implementar, en cada iteración, se eliminaban dos vértices, creando uno nuevo con todas las aristas de los dos por separado. La estructura de matriz de adyacencia, además de tener un coste extra en espacio (se elimina un vértice en cada iteración sin reducir el coste en memoria), se necesitaba recorrerla entera actualizando el valor booleano de cada vértice con la correspondencia de la unión creada. Si mirábamos la lista de adyacencia, se reducía el problema del coste extra en espacio, pero nos encontrábamos ante el mismo problema de recorrer entera toda la estructura para saber si se encontraban los vértices modificados, y actualizar su información con la del nuevo vértice.

Viendo los problemas que generaban las estructuras de datos más comunes, se decidió crear una alternativa que reducía el coste de las operaciones comentadas anteriormente. Esta alternativa estaba compuesta por una tabla hash con todos los vértices que contenía el grafo. Para representar las aristas, se añadió una nueva tabla hash en cada posición de la anterior tabla nombrada. Esta segunda tabla contendría los vértices con los que el vértice de la tabla principal está conectado. La clave de cada índice de la tabla vendría dada por un número que representa el vértice, y cuya correspondencia con el producto se ha comentado en el apartado anterior.

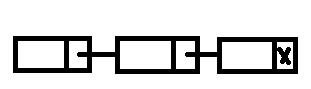
Con esta estructura, añadir un nuevo vértice con la información de los dos unidos tiene coste constante, igual que eliminar los otros dos. Juntar los vértices con los que tienen aristas los dos unidos tendría coste lineal en el número total de aristas de ambos vértices, algo que no mejora en nada sobre las dos estructuras anteriores. Donde está la verdadera mejora de esta estructura de datos es en la operación de actualizar el resto de vértices con la conexión del nuevo. Para realizarlo, la única manera era recorrer la tabla hash principal recorriendo todos los vértices, pero esta vez, no es necesario recorrer el resto de vértices para ver si alguno que era adyacente era alguno de los borrados.

En la matriz y lista de adyacencia, para actualizar la información en todos los vértices del grafo, era necesario recorrer la información del grafo entera; es decir, para cada vértice, mirar si tiene conexión con alguno de los dos borrados, y actualizarla. Para mirar esa conexión, partiendo de un vértice, se debían recorrer todos los demás para ver su conexión con el resto, y realizar el mismo procedimiento para todos. Por todo ello, esta operación tenía coste O(n2), siendo “n” el número de vértices.

Ahora, al estar esta información reflejada en una tabla hash, simplemente accediendo con coste constante a la posición en la que deberían estar los dos vértices borrados (si estuvieran), y modificándolos con la nueva información también con coste constante, se obtiene el grafo actualizado. Sabiendo el coste cuadrático de la operación en las anteriores estructuras, en la nueva que habíamos creado tan sólo tenía coste lineal en el número de vértices, ya que ahora no hace falta recorrerlos todos para ver si hay conexión con los borrados.



A parte de la tabla hash para los vértices adyacentes comentada, el nodo almacenado en la tabla principal tenía que tener también alguna estructura que guardara los vértices (o productos) que se habían unido para generar ese nuevo vértice. Como para este proceso no se necesitaba ningún tipo de eficiencia especial, ya que el coste de todos modos iba a ser lineal en la suma de los vértices unidos, se decidió utilizar una lista dinámica de los identificadores de los vértices ya comentados, haciéndola al menos de este modo eficiente en espacio.



En conclusión, el grafo es una estructura de datos de tipo tabla hash, en la que cada elemento contiene dos estructuras de datos adicionales. La primera de ellas otra tabla hash con los vértices adyacentes para facilitar y reducir el coste de las operaciones de actualización. La segunda una lista dinámica de los identificadores de los vértices que se han unido para generar dicho vértice del grafo. Con todo ello, ya se tiene el grafo final con los productos listo para aplicar el algoritmo de corte mínimo.

**ALGORITMO DE KARGER**

Como ya se ha expuesto en los anteriores apartados un poco para justificar la estructura de datos óptima para el problema, el algoritmo de Karger consiste en ir seleccionando sucesivamente dos vértices del grafo y colapsarlos o unirlos en uno sólo, incluyendo en éste las aristas que contenían los dos anteriores.

**ALGORITMO DE KARGER-STEIN**

El algoritmo de krager-Stein se trata de una mejora del algoritmo de Karger que permite encontrar con una mayor probabilidad el corte mínimo de un grafo. El algoritmo se basa en que la probabilidad de unir dos nodos en el corte mínimo al contraer n nodos hasta tener n/ nodos, es del 50%.

Teniendo en cuenta lo anteriormente explicado, se pueden realizar dos llamadas al algoritmo reduciendo hasta n/ nodos ya que al ser la probabilidad de acierto del 50%, seguramente una de las dos llamadas obtendrá el resultado correcto. Por lo tanto se elegirá el mejor grafo de ambos y se seguirá ejecutando el algoritmo sobre dicho grafo.

Una vez que el número de nodos sea lo suficientemente pequeño se podrá continuar realizando el algoritmo sin necesidad de realizar varias llamadas, ya que la respuesta se podrá encontrar aplicando simplemente fuerza bruta (el algoritmo de contracción que se usaba en la versión de Karger).