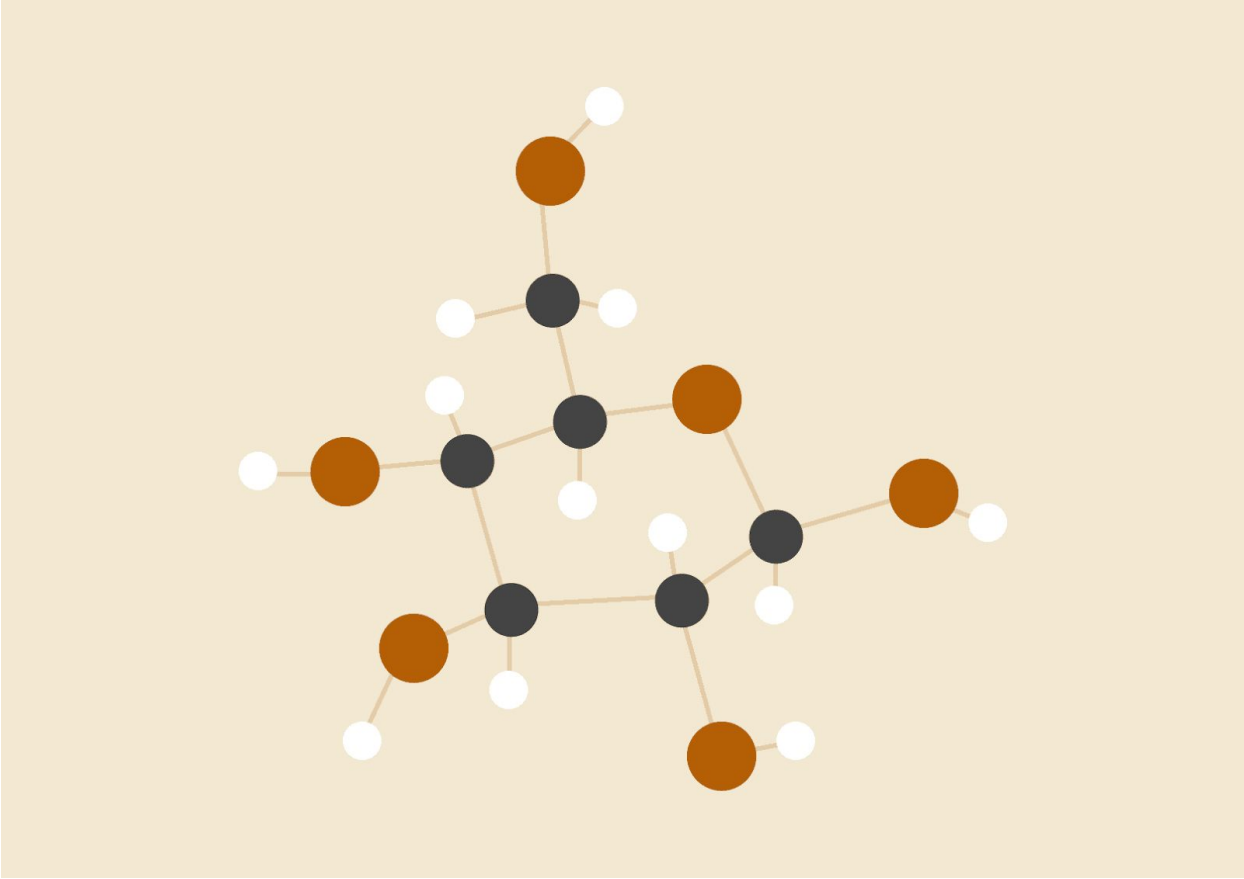
Laboratorio n°1 : matriz dispersa



Rodrigo Mardones

Análisis de algoritmos y Estructura de datos

INTRODUCCIÓN

Las estructuras de datos nos permiten como bien lo indica su nombre, armar representaciones complejas de entidades o estructuras de información que podamos manejar y utilizar para resolver distintos problemas, dependiendo del tipo de estructura.

A su vez, la complejidad algorítmica nos ofrece la posibilidad de calcular la cantidad de recursos necesarios para poder ejecutar un programa.

En el siguiente informe indicaremos los resultados obtenidos del desarrollo de una matriz dispersa escrita en el lenguaje de programación “C”. sus funciones necesarias, su TDA correspondiente y la complejidad algorítmica para obtener la matriz elevada a una potencia.

DESARROLLO

El programa se compone de tres archivos principales, estos son:

- main.c : archivo principal de ejecución, desde aquí se ocupan las dependencias correspondientes.
- lista.h : cabeceras del archivo “lista.c” y sus respectivas definiciones por función
- lista.c : definiciones de funciones de la estructura y su representación como TDA

Para poder correr de forma exitosa el programa es necesario compilarlo y tener como base un archivo de de texto tipo “XXXX.in” donde se contenta una matriz cuadrada, la estructura necesaria es del siguiente tipo:

```
5 3
0 4 0 0 0
-3 0 -5 0 1
0 4 0 0 0
5 0 0 -1 0
0 0 7 8 0
```

La estructura principal para este programa se llama “lista” , la cual consta de una matriz dispersa que aloja los siguientes elementos:

- COL: número entero de columnas
- ROW: número entero de filas
- INSIDE: elemento de tipo nodo que se asume como lista

A su vez la estructura de “nodo” que compone a la lista, está denominada por los siguientes elementos:

- x : posición de fila dentro en la lista
- y: posición de columna dentro de la
- valor: valor de tipo entero dentro de la posición
- next: elemento de tipo nodo que apunta al siguiente nodo dentro de la lista

ANÁLISIS DE COMPLEJIDAD ALGORÍTMICA

Para el análisis de este caso solo veremos el caso de la matriz en potencia de 3, la cual hemos dividido en dos funciones que se encuentran dentro de la definición dentro de los headers y su declaración dentro del archivo “lista.c”, las funciones en cuestion son:

- multiplicar matriz :

```
void multiplicarMatriz(lista* l, lista* n, lista *r){

   nodo * aux = NULL;

   int x = 0;

   for (int i = 0; i < l->rows; i++) {

       for (int j = 0; j < l->cols; j++) {

           x =  0;

           for (int k = 0; k < l->rows; k++){

               int valueNodoA = getValueNodo(l->inside, i, k);

               int valueNodoB = getValueNodo(n->inside, k, j);

               x+= valueNodoA * valueNodoB;

           }

           if(x != 0){

               aux = push(aux, x, i, j);

           }

       }

   }

   r->cols = l->cols;

   r->rows = l->rows;

   r->inside = aux;
```

```
}
```

para el caso anterior el t(n) correspondiente a esta expresión corresponde a :

$$t(n) = n(n(2 + n(5))) + 11$$

$$t(n) = 5n^3 + 2^2 + 11$$

El valor anterior corresponde al peor caso donde se ejecutan todas las lineas de codigo y no hay valores 0 dentro de al buscar en la multiplicación.

Para el caso del O(x) se resuelve de la siguiente manera:

$O(n) = 5n^3 + 2^2$ // se eliminan constantes

$O(n) = n^3 + ^2$ // se eliminan bases multiplicativas

$O(n) = n^3$ // se eliminan casos poco significativos

Para el caso de la función recursiva se tiene la siguiente función:

```
void multiplicarRecursivo(lista *l, lista *n, lista* r, int contador){
   if(contador == 1){
       return;
   }
   multiplicarMatriz(l,n,r);
   multiplicarRecursivo(l,r,r,contador - 1);
}
```

En este caso se tiene que el contador determina la cantidad de llamadas recursivas, por lo que podemos decir que:

- T(n):
    - 1, si acu == 1
    - multiplicarRecursivo(l,n,r, acu -1), si acu > 1

Considerando el valor de la expresión de multiplicar como 1, podríamos decir que que el T(n) de la expresión tiende a T(1).

Ahora bien, al sumar el tiempo de ejecución la multiplicación en O(n) de la respuesta anterior el ejercicio tiene al mismo tiempo de ejecución que multiplicar por lo que podemos decir que el O(n) en ambos casos es el mismo.

CONCLUSIONES

Sin duda una buena forma de práctica para entender complejidad algorítmica dentro de funciones es el ejercicio realizado. Para los casos anteriores las matrices dispersas son una buena herramienta para compresión de información y ahorro en espacio de memoria a la hora de hacer uso de alguna estructura que requiera de guardar y ordenar una serie de datos del mismo tipo. Pues se maneja en tiempo de ejecución la cantidad de memoria a ocupar, lo que la hace mucho más versátil que el uso de matrices comunes, las cuales determinan el uso de memoria en tiempo de compilación, lo que nos lleva a definirlas de esa manera por defecto y no poder modificar su tamaño si así se requiera.

Por último recalcar que si bien el algoritmo diseñado para multiplicar este tipo de matrices no posee un buen excelente T(n), nos ayuda a visualizar qué podemos mejorar para poder crear algoritmos mucho más eficientes a futuro o detectar problemas en términos de complejidad y tiempo para problemas similares.

REFERENCIAS

Para este trabajo se ocuparon las siguientes referencias a fuentes bibliográficas:

- tutorials point. (n.d.). *C - Pointers - Tutorialspoint*. Https://Www.Tutorialspoint.Com/Cprogramming/Index.Htm. https://www.tutorialspoint.com/cprogramming/c_pointers.htm#:%7E:text=A%20pointer%20is%20a%20variable,to%20store%20any%20variable%20address.
- Mañas, J. M. (2017). *Análisis de Algoritmos*. UPM.