CarCarBla-V2

Documentación

Autor: José Manuel de Torres Domínguez

CarCarBla-V2 es la segunda versión de la actividad entregable de grafos. Esta versión incluye todos los métodos y documentación (a diferencia de CarCarBla-V1, entregado en el VPL) que exige el enunciado de la actividad.

El proyecto/programa CarCarBla-V2 está compuesto de 3 componentes principales:

- Clase Grafo
  - grafo.h → Contiene las definiciones de los métodos, constantes y variables, así como una documentación interna que resume lo que hace ese método, así como sus parámetros de entrada y salida.
  - grafo.cpp → Contiene el código de cada método anteriormente descrito en *grafo.h*.
- Programa principal
  - main.cpp → Contiene el código necesario para llevar a cabo las operaciones I/O por pantalla, la llamada y procesamiento de los datos introducidos en la estructura de datos de *Grafo*. Aquí se lleva a cabo, por medio de los métodos de la *Clase Grafo*, las exigencias funcionales de la actividad.

Decisiones principales de diseño (variables internas):

Para llevar a cabo los procedimientos exigidos, se ha propuesto un Grafo implementado con matrices de adyacencia y vectores. En total, se han usado 4 matrices diferentes y dos vectores:

- Constante MAX (static const int MAX=20): Constante que define el máximo de los vectores implementados.
- Vector de Conjuntos de Vértices (string Cjtovertices[MAX]): Incluye todos los vértices existentes dentro del grafo.
- Matriz de Adyacencia (float): Incluye todas las asociaciones entre los vértices existentes del grafo. Se representan en una matriz MAX*MAX, en la que 0 (máximo de float) representa que no existe asociación y distinto y mayor que 0 que existe una asociación con dicho coste.
- Matriz de Floyd (float): Es la matriz en la que se calcula, a partir de la Matriz de Adyacencia, una versión calculada con el algoritmo de Floyd. La representación visual es la misma que la Matriz de Adyacencia. Su contenido representa el coste de ir de un punto X a un punto Y, pasando por todos los vértices necesarios. Si es 0, no existe un camino que permita ir de X a Y.
- Matriz de P (int): Es una matriz adicional que resulta de calcular la matriz de Floyd. Su contenido representa por cuantos vértices intermedios se requiere pasar hasta llegar a un vértice Y desde un vértice X.
- Matriz de Prim (float): Es la matriz en la que se almacena el resultado del algoritmo de Prim, un algoritmo de árboles de expansión mínimo. En concreto, esta matriz será una mezcla de los valores de la Matriz de Adyacencia y la Matriz de Floyd (en el momento de la ejecución contendría los datos de la Matriz de Adyacencia de las carreteras original, no su versión de Floyd).

Estas implementaciones se han hecho de esta manera pues son soluciones fáciles de implementar y sencillas para operar. Los vectores son suficientes para representar conjuntos y las matrices permiten poder hacer consultas de manera rápida y eficiente, a pesar de su consumo de memoria.

Definición de métodos (Clase *Grafo*):

Nota: Dado cualquier método, cualquier propiedad (PRE, POST) no incluida en la descripción se omite debido a que, o bien los parámetros de entrada no requieren restricciones (o no hay parámetros de entrada), o bien el método no devuelve ningún valor, respectivamente.
La descripción puede estar acompañada de una justificación adicional de el por qué se ha elegido esta implementación, si es necesario.

MÉTODO (COMPLEJIDAD) Donde max = CONST INT MAX, n = INT N;	DESCRIPCIÓN
Grafo() O(max²)	DESC: Constructor por defecto de Grafo: Inicializa una instancia de Grafo.
void restablecerMatAdy() O(max²)	DESC: Restablece la Matriz de Adyacencia a su valor por defecto.
void restablecerMatFloyd() O(max²)	DESC: Restablece la Matriz de Floyd a su valor por defecto.
void restablecerMatP() O(max²)	DESC: Restablece la Matriz de P a su valor por defecto.
OPERACIONES SOBRE MATRIZ	
void insertarVertice(string cad1) O(1)	PRE: string cad1 no existente dentro de Cjtovertices. DESC: Inserta un vértice tipo 'String' en el conjunto de vértices.
void insertarArista(string cad1, string cad2, float coste) O(n)	PRE: string cad, cad2 existentes dentro de Cjtovertices. DESC: Inserta una arista, compuesta de dos strings existentes en el conjunto de vértices y un coste, y lo inserta en el conjunto de aristas. Si no se encuentra los strings, no se inserta.
int buscarPosicion(string cad1) O(n)	DESC: Busca en el conjunto de vértices el tipo 'String' indicado por parámetro. POST: Devuelve el índice de dicho vértice en el vector del conjunto de vértices, o -1 en caso de no encontrar nada.
string getVertice(int index) O(1)	DESC: Busca el vértice de tipo 'String' ubicado en la posición del vector del conjunto de vectores dada por index (tipo 'Integer'). POST: Devuelve dicho vértice en forma de 'String'.
void MostrarMatrizAdy() O(n²)	DESC: Muestra por pantalla la Matriz de Adyacencia. Pensadas para su uso en depuración.
void MostrarMatrizFloyd() O(n²)	DESC: Muestra por pantalla la Matriz de Floyd. Pensadas para su uso en depuración.
void MostrarMatrizP() O(n²)	DESC: Muestra por pantalla la Matriz P (relacionada con Floyd). Pensadas para su uso en depuración.
void MostrarMatrizPrim() O(n²)	DESC: Muestra por pantalla la Matriz Prim (relacionada con la Adyacencia y Floyd). Pensadas para su uso en depuración.

CÁLCULOS DE CAMINOS

`void calcularFloyd()`
$O(n^3)$

PRE: MatAdyacencia[MAX][MAX] debe estar inicializada con los valores a calcular. MatP[MAX][MAX] debe estar inicializada a 0.

DESC: Realiza los cálculos necesarios para calcular la Matriz de Floyd a partir de la Matriz de Adyacencia. Primero, copia la Matriz de Adyacencia en la Matriz de Floyd. Después, calcula el algoritmo de Floyd sobre dicha matriz.
La Matriz P recibe modificaciones. Es imperativo que dicha matriz esté vacía antes de operar.

`float calcularMinimoCamino(string cad1, string cad2)`
REC -> O(n)

PRE: Requiere calcular la Matriz de Floyd y P previamente.
DESC: Muestra por pantalla el camino más corto entre dos vértices (de tipo 'String'). El método invoca a la versión recursiva para hacer el recorrido por los vértices y aristas.
POST: Devuelve un 'Float' equivalente al costo total de dicho recorrido, o -1 si no existe un camino posible entre dichos vértices.

Se ha implementado de esta forma para realizar cálculos mínimos de caminos de forma eficiente con el uso de la Matriz P al utilizar el algoritmo de Floyd. Es una de las razones también por el que se utiliza Floyd en primer lugar.

`void calcularMinimoCaminoRec(int i, int j)`
T(n) = T(n-1) + c2 → TM1 : n^(k+1) ->
O(n)

DESC: Versión recursiva que hace el trabajo de calcular el mínimo camino de calcularMinimoCamino(string, string).

La versión recursiva es fácil y rápida de calcular.

`float calcularPrim()`
O(n^3)

while (tamaño de U, que es N)
for(recorre i elementos, hasta N)
for (recorre N elementos)

$$\sum_{i=1}^{n}\left(\sum_{j=1}^{N-i+1}\left(\sum_{k=1}^{n}1\right)\right)$$

PRE: MatAdyancencia[MAX][MAX] inicializada con aristas, MatPrim[MAX][MAX] inicializada por defecto, MatFloyd[MAX][MAX] inicializado con las mismas aristas que MatAdy.

DESC: Utiliza el algoritmo de Prim para calcular un árbol de expansión mínimo de la Matriz de Adyacencia.
Guarda en la Matriz de Prim las aristas de MatAdyacencia[MAX][MAX] resultantes del algoritmo con el costo almacenado en MatFloyd[MAX][MAX] de dicha arista.

POST: Devuelve un entero equivalente a la suma total de los costos de las aristas elegidas.

El algoritmo de Prim es un algoritmo que, pese a su inferior eficiencia a la hora de calcular grandes cantidades de datos en comparación con Kruskal, su relativamente sencilla implementación permite poder desarrollar las funciones del programa de una manera más rápida y clara. Puesto que los datos de entrada no se esperan que sean muchos (no nos esperamos cientos de carreteras interurbanas de una ciudad a otra), Prim nos ofrece una solución sencilla y bastante eficiente para un número de datos de entrada no muy elevados.

OTROS MÉTODOS RELACIONADOS CON OPERACIONES CON MATRIZ

`void copiarPrimAdyacencia()`
$O(max^2)$

DESC: Copia los datos de la Matriz de Prim a la Matriz de Adyacencia.

Nos permite llevar a cabo operaciones con subgrafos dentro de un mismo objeto, tomando las precauciones necesarias.

`void copiarAdyacenciaFloyd()`
$O(max^2)$

DESC: Copia los datos de la Matriz de Adyacencia a la Matriz de Floyd.

Podemos reutilizar la memoria privada asignada de las matrices para realizar varios subgrafos en uno solo, a costa de sacrificar algo de claridad y buenas prácticas de programación.

Conclusión:

La actividad ha sido interesante de hacer, no muy compleja, ni tampoco demasiado larga. Sin embargo, si se introduce el peso de los envíos, exámenes y tareas de las demás asignaturas, se convierte rápidamente en una tarea demasiado larga, que requiere volver a repasar todo el temario de grafos.

Sin duda alguna, el problema principal ha sido los ejemplos propuestos. Uno pierde mucho tiempo intentando figurar por qué una cosa está mal en la salida de su programa, hasta que se da cuenta que ni si quiera la salida del ejemplo es la correcta.

Por otro lado, no creo que hacer un estudio completo para esta tarea sea apropiado: Ya vamos mal con el tiempo de otras entregas, invertir más tiempo en esta del que ya he invertido sería un gran problema.

En general, me habría gustado disfrutar más de esta actividad, porque, mirando hacia atrás la verdad ha sido un reto interesante de hacer y uno termina aprendiendo cosas, pero durante el desarrollo ha sido muy tedioso y, en general, uno siente que siempre va a contrarreloj todo el tiempo.