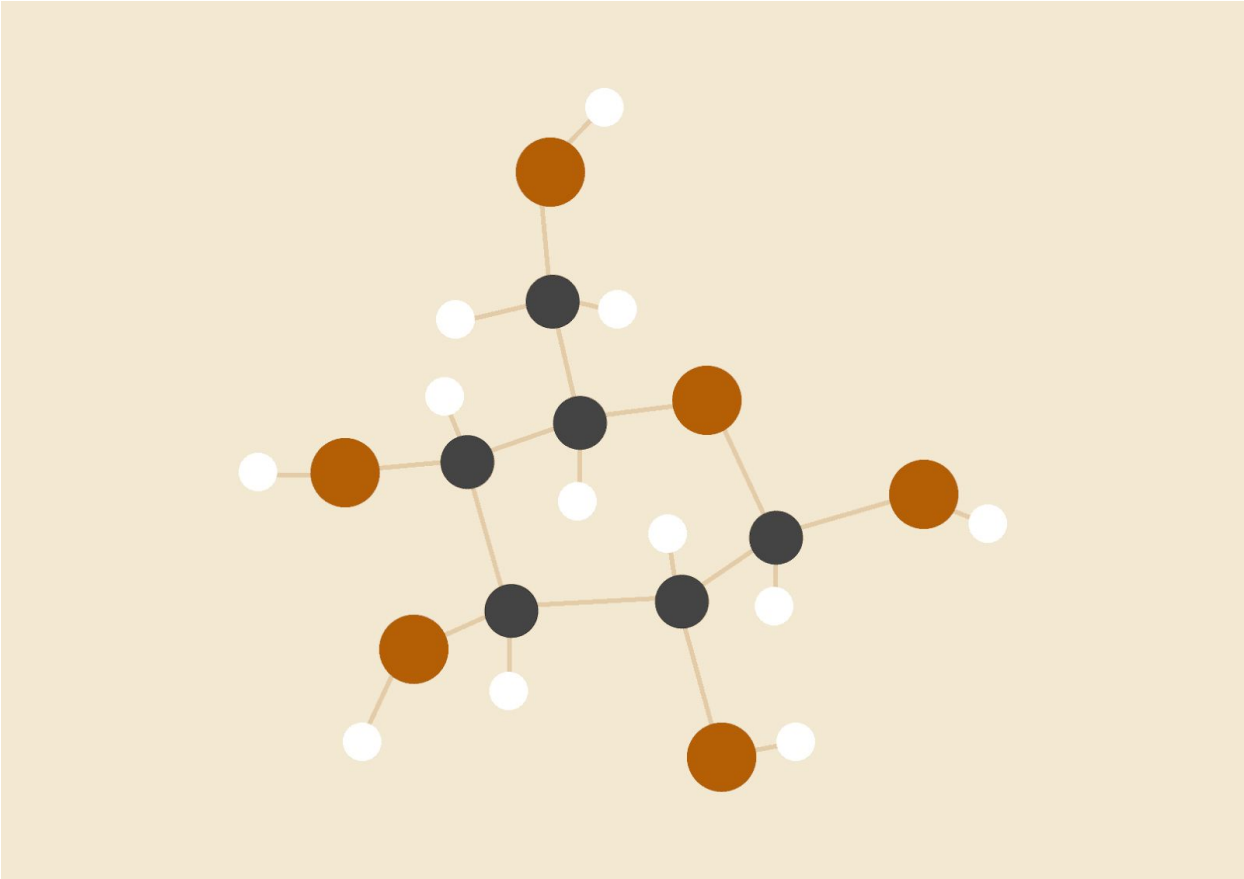
LABORATORIO Nº2: GRAFOS



RODRIGO MARDONES AGUILAR

ANÁLISIS DE ALGORITMOS

Y

ESTRUCTURA DE DATOS

INTRODUCCIÓN

En las ciencias de la computación, las estructuras de datos son, como su nombre lo indica, estructuras o formas de ordenar y organizar datos para que estos puedan representar de mejor manera la información que necesitamos obtener de algún problema específico. Estas estructuras simplifican de sobremanera la resolución de aquellos problemas que, con tipos de datos simples no podemos modelar.

Los grafos, a su vez, son estructuras, son estructuras conformados por dos elementos simples, las “vértices”(elemento de nodo simple) y las “aristas” (relación entre uno o más vértices).

El objetivo general de este informe es resolver el problema planteado en el enunciado, el cual consiste en la búsqueda de puntos críticos de una red de agua. Es decir, detectar aquellos nodos o vértices que, al ser eliminados de una red o grafo, generen un grafo disconexo.

Para esto los objetivos específicos a completar son los siguientes:

- crear TDA de datos necesarias para modelar red de suministro de agua
- definir funciones asociadas a cada TDA para su utilización en la solución
- plantear solución con algún algoritmos de recorrido
- ocupar algoritmo y ver si satisface las necesidades del problema
- análisis de resultados

DESCRIPCIÓN DE LA SOLUCIÓN

Definición de TDA

Para el problema en cuestión, se plantean las siguientes estructuras de datos que nos ayudarán a modelar el problema como un grafo:

- nodo: estructura base la cual contiene un valor llamado “destino” y un puntero al siguiente nodo llamado “siguiente”.
- listNodo: estructura lista de nodos, se plantea esta estructura para crear una lista lineal simple de adyacencia, la cual nos permitirá modelar de mejor manera el grafo, esta contiene un “nodo” llamado “cabeza”.
- grafo: Estructura general la cual contiene un entero llamado “v” que es el número de vértices contenidas en el grafo y un listNodo “array”, que será nuestro arreglo de listNodo , para armar nuestra arreglo de listas lineales simples de adyacencia.

a continuación su implementación en el código:

```
typedef struct Nodo {
   int destino;
   struct Nodo *siguiente;
} nodo;


typedef struct ListNodo {
   nodo *cabeza;
} listNodo;


typedef struct Grafo {
   int v;
   listNodo *array;
} grafo;
```

definición de funciones de TDA

Para los TDA anteriores se han creado distintas funciones que cumplen las necesidades de constructor, accesor, modificador y otros. Estos son:

- crearNodo : crea un nodo simple con un valor de destino específico.
- crearGrafo: crea un grafo a partir de un valor entero para la cantidad de vértices, el grafo creado no tiene ninguna arista asociada en la lista de adyacencia.
- agregarArista: agrega una arista en la posición del arreglo de lista de adyacencia de un grafo específico, asociando la arista de manera inversa entre fuente y destino, al ser un grafo disconexo.
- mostrarGrafo: muestra el grafo de ser necesario para un mayor entendimiento gráfico.
- rellenarGrafo: toma desde el archivo “entrada.in” la cantidad de vértices y una matriz de adyacencia y la pasa a nuestra estructura de grafo, convirtiéndola en un arreglo de listas de adyacencia por vértice.

Cabe destacar que todas las entradas y salidas de cada función de definición para los TDA se encuentran debidamente documentadas en el archivo de cabecera correspondiente.

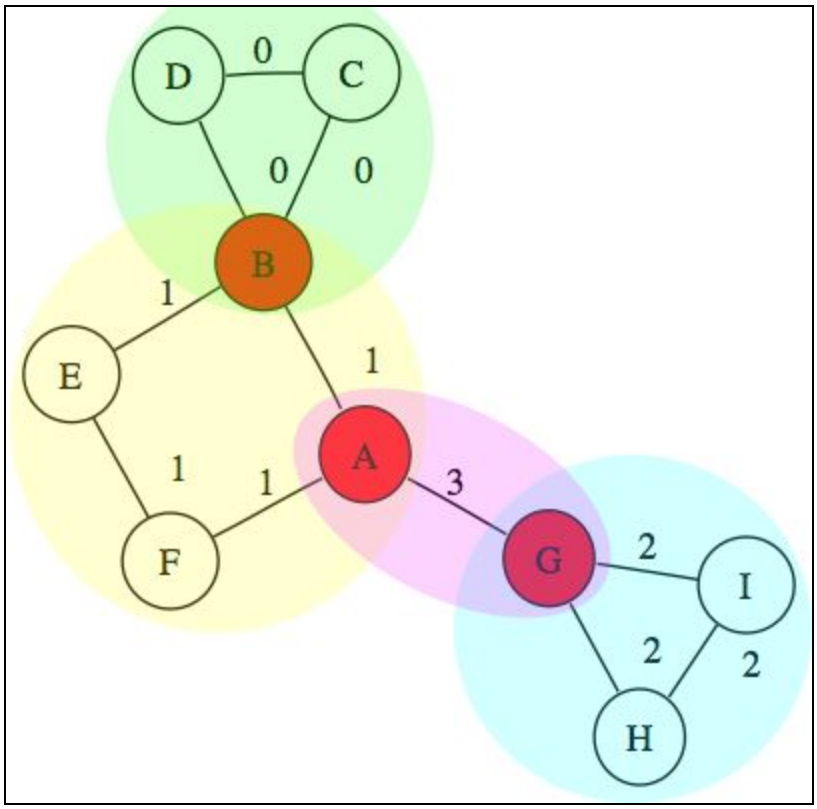
Descripción del problema y solución

Desde el punto de vista del usuario, el problema consiste en, al tener una red de agua con cierta cantidad de “puntos de suministro”, todos estos conectados entre sí. Se desea conocer aquellos “puntos de suministro” que al ser cerrados o quitados de la red por algún tipo de falla, dejen a la red inconexa de alguna manera, ya sea aislando a uno o más puntos de suministro.

Desde el punto de vista de la teoría de grafos, indicaremos que los “puntos de suministro” corresponden a las vértices de nuestro grafo. Este grafo será un grafo no dirigido, por lo tanto la relación entre sus vértices es bidireccional.

El problema en cuestión que se plantea desde el punto de vista del usuario, ahora orientado a la teoría de grafo es conocido como “punto crítico” o “punto de articulación”. Un punto de articulación es un vértice dentro de un grafo que al ser quitado o desconectado deja a uno o más vértices desconectados del resto. provocando dos o más sub-grafos pertenecientes al grafo original.

Imagen de ejemplo: los vértices rojos son “puntos críticos” A, B y G.



Para la solución a este problema se ha decidido emplear como método de recorrido para el caso de estudio el “recorrido en profundidad” junto con una serie de arreglos que guardarán, los vértices visitados, la cantidad de visitas por vértice, los valores más bajos y la cantidad de hijos por vértice dentro del grafo.

Para determinar exactamente si un vértice es o no un “punto crítico” primero debemos saber si fue visitado, si no fue visitado, debe cumplir alguna de las siguientes condiciones:

- el vértice es raíz dentro del árbol que se forma al recorrer(recorrido en profundidad) y tiene dos o más vértices hijos.
- si no es raíz, entonces el valor más bajo que se encuentre debe ser menor que la cantidad de veces descubierta que su destino.

Dicho esto se debe recorrer por cada vértice y revisar si este fue visitado, ver si tiene hijos que visitar y corroborar esta información. Si la tiene será guardado en otro arreglo de vértices para poder ser mostrado una vez terminada la ejecución. La solución del problema es de carácter recursivo por lo que se ha enmascarado en otra función para

mantener constancia de todos los campos anteriormente mencionados.

A continuación se mostrarán el código correspondiente, ya escrito en el programa:

```
1.  void criticoProfundidad(int u, int *visitados, int *descubiertos, int *padre, int *pc,
    int *menor, grafo *g, int *tiempo){
2.
3.      nodo *adj = g->array[u].cabeza;
4.
5.      int hijos = 0;
6.      visitados[u] = 1;
7.      descubiertos[u] = *tiempo;
8.      menor[u] = *tiempo;
9.      *tiempo+=1;
10.
11.     while(adj){
12.         int v = adj->destino;
13.         if(!visitados[v]){
14.
15.             padre[v] = u;
16.             hijos+=1;
17.             criticoProfundidad(v, visitados, descubiertos, padre, pc, menor, g, tiempo);
18.
19.             menor[u] = min(menor[u], menor[v]);
20.
21.                  if((padre[u] == -1 && hijos > 1) || (padre[u] != -1 && menor[v] >=
    descubiertos[u])){
22.                 pc[u] = 1;
23.             }
24.         }
25.         else{
26.             if(v != padre[u]){
27.                 menor[u] = min(menor[u], descubiertos[v]);
28.             }
29.         }
30.         adj = adj->siguiente;
31.     }
32. }
```

Esto se ha enmascarado con la siguiente:

```
1. void puntoCritico(grafo *g){
2. 
3.     // se inicializan todos los elementos a usar
4. 
5.     int tiempo, *visitados, *descubiertos, *menor, *padre, *pc;
6.     tiempo = 0;
7.     visitados = (int*)malloc(g->v * sizeof(int));
8.     descubiertos = (int*)malloc(g->v * sizeof(int));
9.     menor = (int*)malloc(g->v * sizeof(int));
10.    padre = (int*)malloc(g->v * sizeof(int));
11.    pc = (int*)malloc(g->v * sizeof(int));
12. 
13.    // se llenan para limpiarlos de datos basura
14.    for(int i = 0; i < g->v; i++){
15.        padre[i] = -1;
16.        visitados[i] = 0;
17.        pc[i] = 0;
18.    }
19. 
20.    for(int j = 0;j < g->v;j++){
21.        if(visitados[j] == 0){
22.              criticoProfundidad(j, visitados, descubiertos, padre, pc, menor, g,
    &tiempo);
23.        }
24.    }
25. 
26.    for(int k = 0; k< g->v ; k++){
27.        if(pc[k]){
28.            printf("nodo critico: %d \n", k + 1);
29.        }
30.    }
31.    // liberacion de memoria
32.    free(visitados);
33.    free(descubiertos);
34.    free(menor);
35.    free(padre);
36.    free(pc);
37. }
```

ANÁLISIS DE LOS RESULTADOS

Una vez completado lo anterior, ya podemos indicar que nuestros TDAs y algoritmos son capaces de resolver el problema presentado por la compañía de agua. Otros casos fueron modelados y se encuentran comentados en el código fuente del programa, esto con el fin de poder probar distintos escenarios y destacar que la solución no resuelve un problema en particular, sino que se puede generalizar a cualquier grafo, siguiendo los TDA descritos anteriormente. De igual manera profundizaremos en el análisis de la complejidad algorítmica de ambas funciones.

Análisis de complejidad algorítmica

Para el caso de la función “criticoProfundidad”, que es la función que se encarga de realizar el recorrido en profundidad en todos los vértices y sus vértices adyacentes tenemos lo siguiente:

$$T(n) = 9 + n(v1 + v2 + v3 + ... + vn)$$

$$O = n$$

donde “n” es el número de incidencias vértices y “v” es la cantidad de incidencias por incidencia de vértice, además de valores de unidad por acceso a elementos utilizados para marcar visitas, hijos y otros, finalmente podemos decir que “n” tiende al número de adyacencia por vértice.

Para el caso de la función “puntoCritico”, que es la función que enmascara el recorrido en profundidad e integra los marcadores ocupados, se tiene:

$$T(n) = 6 + n(5) + n(adj1 + adj2 + ... adjv)$$

$$O = n$$

donde “n” tiende a “v”, donde “v” que es el número de vértices de adyacencia. finalmente para la solución general podemos indicar que el proceso completo se puede escribir como

$$O = N + M$$

Donde “N” tiende al número de vértices y “M” al número de hijos por vértice.

Consideraciones

Se decidió dejar esta representación del grafo puesto que el costo en memoria y de recorrido es mucho mayor siendo una matriz de adyacencia que una lista lineal simple, en la medida en la que el número de vértices crece.

Ahora bien, dependiendo de el cómo se recorra la representación es finalmente lo que más incide, un punto de mejora podría ser buscar algún otro algoritmo de recorrido más eficiente para este tipo de casos, puesto que recorrer en profundidad es fue la primera respuesta lógica según lo estudiado en clase.

CONCLUSIONES

Podemos finalizar este informe concretando que, en base al análisis presentado de los resultados, concretamos todos los objetivos específicos planteados al comienzo de todo.

Hemos podido modelar de manera correcta, la representación de red de aguas pedidas por la empresa, crear un TDA asociado al modelo pedido, con funciones específicas para este modelo, detectar de buena manera con soluciones conocidas, los puntos de la red de agua que al ser quitados hacen que esta quede disconexa.

Si bien no se indican más formas de solucionar el problema, se deja a criterio del lector de este informe indagar si es necesario ocupar algún otro método o tomar algún otro criterio que no fuese indicado para modificar las funciones indicadas y buscar una solución más acorde al problema de la empresa.