ALGORITMOS Y ESTRUCTURA DE DATOS I

TEMA 4 GRAFOS

NOMBRE: ABELWANCHENG JIANGWAN

GRUPO: 1.1

PROFESOR: EDUARDO TIGERAS GIL

Contenido

1. LISTADO DE PROBLEMAS RESUELTOS 2
2. RESOLUCIÓN DE PROBLEMAS 2
2.1 ANALISÍS, DISEÑO Y EFICIENCIA 2
2.2 LISTADO DE CODIGO 4
3. CONCLUSIONES 10

1. LISTADO DE PROBLEMAS RESUELTOS

EJERCICIO	NUMERO
401	877
402	3464
403	3475
411	3480

He decidido aplicar un comodín para este tema 4 de grafos.

2. RESOLUCIÓN DE PROBLEMAS

2.1 ANALISÍS, DISEÑO Y EFICIENCIA

401.cpp

Solucionado en el ejercicio.

402.cpp

Análisis: He decidido implementar un grafo no dirigido que empieza por la letra 'A' y así siguiendo un orden alfabético 'A-Z' como indica el enunciado.

Diseño: He utilizado una matriz de Adyacencia donde hay de 0 hasta 25 números que son el equivalente a las letras del abecedario sin contar la ñ. Para el procedimiento recursivo de búsqueda primero en anchura he aplicado una cola para gestionar los niveles, primero encolamos el primer nodo y lo marcamos como visitado mientras que la cola no este vacía imprime y revisa sus adyacente por último los que no están visitados se encolan.

Eficiencia: como usamos una matriz G el orden de complejidad seria de O(n^2) en cambio los arrays de visitados o cola solo requieren el orden de complejidad de O(n). Por otra parte también tenemos en cuenta que el máximos de nodos que vamos a usar es de 26 que es la cantidad de letras que hay en el abecedario español por lo que esto puede ser insignificante para las maquinas actuales que usamos.

Problemas: Siendo sincero debido a que era el primer problema que me enfrentaba no sabía cómo empezarlo así que decidí ayudarme de Gemini aunque el programa funcionaba correctamente y compilaba bien no entendía algunas funciones que aplicaba ni comprendía muy bien el código así que al final decidí hacerlo por mi cuenta en base a los conocimientos que hemos dado en la clase de teoría.

403.cpp

Análisis: Nos dice que tenemos que encontrar desde la entrada hasta la salida del laberinto por otra parte también nos dice que tenemos que retroceder en el caso de que haya un callejo sin salida o las salas ya estén visitadas.

Diseño: Representamos una Lista de Adyacencia donde el número de nodos nos puede ser mayor a 20005 (no es 20000 ya que a la hora de ejecutar el programa me daba error de segmentación asique probe dejando un pequeño margen para este nodo). Donde he implementado una matriz G y un vector para conocer el número de vecinos de cada sala. Por otra parte también he aplicado el algoritmo de Búsqueda Primero en Profundidad con recursividad donde tenemos la variable global éxito para terminar con la recursión, para gestionar el camino si tras la llamada recursiva no ha habido éxito entonces tenemos que retroceder al nodo 'u' como se muestra en el código a esto también se le puede denominar backtracking que se da en la asignatura de Algoritmos y estructuras de datos II.

Eficiencia: En cuanto a la eficiencia el orden de complejidad seria de O(N + A) = O(20005+10) donde N es el número de nodos y A es el número de aristas.

Problemas: a la hora de realizar la función leeGrafo en un principio lo hice con enteros pero me daba problemas ya que no me leía los saltos de línea así que para poder realizarlo y que me lea también los saltos de línea lo he hecho directamente convirtiendo los caracteres a enteros.

411.cpp

Análisis: El objetivo de este problema es identificar cuantas islas existen y a que isla pertenece cada persona. Donde las personas se conocen ya sea directa o indirectamente. He decidido aplicar un grafo no dirigido.

Diseño: Utilizamos una lista de adyacencia como en el ejercicio anterior por otra parte como es no dirigido entonces se deberá de leer tanto de (v, u) = (u, v). He incorporado un algoritmo de búsqueda de profundidad donde cuando iniciamos la búsqueda con el recorrido for, marcamos los miembros que pertenecen a la isla en caso de que no sea visitado hacemos una llamada recursiva. Por ultimo para resolver e imprimir este ejercicio he declarado la función resolverCaso donde declaramos el contador de islas y recorremos todas las personas de 1…N si no hay visitado entonces incrementamos el contador de islas y llamamos la función de búsqueda de profundidad luego imprimir el caso y la isla que pertenece cada persona.

Eficiencia: El orden de complejidad es de O(N+ M) dependiendo de número de personas y el numero pares de personas que se conocen entre sí.

Problemas: No he tenido demasiado lo único ha sido que no me daba la solución correcta y era debido al main donde se estaba inicializando a 0 en vez de 1.

2.2 LISTADO DE CODIGO

401.cpp

```
#include <stdlib.h>  // Funcion exit
#include <string.h>  // Funcion memset
#include <iostream>  // Variables cin y cout
using namespace std;

#define MAX_NODOS 26

////////////////////////////////////////////////////////////
////////////        VARIABLES GLOBALES        ///////////////
////////////////////////////////////////////////////////////

int nnodos;                   // Numero de nodos del grafo
int naristas;                 // Numero de aristas del grafo
bool G[MAX_NODOS][MAX_NODOS]; // Matriz de adyacencia
bool visitado[MAX_NODOS];     // Marcas de nodos visitados

////////////////////////////////////////////////////////////
////////////     FUNCIONES DEL PROGRAMA       ///////////////
////////////////////////////////////////////////////////////

void leeGrafo (void)
// Procedimiento para leer un grafo de la entrada
{
  cin >> nnodos >> naristas; // Lectura numero de nodos y aristas
  if (nnodos<0 || nnodos>MAX_NODOS) { // Comprobación validez
    cerr << "Numero de nodos (" << nnodos << ") no valido\n";
    exit(0);
  }
  memset(G, 0, sizeof(G)); // inicialización matriz de adyacencia a 0
  char c1, c2;
  for (int i= 0; i<naristas; i++) { // Lectura de aristas
    cin >> c1 >> c2;
    G[c1-'A'][c2-'A']= true; // actualización matriz
  }
}

void bpp(int v)
// Procedimiento recursivo de la búsqueda primero en profundidad
//   v - primer nodo visitado en la bpp
{
  visitado[v]= true;
```

```
  cout << char(v+'A');
  for (int w= 0; w<nnodos; w++)
    if (!visitado[w] && G[v][w])
      bpp(w);
}

void busquedaPP (void)
// Procedimiento principal de la búsqueda en profundidad
{
  memset(visitado, 0, sizeof(visitado));
  for (int v= 0; v<nnodos; v++)
    if (!visitado[v])
      bpp(v);
  cout << endl;
}

////////////////////////////////////////////////////////////
////////////        PROGRAMA PRINCIPAL        ///////////////
////////////////////////////////////////////////////////////

int main (void)
{
  int ncasos;
  cin >> ncasos;
  for (int i= 0; i<ncasos; i++) {
    leeGrafo();
    busquedaPP();
  }
}
```

402.cpp

```
#include <stdlib.h>  // Función exit
#include <string.h>  // Función memset
#include <iostream>  // Variables cin y cout
using namespace std;

#define MAX_NODOS 26

////////////////////////////////////////////////////////////
////////////        VARIABLES GLOBALES        ///////////////
////////////////////////////////////////////////////////////

int nnodos;                   // Numero de nodos del grafo
int naristas;                 // Numero de aristas del grafo
bool G[MAX_NODOS][MAX_NODOS]; // Matriz de adyacencia
bool visitado[MAX_NODOS];     // Marcas de nodos visitados
```

```
////////////////////////////////////////////////////////////
////////////     FUNCIONES DEL PROGRAMA       ///////////////
////////////////////////////////////////////////////////////

void leeGrafo (void)
// Procedimiento para leer un grafo de la entrada
{
  cin >> nnodos >> naristas; // Lectura número de nodos y aristas
  if (nnodos<0 || nnodos>MAX_NODOS) { // Comprobación validez
    cerr << "Numero de nodos (" << nnodos << ") no valido\n";
    exit(0);
  }
  memset(G, 0, sizeof(G)); // inicialización matriz de adyacencia a 0
  char c1, c2;
  for (int i= 0; i<naristas; i++) { // Lectura de aristas
    cin >> c1 >> c2;
    G[c1-'A'][c2-'A']= true; // actualización matriz
  }
}


// Procedimiento recursivo de la búsqueda primero en anchura
// primero visitamos la v
// luego visitamos sus adyacentes
// luego los adyacentes de los adyacentes

void bpa(int v) {
    // Inicializacion cola
    int cola[MAX_NODOS];
    int frente = 0;
    int final = 0;
    // Encolar primer nodo
    visitado[v] = true;
    cola[final] = v;
    final++;

    while (frente < final) { // Mientras NOT EsVaciaCola
        int v = cola[frente]; // FrenteCola
        frente++;
        cout << char(v + 'A');

        for (int w = 0; w < nnodos; w++) { // Iterador sobre adyacentes
            if (!visitado[w] && G[v][w]){
                visitado[w] = true; // Marcar nodo como visitado
                cola[final] = w; // Encolar
                final++; // Incrementar final
            }
        }
```

```
    }
}

void busquedaPA() {
    memset(visitado, 0, sizeof(visitado)); // BorraMarcas
    for (int v = 0; v < nnodos; v++) {
        if (!visitado[v]) {
            bpa(v); // Llamada a bpa
        }
    }
    cout << endl;
}

////////////////////////////////////////////////////////////
////////////        PROGRAMA PRINCIPAL        ///////////////
////////////////////////////////////////////////////////////

int main (void) {
  int ncasos;
  cin >> ncasos;
  for (int i= 0; i<ncasos; i++) {
    leeGrafo();
    busquedaPA();
  }
}
```

403.cpp

```
#include <iostream>
#include <string.h>  // Función memset
#include <stdlib.h>  // Función exit

using namespace std;

// --- CONSTANTES Y GLOBALES ---
#define MAX_NODOS 20005    // máximo número de nodos
#define MAX_ADY 10         // máximo 10 adyacentes
#define MAX_CAMINO 200000  // Espacio suficiente para el recorrido con
retrocesos

int nnodos;                // Numero de nodos del grafo
int G[MAX_NODOS][MAX_ADY]; // Matriz de adyacencia
int grado[MAX_NODOS];      // El número de vecinos de cada sala

bool visitado[MAX_NODOS];  // Marcas de nodos visitados
int camino[MAX_CAMINO];    // Camino actual (con retrocesos)
int long_camino;           // Longitud del camino actual
bool exito;                // Indicador de éxito en la búsqueda
```

```
// --- FUNCIONES DEL PROGRAMA ---

// construimos la lista de adyacencia transformando los caracteres a
enteros
// separados por un salto do línea

void leeGrafo (void) {
  cin >> nnodos;
  memset(grado, 0, sizeof(grado));
  // leemos el salto de línea de numero de nodos ya que este número no me
sirve
  char c;
  cin.get(c); // Leer el salto de línea después del número de nodos
  for (int i = 1; i <= nnodos; i++) {
    cin.get(c); // leemos el primer carácter de la línea
    while (c != '\n') { // mientras no sea salto de línea
      if (c >= '0' && c <= '9') { // si es un digito
        int valor = 0;
        while (c >= '0' && c <= '9') {
          valor = valor * 10 + (c - '0'); // convertimos el carácter a
entero cogemos el valor ASCII y restamos el de '0'
          cin.get(c);
      }
          G[i][grado[i]] = valor;
            grado[i]++;
      } else { // en el caso de que no sea digito entonces leemos el
siguiente carácter que será un espacio o salto de línea
          cin.get(c);
          }
      }
  }
}


// Procedimiento recursivo de la búsqueda en profundidad con retrocesos
// depth first search
void dfs(int u) {
  camino[long_camino] = u;
  long_camino++;
  if (u == nnodos) {
    exito = true; // exitoso
    return; // terminar la búsqueda
  }
  visitado[u] = true; // marcar como visitado
  // comprobamos todos los posibles adyacentes empezando desde el nodo u
  for (int i = 0; i < grado[u]; i++) { // iterar sobre adyacentes
    int v = G[u][i];  // nodo adyacente
```

```
    if (!visitado[v]) { // si no ha sido visitado
      dfs(v); // llamada recursiva para visitar v
      if (exito) {
        return; // si se encontró el camino, terminamos
      } else { // si no se encontró el camino, retrocedemos
        camino[long_camino] = u; // retroceder al nodo u
        long_camino++; // incrementar longitud del camino
    }
  }
}
}

// invocamos el algoritmo de búsqueda en profundidad y luego
// imprimimos las salidas

void resolverCaso(int nCaso) {
  long_camino = 0; // Longitud del camino actual
  exito = false; // Indicador de éxito en la búsqueda
  memset(visitado, 0, sizeof(visitado)); // inicialización marcas de
visitado

  if (nnodos > 0) { // Si hay nodos en el grafo
    dfs(1); // Iniciar DFS desde el nodo 1 como indica en el enunciado
  }

  cout << "Caso " << nCaso << endl;
  if (exito) {  // Si se ha encontrado un camino
    cout << long_camino << endl;
    for (int i = 0; i < long_camino; i++) {
      cout << camino[i] << endl; // Imprimir cada nodo del camino
    }
  } else {
    cout << "INFINITO" << endl;  // No se ha encontrado un camino
  }
}


int main (void)
{
  int ncasos;
  cin >> ncasos;
  for (int i= 0; i<ncasos; i++) {
    leeGrafo();
    resolverCaso(i+1);
  }
}
```

411.cpp

```cpp
// we are islands
# include <iostream>
# include <string.h>  // función memset

using namespace std;

// --- CONSTANTES Y GLOBALES ---
#define MAX_NODOS 20000   // máximo número de nodos lo suficiente para
que quepa toda la entrada de mooshak
#define MAX_ADY 10      // máximo 10 adyacentes

int N,M;
int G[MAX_NODOS][MAX_ADY]; // Matriz de adyacencia
int grado[MAX_NODOS];      // Grado de cada nodo número de amigos

int isla[MAX_NODOS]; // para saber a qué isla pertenece cada nodo
bool visitado[MAX_NODOS];  // Marcas de nodos visitados

// lectura del grafo

/*
leemos el número de personas y sus relaciones para construir el mapa
usando listas de adyacencia
una vez leídas guardamos y limpiamos gracias al memset
*/
void leeGrafo (void) {
    cin >> N >> M; // Lectura número de nodos y aristas
    // limpiamos las memorias para los nuevos casos
    memset(grado, 0, sizeof(grado));
    memset(visitado, 0, sizeof(visitado));
    memset(isla, 0, sizeof(isla));

    for (int i=0; i<M ; i++) {
        int u,v;
        cin >> u >> v;

        G[u][grado[u]] = v;
        grado[u]++;
        G[v][grado[v]] = u;
        grado[v]++;
    }
}
```

```
// vamos a usar el algoritmo búsqueda en profundidad
// y marcamos su isla

void bpp (int u, int num_isla) {
    visitado[u] = true;
    isla[u] = num_isla;
    // recorremos todos los vecinos del nodo u
    for (int i=0; i<grado[u]; i++) {// recorremos todos los vecinos de u
        int v = G[u][i];
        // si no ha sido visitado continuamos la búsqueda recursiva
        if (!visitado[v]) {
            bpp(v, num_isla);
        }
    }
}


// resolvemos
// recorremos las personas cada vez que encuentra a alguien que no es
visitado
// identifica una nueva isla iniciando un recorrido al final imprimimos
el número de casos
// y la isla asignada a cada persona
void resolverCaso(int nCaso) {
    int contador_islas = 0;
    // recorremos las personas desde 1 hasta N
    for (int i=1; i<=N; i++) {
        if (!visitado[i]) {
            contador_islas++;
            bpp(i, contador_islas); // llamada a bpp
        }
    }
    //imprimimos la salida de datos como indica el enunciado
    cout << "Caso "<< nCaso << endl;
    cout << contador_islas << endl;

    // imprimimos a que isla pertenece a la persona
    for (int i=1; i<=N; i++) {
        cout << isla[i] << endl;
    }
}

int main (void)
{
  int ncasos;
  cin >> ncasos;
  for (int i= 0; i<ncasos; i++) {
    leeGrafo();
```

```
    resolverCaso(i+1);
  }
}
```

3. CONCLUSIÓN

En resumen realizar este proyecto me ha ayudado a afianzar un poco más los conocimientos acerca sobre los grafos también a usar algunas funciones de librerías que desconocía y por último las clases de teoría y las diapositivas han sido una gran ayuda para

Por otra parte también en el uso de la IA está bien para realizar preguntas concretas sobre algoritmos u otro tipo de cosas concretas. Al final he llegado a la conclusión de que tardas mucho menos en comprender el tema y aplicar lo que se debe de aplicar antes que comprender todos los nuevos conceptos que aplica la IA y nunca lo he implementado en el código.

Por otra parte el tiempo que he estado realizando el proyecto rondara entre las 15 a las 20 horas.