GRAFOS

AED I, ACTIVIDAD CAPÍTULO 4

Jose Antonio Lorencio Abril
Grupo PCEO
Profs: Norberto Marín Pérez y Ginés García Mateos
Usuario D18

Listado problemas resueltos

- Problema 402, envío 88.
- Problema 403, envío 121.
- Problema 404, envío 1083.
- Problema 405, envío 476.
- Problema 407, envío 513.
- Problema 411, envío 112.
- Problema 412, envío 1364.
- Problema 414, envío 2178.
- Problema 415, envío 1490.

Resolución de problemas

Paso aquí a describir la resolución de los problemas listados anteriormente. A la hora de analizar la eficiencia de cada problema, voy a obviar la parte de creación del grafo, pues en cualquier caso se tiene que, dado un grafo de n nodos y m aristas, de una u otra forma voy a realizar, a lo más n^2 operaciones cuando está implementado mediante una matriz, o n+m cuando lo está mediante listas.

402

Este era un problema muy sencillo, pues su resolución era la implementación del algoritmo de búsqueda en anchura explicado en teoría, cuyo pseudocódigo se encuentra en los apuntes.

Al haberlo implementado usando matrices, la eficiencia es de $O(n^2)$.
Se podría haber conseguido mejor eficiencia usando listas, pero era aún un novato al momento de resolver esta cuestión.

```
#include <stdlib.h>  // Funcion exit
#include <string.h>  // Funcion memset
#include <iostream>  // Variables cin y cout
#include <queue>     // Uso de colas
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;
  if (nnodos<0 || nnodos>MAX_NODOS) {
    cerr << "Numero de nodos (" << nnodos << ") no valido\n";
    exit(0);
  }
  memset(G, 0, sizeof(G));
  char c1, c2;
  for (int i= 0; i<naristas; i++) {
    cin >> c1 >> c2;
    G[c1-'A'][c2-'A']= true;
  }
}

void bpa(int v)
// Procedimiento iterativo de la busqueda primero en anchura
//   v - primer nodo visitado en la bpp
{
  queue<int> q;
  int x;
  visitado[v]= true;
  cout << char(v+'A');
  q.push(v);
  while(!q.empty()){
    x=q.front();
    q.pop();
    for(int y=0; y<nnodos; y++){
        if((G[x][y])&&(!visitado[y])){
            cout << char(y+'A');
            visitado[y]=true;
            q.push(y);
        }
    }
  }
}
```

```
void busquedaPA (void)
// Procedimiento principal de la busqueda en anchura
{
  memset(visitado, 0, sizeof(visitado));
  for (int v= 0; v<nnodos; v++)
    if (!visitado[v])
      bpa(v);
  cout << endl;
}

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

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

403

Idea: realizar una búsqueda en anchura únicamente a partir del nodo 0. Si llegamos al nodo n, hemos escapado del laberinto, si no, nunca saldremos.

Para tener en cuenta el orden en que se habían de recorrer los nodos (de derecha a izquierda), implementé el grafo con listas de adyacencia, de tal forma que el primer elemento de la lista, correspondía con el nodo de más a la derecha, y el último con el de más a la izquierda.
De este modo, lanzando una búsqueda en anchura en el nodo 0, y almacenando los nodos por los que paso en una cola, obtengo que, al acabar el recorrido en anchura:

a) Nodo n **sí** ha sido visitado. En este caso Pablo se las apañó bien para escapar del laberinto, y el recorrido que siguió viene descrito por los elementos de la cola.
b) Nodo n **no** ha sido visitado. Pablo no fue tan afortunado, y vivirá por tiempo "infinito" dentro del laberinto.

La eficiencia, de este modo es O(n+m), siendo n el número de nodos, y m el de aristas.

```
#include <stdlib.h>  // Funcion exit
#include <string.h>  // Funcion memset
#include <iostream>  // Variables cin y cout
#include <sstream>  // Para poder leer líneas de enteros
#include <queue>     // Para implementar el recorrido
#include <list>     // Para representar el grafo
using namespace std;

#define MAX_NODOS 20000

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

int nnodos;                   // Numero de nodos del grafo
list<int> *G;                 // Array listas de adyacencia.
bool visitado[MAX_NODOS];     // Marcas de nodos visitados
queue<int> q;                 // cola para recorrer en profundidad
y guardar los nodos en orden, por si no fuese conexo.

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

```
////////////////////////////////////////////////////////////////

void bpp(int v)
// Procedimiento recursivo de la busqueda primero en profundidad
//   v - primer nodo visitado en la bpp
// Se apoya en colas pues solo imprimimos los nodos si el grafo
conecta 1 con N
{
  visitado[v]= true;
  q.push(v+1);
  if(v+1==nnodos) return;
  for(int w : G[v]){
    if(!visitado[w-1] && !visitado[nnodos-1]){
      bpp(w-1);
      if(!visitado[nnodos-1]) q.push(v+1);
    }
  }
}


void busquedaPP (void)
// Hago búsqueda en profundidad solo en el primer elemento, para
saber si conecta con el último
{
  memset(visitado, 0, sizeof(visitado));
  bpp(0);
  if (visitado[nnodos-1]) {
    cout << q.size() << endl;
    while(!q.empty()){
      int x=q.front();
      q.pop();
      cout << x << endl;
    }
  }
  else{
    cout << "INFINITO" << endl;
    while(!q.empty()) q.pop();
  }
}
```

```
void leeGrafo (void)
// Procedimiento para leer un grafo de la entrada
{
  cin >> nnodos;
  cin.ignore();
  if (nnodos<0 || nnodos>MAX_NODOS) {
    cerr << "Numero de nodos (" << nnodos << ") no valido\n";
    exit(0);
  }
  delete[] G;
  G = new list<int>[nnodos];
  int c;
  for (int i=0; i<nnodos; i++){
      string linea;
      getline(cin, linea);
      istringstream iss (linea);
      while(iss >> c){
        G[i].push_back(c);
      }
  }
}

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

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

404

Idea: aplicar dos veces Dijkstra, una entre el inicio y el nodo de paso, y otra entre el nodo de paso y el nodo de llegada.

Representé el grafo como listas de pares de enteros, siendo par.first el nodo conectado, y par.second el peso de la arista.
Implementé Dijsktra usando una cola por prioridad, ordenada en orden creciente, de tal forma que encontrar el nodo más cercano al actual era trivial.
Además, hice que Dijsktra devolviese la longitud del camino mínimo entre dos nodos pasados como parámetro. De esta forma, ejecutando Dijsktra(inicio, paso) y Dijkstra(paso, llegada) obtenía las dos distancias necesarias para la resolución del problema.
Aunque, eso sí, teniendo en cuenta que si no puede llegarse del nodo de inicio al de paso, será "imposible" realizar el camino pedido, como también lo será si no puede alcanzarse el nodo de llegada desde el de paso.
En retrospectiva, y cuando el tiempo es escaso, uno se da cuenta de sus errores, como yo me he dado cuenta de que alargo de más la búsqueda, siguiendo el algoritmo una vez que ya he encontrado la distancia mínima entre los dos nodos dados.
La eficiencia queda del siguiente modo:

Aunque el uso de la cola con prioridad pueda haga que el tiempo de elección del siguiente nodo sea O(1), debemos tener en cuenta el tiempo de inserción en la cola por prioridad, que viene a ser el de una lista ordenada, o sea, O(n).
De este modo, el algoritmo de Dijkstra, como ya vimos en teoría, implementado con listas de adyacencia, sigue un orden $O(n^2)$, si bien esto puede mejorarse, como me doy cuenta más tarde, en algún problema posterior.
Aunque ejecutamos Dijsktra dos veces, sabemos que las constantes multiplicativas no afectan al orden de complejidad.

```
#include <stdlib.h>  // Funcion exit
#include <iostream>  // Variables cin y cout
#include <queue>     // Para implementar el recorrido
#include <list>     // Para representar el grafo
#include <vector>   // Para representar el grafo
using namespace std;

#define MAX_NODOS 20000
#define MAX_DIST 10000

////////////////////////////////////////////////////////////
```

```
////////////        VARIABLES GLOBALES        ////////////////
//////////////////////////////////////////////////////////////

int nnodos;                         // Número de nodos del grafo
int naristas;                       // Número de aristas
int inicio;                         // Nodo de inicio
int paso;                           // Nodo de paso
int llegada;                        // Nodo de llegada
list<pair<int,int>> *G;             // Array de listas de adyacencia

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

void leeGrafo (void)
// Procedimiento para leer un grafo de la entrada
{
  cin >> nnodos;
  cin >> naristas;
  cin >> inicio;
  cin >> llegada;
  cin >> paso;
  if (nnodos<0 || nnodos>MAX_NODOS) {
    cerr << "Numero de nodos (" << nnodos << ") no valido\n";
    exit(0);
  }
  G = new list<pair<int,int>>[nnodos]();

  for(int i=0; i<naristas; i++){
    int u, v, c;
    cin >> u >> v >> c;
    G[u-1].push_back({v-1,c});
    G[v-1].push_back({u-1,c});
  }
}
```

```
int minDist(int na, int nb){
  for(pair<int, int> p : G[na]){
      if(p.first==nb){
          return p.second;
      }
  }
  return MAX_DIST;
}

int Dijkstra(int inicio, vector<bool> &escogidos, int fin)
{
  priority_queue<pair<int,int>,vector<pair<int,int> >,
greater<pair<int,int> > > cola;
  vector<int> D(nnodos, MAX_DIST);
  D[inicio]=0;
  cola.push({0, inicio});
  while(!cola.empty()){
    int v = cola.top().second;
    cola.pop();
    escogidos[v]=true;
    for(int i=0; i<nnodos; i++){
      if(!escogidos[i]){
        int maybe=minDist(v, i);
        if(maybe+D[v]<D[i]) {
            D[i]=D[v]+maybe;
            cola.push({D[i], i});
        }
      }
    }
  }
  return D[fin];
}
```

```
void Dijk(){
  vector<bool> escogidos(nnodos,false);
  int dist=Dijkstra(inicio-1, escogidos, paso-1);
  if(!escogidos[paso-1]) cout << "IMPOSIBLE" << endl;
  else {
      vector<bool> escogidos2(nnodos, false);
      dist+=Dijkstra(paso-1, escogidos2, llegada-1);
      if(!escogidos2[llegada-1]) cout << "IMPOSIBLE" << endl;
      else{
          cout << dist << endl;
      }
  }
}

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

int main (void)
{
  int ncasos;
  cin >> ncasos;
  for (int i= 0; i<ncasos;i++) {
    leeGrafo();
    Dijk();
    delete[] G;
  }
}
```

405

Idea: de nuevo, aplicar Dijsktra.

En este caso, solo había que aplicar Dijsktra, teniendo precaución de hacer una función que calculase de forma correcta los pesos.
Función que describo a continuación:

entero minDist(**entero** na, **entero** nb, **entero** actual) *//na=nodo a, nb=nodo b, actual=peso del camino total recorrido hasta el momento*
{
	entero devolver=infinito;
	entero centenas=actual/100;
	actual%=100;
	para entero m : momentos del día en que se encuentran a y b **hacer**
		si m>=actual **YDESPUÉS** m<devolver **entonces**
			devolver=m;
		sino si (devolver==infinito **YDESPUÉS** m<actual) **ó** (devolver!=infinito **YDESPUÉS** devolver%100<actual **YDESPUÉS** m<devolver%100) **entonces**
			devolver=100+m;
		finsi
	finpara
	si devolver!=infinito **entonces** devolver=100*centenas+devolver;
return devolver;
}

Así, la complejidad es la de un Dijsktra normal ($O(n^2)$), añadiéndole la complejidad del cálculo del peso en la actualización de la matriz de costes. Este coste es, dado un par de nodos, que se encuentran en k instantes, O(k).
Además, una vez hemos calculado los caminos mínimos, tengo que encontrar el mayor de ellos, con un coste de n.
De este modo, la complejidad será O(n(n+k)+n), siendo ahora k la media de los momentos en los que se encuentran dos nodos cualesquiera. Aunque este coste, para n suficientemente grande, deriva en $O(n^2)$.

```
#include <stdlib.h>  // Funcion exit
#include <string.h>  // Funcion memset
#include <iostream>  // Variables cin y cout
#include <sstream>   // Para poder leer líneas de enteros
#include <queue>     // Para implementar el recorrido
#include <list>      // Para representar el grafo
```

```
#include <vector>    // Para representar el grafo
using namespace std;

#define MAX_NODOS 20000
#define MAX_INSTANTE 10000
////////////////////////////////////////////////////////////
////////////        VARIABLES GLOBALES        ///////////////
////////////////////////////////////////////////////////////
int M;                          // Número de nodos del grafo
int K;                          // Número de pares de aristas
list<vector<int>> *G;           // Array de listas de adyacencia
////////////////////////////////////////////////////////////
////////////     FUNCIONES DEL PROGRAMA       ///////////////
////////////////////////////////////////////////////////////
void leeGrafo (void)
// Procedimiento para leer un grafo de la entrada
{
  cin >> M;
  cin >> K;
  cin.ignore();
  if (M<0 || M>MAX_NODOS) {
    cerr << "Numero de nodos (" << M << ") no valido\n";
    exit(0);
  }
  G = new list<vector<int>>[M]();

  for(int i=0; i<K; i++){
    int c1, c2, v;
    cin >> c1 >> c2 >> v;
    vector<int> vec(v+1);
    vec[0]=c2-1;
    vector<int> vecc(v+1);
    vecc[0]=c1-1;
    for(int j=1; j<=v; j++){
      int ins;
      cin >> ins;
      vec[j]=ins;
      vecc[j]=ins;
    }
    G[c1-1].push_back(vec);
    G[c2-1].push_back(vecc);
```

```
  }
}

int minDist(int na, int nb, int actual){
  int ret=MAX_INSTANTE;
  int centenas=actual/100;
  actual%=100;
  for(vector<int> x : G[na]){
    if(x[0]==nb){
      int size=x.size();
      for(int i=1; i<size; i++) {
        if(x[i] >= actual && x[i] < ret){
          ret=x[i];
        }
        else if(ret==MAX_INSTANTE && x[i] < actual ){
          ret=100+x[i];
        }
        else if(ret!=MAX_INSTANTE && ret%100 < actual &&
x[i]<ret%100){
          ret=100+x[i];
        }
      }
    }
  }
  if(ret!=MAX_INSTANTE) ret=100*centenas+ret;
  return ret;
}
```

```
void Dijkstra()
{
  vector<bool> escogidos(M,false);
  priority_queue<pair<int,int>,vector<pair<int,int> >,
greater<pair<int,int> > > cola;
  vector<int> D(M, MAX_INSTANTE);
  D[0]=0;
  cola.push({0, 0});
  while(!cola.empty()){
    int v = cola.top().second;
    cola.pop();
    escogidos[v]=true;
    for(int i=0; i<M; i++){
      int maybe=minDist(v, i, D[v]);
      if(maybe<D[i]) {
        D[i]=maybe;

        cola.push({D[i], i});
      }
    }
  }
  int max=0;
  for(int i=1; i<M;i++){
    if (!escogidos[i]){
      max=-1;
      break;
    }
    else {
      if (D[i] > max) max=D[i];
    }
  }
  cout << max << endl;
}

////////////////////////////////////////////////////////////
////////////        PROGRAMA PRINCIPAL        ///////////////
////////////////////////////////////////////////////////////
```

```
int main (void)
{
  int ncasos;
  cin >> ncasos;
  for (int i= 0; i<ncasos;i++) {
    leeGrafo();
    Dijkstra();
    delete[] G;
  }
}
```

407

Idea: usar el algoritmo de Floyd para calcular la distancia entre todo par de ciudades del mapa, y calcular la excentricidad de cada uno, quedándonos con el que posee menor valor de esta última.

En este caso volví a la representación con matrices, pues de esta forma el algoritmo es muy sencillo de implementar, tal como está recogido en los apuntes.
Así realicé simplemente el algoritmo de Floyd, y después calculé la capital del país (la ciudad con menor excentricidad) de la siguiente forma:

entero excentricidad=infinito;
para i=1,...,M **hacer**
    entero mayor=0;
    para j=1,...,M **hacer**
        si i!=j **YDESPUÉS** D[i,j]>mayor **entonces** mayor=D[i,j]; *//Siendo D la matriz de costes mínimos entre pares, tras ejecutar Floyd*
    finpara
    si mayor<excentricidad **ó** (mayor==excentricidad **YDESPUÉS** C[i]<C[capital]) **entonces** *//Siendo C la matriz que relaciona cada nombre de la ciudad con su número de nodo, y esta última condición debida a que en caso de excentricidades iguales, nos quedamos con aquella menor lexicográficamente*
        capital=i;
        excentricidad=mayor;
    finsi
finpara

De esta manera, obtenemos que el orden de complejidad del algoritmo es bastante elevado. El orden de este algoritmo para calcular la capital es $O(n^2)$.
Así, junto con el de Floyd, queda un orden $O(n^3 + n^2)$, si bien para n considerablemente grande, el n^3 se "*merienda*" al n^2.

```
#include <stdlib.h>  // Funcion exit
#include <string.h>  // Funcion memset
#include <iostream>  // Variables cin y cout
#include <sstream>   // Para poder leer líneas de enteros
#include <queue>     // Para implementar el recorrido
#include <list>      // Para representar el grafo
#include <vector>    // Para representar el grafo
using namespace std;
```

```
#define MAX_NODOS 200
#define inf 20000

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

int M;                          // Número de nodos del grafo
int K;                          // Número de pares de cotillas
string *C;                      // Array para guardar a qué número
corresponde cada ciudad
int D[MAX_NODOS][MAX_NODOS];  // Matriz de caminos mínimos
////////////////////////////////////////////////////////////
////////////     FUNCIONES DEL PROGRAMA       ///////////////
////////////////////////////////////////////////////////////

void leeGrafo (void)
// Procedimiento para leer un grafo de la entrada
{
  cin >> M;
  cin.ignore();
  if (M<0 || M>MAX_NODOS) {
    cerr << "Numero de nodos (" << M << ") no valido\n";
    exit(0);
  }

  C = new string[M];

  for(int i=0; i<M; i++){
    string city;
    getline(cin, city);
    C[i]=city;
  }

  for(int i=0; i<M; i++){
      for(int j=0; j<M; j++){
          D[i][j]=inf;
      }
  }

  int c1, c2, L;
```

```
  while(true){
      cin >> c1 >> c2 >> L;
      if(c1==0 && c2==0 && L==0) break;
      else {
          D[c1][c2]=D[c2][c1]=L;
      }
  }
}

int min(int a, int b){
    if(a<=b) return a;
    return b;
}

void Floyd()
{
  for(int k=0; k<M; k++){
      for(int i=0; i<M; i++){
          for(int j=0; j<M; j++){
              D[i][j]=min(D[i][j], D[i][k]+D[k][j]);
          }
      }
  }
  int capital=0;
  int excentricidad=inf;
  for(int i=0; i<M; i++){
      int mayor=0;
      for(int j=0; j<M; j++){
          if(i!=j && D[i][j]>mayor) mayor=D[i][j];
      }
      if(mayor < excentricidad || (mayor==excentricidad &&
C[i]<C[capital])){
          capital=i;
          excentricidad=mayor;
      }
  }
  cout << C[capital] << endl;
  cout << excentricidad << endl;
}
```

```
////////////////////////////////////////////////////////////
////////////        PROGRAMA PRINCIPAL        ///////////////
////////////////////////////////////////////////////////////

int main (void)
{
  int ncasos;
  cin >> ncasos;
  for (int i= 0; i<ncasos;i++) {
    leeGrafo();
    Floyd();
    delete[] C;
  }
}
```

411

Idea: usar la relación de equivalencia “ser conocido de”.

Apliqué lo aprendido en el capítulo 3 sobre relaciones de equivalencia, y resolví este problema usando la operación buscarRaiz, apoyada en la compresión de caminos.

De esta forma, lo único que había que hacer era, una vez almacenados todos los nodos, calcular el número de raíces, y saber en qué conjunto cociente se encuentra cada nodo, lo cual hacía del siguiente modo:

```
void busquedaIslas()
{
        entero nislas=0, isla=-1;
        para i=1,...,n hacer
                si i es raíz entonces
                        nislas++;
                        G[i]=isla; // esto quiere decir “i pertenece a la isla (-isla)
                        isla--;
                finsi
        finpara
        escribir(nislas);
        para i=1,...n hacer
                si i no es raíz entonces
                        imprimir(-G[ buscarRaiz(G[i]) ] );
                sino imprimir(-G[i]);
}

entero buscarRaiz(entero j)
{
        si G[j]<=0 entonces return j;
        G[j] = buscarRaiz(G[j]);
        return G[j];
}
```

De este modo, en base a lo estudiado sobre relaciones de equivalencia, el orden de complejidad sería O(n logn), pues debemos recorrer todos los nodos un par de veces (2n ~ n) y la operación buscarRaiz es de orden logn, como mucho, gracias a la compresión de caminos usada.

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

```
#include <iostream>  // Variables cin y cout
#include <queue>     // Uso de colas
using namespace std;

#define MAX_NODOS 20000

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

int nnodos;                   // Numero de nodos del grafo
int naristas;                 // Numero de aristas del grafo
int G[MAX_NODOS];              // Relación de equivalencia
mediante punteros
bool visitado[MAX_NODOS];     // Marcas de nodos visitados

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

int buscarRaiz(int j){
    if(G[j]<=0) return j;
    G[j] = buscarRaiz(G[j]);
    return G[j];
}

void leeGrafo (void)
// Procedimiento para leer un grafo de la entrada
{
  cin >> nnodos >> naristas;
  if (nnodos<0 || nnodos>MAX_NODOS) {
    cerr << "Numero de nodos (" << nnodos << ") no valido\n";
    exit(0);
  }
  memset(G, 0, sizeof(G));
  int i1, i2;
  for (int i= 0; i<naristas; i++) {
    cin >> i1 >> i2;
    int r1=buscarRaiz(i1);
    int r2=buscarRaiz(i2);
    if(r1 < r2){
```

```
        G[r2] = r1;
    }
    else if(r1==r2){
        G[i2]=r1;
    }
    else{
        G[r1] = r2;
    }
    }
}

void busquedaIslas(){
    int nislas=0;
    int isla=-1;
    for(int i=1; i<=nnodos; i++){
        if(G[i]==0) {
            nislas++;
            G[i] = isla;
            isla--;
        }
    }
    cout << nislas << endl;
    for(int i=1; i<=nnodos; i++){
        if(G[i]>=0){
            int j=buscarRaiz(G[i]);
            cout << -G[j] << endl;
        }
        else cout << -G[i] << endl;
    }

}

////////////////////////////////////////////////////////////
////////////        PROGRAMA PRINCIPAL        ///////////////
////////////////////////////////////////////////////////////
```

```
int main (void)
{
  int ncasos;
  cin >> ncasos;
  for (int i= 0; i<ncasos;) {
    cout << "Caso " << ++i << endl;
    leeGrafo();
    busquedaIslas();
  }
}
```

412

Idea: usar Dijsktra, pero calcular las distancias como la máxima distancia directa (pero finita) entre dos planetas dados del camino en que estamos.

No tiene más misterio, solo debemos cambiar el modo en que se actualizan los pesos de los caminos mínimos.

El orden de ejecución, en esta ocasión es O(m*log n), siendo n el número de nodos, y m el de aristas. Pues esta vez ya sí que caí en que al representar el grafo con listas de adyacencia, podía ahorrar iteraciones (bastantes, además) actualizando solo aquellos nodos que pueden ser modificados, es decir, a los que el nodo actual apunta directamente.

```
#include <stdlib.h>  // Funcion exit
#include <iostream>  // Variables cin y cout
#include <string.h>  // Funcion memset
#include <queue>     // Para implementar el recorrido
#include <list>     // Para representar el grafo
#include <vector>   // Para representar el grafo
using namespace std;

#define MAX_NODOS 10000
#define MAX_DIST 2000000

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

int nnodos;                     // Numero de nodos del grafo
int naristas;                   // Numero de aristas
int nrutas;                     // Numero de rutas a calcular
list<pair<int,int>> *G;         // Grafo

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

```
void leeGrafo (void)
// Procedimiento para leer un grafo de la entrada
{
  if(nnodos==0 && naristas==0) return;
  if (nnodos<0 || nnodos>MAX_NODOS) {
    cerr << "Numero de nodos (" << nnodos << ") no valido\n";
    exit(0);
  }
  G = new list<pair<int,int>>[nnodos];
  for(int i=0; i<naristas; i++){
    int u, v, c;
    cin >> u >> v >> c;
    G[u].push_back({v,c});
    G[v].push_back({u,c});
  }
}

int max(int a, int b){
    if(a>=b) return a;
    return b;
}

int min(int a, int b){
    if(a<=b) return a;
    return b;
}
```

```
int Dijkstra(int inicio, int fin)
{
  priority_queue<pair<int,int>,vector<pair<int,int> >,
greater<pair<int,int> > > cola;
  vector<int> D(nnodos, MAX_DIST);
  vector<bool> escogidos(nnodos,false);
  D[inicio]=0;
  cola.push({0, inicio});
  while(!cola.empty()){
    int v = cola.top().second;
    cola.pop();
    escogidos[v]=true;
    for(pair<int,int> p : G[v]){
        if(!escogidos[p.first]){
            int pp=D[p.first];
            D[p.first]=min(D[p.first], max(D[v], p.second));
            if(pp!=D[p.first])
                cola.push({D[p.first], p.first});
        }
    }
  }
  return D[fin];
}

void Dijk(){
  int inicio, llegada;
  cin >> inicio >> llegada;
  if(inicio==llegada) cout << 0 << endl;
  else {
      int dist=Dijkstra(inicio, llegada);
      cout << dist << endl;
  }
}

////////////////////////////////////////////////////////////
////////////        PROGRAMA PRINCIPAL        ///////////////
////////////////////////////////////////////////////////////
```

```
int main (void)
{
  cin >> nnodos >> naristas;
  while(nnodos!=0 || naristas !=0) {
    leeGrafo();
    if(nnodos!=0 || naristas!=0){
        int nrutas;
        cin >> nrutas;
        for(int i=0; i<nrutas; i++){
            Dijk();
        }
        delete[] G;
        cin >> nnodos >> naristas;
        if(nnodos!=0) cout << endl;
    }
  }
}
```

414

Idea: mantener una variable global booleana que nos indique cuando la marea está alta y baja. Aplicar Dijkstra teniendo esta variable en consideración.

De nuevo, este problema se resuelve con una ligera modificación en la actualización de distancias mínimas, teniendo en cuenta el factor de la marea. En esta ocasión sí que tuve en cuenta que una vez visitado el nodo de llegada (McDonald), no era necesario continuar con el algoritmo de Dijkstra (sí… elemental).
El orden de complejidad es, así, O(mlog n).

```
#include <stdlib.h>  // Funcion exit
#include <iostream>  // Variables cin y cout
#include <queue>     // Para implementar el recorrido
#include <list>      // Para representar el grafo
#include <vector>    // Para representar el grafo
using namespace std;

#define MAX_NODOS 20000
#define MAX_DIST 1000000

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

int N;                        // Numero de casas
int S;                        // Lo que tarda en subir la marea
int B;                        // Lo que tarda en bajar
int nnodos;                   // Número nodos
int naristas;                 // Número aristas
pair<int,int> especial;       // (Casa, McDonalds)
list<vector<int>> *G;         // Array de listas de adyacencia
bool marea=false;

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

```
void leeGrafo (void)
// Procedimiento para leer un grafo de la entrada
{
  cin >> S >> B;
  cin >> nnodos >> naristas;
  if (nnodos<0 || nnodos>MAX_NODOS) {
    cerr << "Numero de nodos (" << nnodos << ") no valido\n";
    exit(0);
  }
  G = new list<vector<int>>[nnodos]();

  cin >> especial.first >> especial.second;
  for(int i=0; i<naristas; i++){
    int u, v, p, s;
    cin >> u >> v >> p >> s;

    vector<int> vec(3,0), vecc(3,0);

    vec[0]=v; vec[1]=p; vec[2]=s;
    vecc[0]=u; vecc[1]=p; vecc[2]=s;

    G[u].push_back(vec);
    G[v].push_back(vecc);
  }
}

int minDist(int na, int nb, vector<int> D){
  for(vector<int> v : G[na]){
      if(v[0]==nb && D[na]+v[1]<D[nb] && (v[2]==1 || !marea))
          return v[1]+D[na];
      else if(v[0]==nb && D[na]+v[1]<D[nb] && (v[2]==0 && marea))
          return D[na]+v[1]+B-(D[na]%S);
      else if (v[0]==nb) return MAX_DIST;
  }
  return MAX_DIST;
}

int min(int a, int b){
    if(a<=b) return a;
```

```
    return b;
}

int Dijkstra(int inicio, vector<bool> &escogidos, int fin)
{
  priority_queue<pair<int,int>,vector<pair<int,int> >,
greater<pair<int,int> > > cola;
  vector<int> D(nnodos, MAX_DIST);
  D[inicio]=0;
  marea=false;
  cola.push({0, inicio});
  while(!cola.empty() && !escogidos[fin]){
    int v = cola.top().second;
    cola.pop();
    escogidos[v]=true;
    if(D[v]%(S+B)>=S) marea=true;
    else marea=false;
    for(vector<int> vec : G[v]){
        if(!escogidos[vec[0]]){
            int maybe=D[vec[0]];
            D[vec[0]]=min(minDist(v, vec[0], D), D[vec[0]]);
            if(maybe!=D[vec[0]])
                cola.push({D[vec[0]], vec[0]});
        }
    }
  }
  return D[fin];
}

void Dijk(){
  vector<bool> escogidos(nnodos,false);
  int dist=Dijkstra(especial.first, escogidos, especial.second);
  cout << dist << endl;
}


////////////////////////////////////////////////////////////
////////////        PROGRAMA PRINCIPAL        ///////////////
////////////////////////////////////////////////////////////
```

```
int main (void)
{
  int ncasos;
  cin >> ncasos;
  for (int i= 0; i<ncasos;i++) {
    leeGrafo();
    Dijk();
    delete[] G;
  }
}
```

415

Idea: comprobar que no hay ciclos (lo cual imposibilitaría la consecución de las tareas), y, una vez que sabemos que es un GDA, podemos aplicar un ordenamiento topológico como el visto en clase de teoría.

El orden de ejecución del recorrido topológico, implementado el grafo con listas de adyacencia, es O(n+a), la comprobación de si hay ciclos es de orden O(n), de forma que no influye en cómputo global de la complejidad del programa.

```
#include <stdlib.h>  // Funcion exit
#include <iostream>  // Variables cin y cout
#include <string.h>  // Funcion memset
#include <queue>     // Para implementar el recorrido topológico
#include <list>      // Para representar el grafo
#include <vector>    // Para representar el grafo
using namespace std;

#define MAX_NODOS 1000
#define MAX_DIST 1000

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

int nnodos;                    // Numero de nodos del grafo
int naristas;                  // Numero de aristas
list<pair<int,int>> *G;        // Listas de adyacencia
int *grado;                    // Array de grados
int *tiempo;                   // Array de tiempos

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

```
void leeGrafo (void)
// Procedimiento para leer un grafo de la entrada
{
  cin >> nnodos;
  if (nnodos<0 || nnodos>MAX_NODOS) {
    cerr << "Numero de nodos (" << nnodos << ") no valido\n";
    exit(0);
  }
  G = new list<pair<int,int>>[nnodos];
  grado = new int[nnodos];
  tiempo = new int[nnodos];
  for(int i=0; i<nnodos; i++){
      grado[i]=0;
  }
  for(int i=0; i<nnodos; i++){
    int pre, ti;
    cin >> ti >> pre;
    tiempo[i]=ti;
    while(pre!=0){
        G[pre-1].push_back({i, ti});
        grado[i]++;
        cin >> pre;
    }
  }
}

int max(int a, int b){
    if(a>=b) return a;
    return b;
}

int min(int a, int b){
    if(a<=b) return a;
    return b;
}
```

```
void Topological(){
    vector<bool> escogidos(nnodos,false);
    vector<int> toposort;
    vector<int> dist(nnodos, 0);
    queue<int> c;
    for(int i=0; i<nnodos; i++)
        if(grado[i]==0) {
            c.push(i);
            dist[i]=tiempo[i];
        }

    for(int i=0; i<nnodos; i++){
        if(c.empty()) break;
        int cero=c.front();
        c.pop();
        toposort.push_back(cero);
        escogidos[cero]=true;
        for (pair<int,int> p:G[cero]){
            grado[p.first]--;
            dist[p.first]=max(dist[p.first], dist[cero]+p.second);
            if(grado[p.first]==0)
                c.push(p.first);
        }
    }

    bool b=false;
    for(int i=0; i<nnodos;i++){
        if(!escogidos[i]) b=true;
    }
    if(b) cout << "IMPOSIBLE" << endl;
    else {
        int dista=0;
        for(int i=0; i<nnodos; i++){
            if(dist[i]>dista) dista=dist[i];
        }
        cout << dista << endl;
    }
}


//////////////////////////////////////////////////////////////
```

```
////////////        PROGRAMA PRINCIPAL        ////////////////
//////////////////////////////////////////////////////////////

int main (void)
{
  int ncasos;
  cin >> ncasos;
  for (int i= 0; i<ncasos;i++) {
    leeGrafo();
    else Topological();
    delete[] G;
  }
}
```

Conclusiones

La resolución de tan diversos problemas de grafos me ha hecho comprender el alcance de estos, la gran cantidad de escenarios que podemos modelizar por medio de grafos muestra su relevancia.
Además, me parecen la forma idónea de cerrar la asignatura, pues dejan entrever las distintas formas que hay de enfrentarse a una cuestión compleja, por un lado, y por otro muestran el potencial que puede tener un buen algoritmo.
AEDI es una asignatura que, sin duda, ayuda a concretar las ideas un poco difuminadas que obtenemos en primero, y es la asignatura que te hace ponerte a pie del cañón, con la inestimable compañía de tus algoritmos, y resolver problemas informáticos cercanos a los que podemos encontrarnos en la realidad.

Jose Antonio Lorencio Abril