Problemas, Algoritmos, y Programación
Trabajo Práctico 2

Manindra Agrawal

Segundo Cuatrimestre 2016

Índice

1. Problema B: Buenos gráficos

Peso del ejercicio: 9

1.1. Descripción del problema

Dados los precios de A acciones durante D dias, se nos pregunta la mínima cantidad de gráficos para graficar los precios de estas acciones en el tiempo de tal manera que los precios no se intersecten. Si los precios de dos acciones no se intersectan pueden dibujarse en un mismo gráfico.

1.2. Solución al problema

Nos construimos un grafo donde los vertices representan las acciones y el vertice i esta conectada con el vertice j si los precios de la acción j son estrictamente mayores que los precios de la acción i. Simbolicamente $G = (V, E)$ donde $V = \{a_1, \ldots, a_A\}$ y si $P_{i,t}$ denota al precio de la accioń i-esima a tiempo t entonces $(i, j) \in E$ si y solo si $P_{i,t} < P_{j,t}$ para todo $1 \leq t \leq D$.

El grafo es dirigido claramente, es transitivo pues si (i, j),$(j, k) \in E$ entonces $P_{i,t} < P_{j,t}$ y $P_{j,t} < P_{k,t}$ para todo $1 \leq t \leq D$ entonces $P_{i,t} < P_{k,t}$ para todo $1 \leq t \leq D$ entonces $(i, k) \in E$. También es aciclico pues si i y j estan en un ciclo por transitividad vale que (i, j),$(j, i) \in E$ entonces $P_{i,t} < P_{j,t}$ y $P_{j,t} < P_{i,t}$, absurdo. Luego G es un DAG.

Si llamamos r a la cantidad mínima de caminos para cubrir el DAG y s a la cantidad mínima de gráficos para repartir las acciones sin que haya solapamiento, quiero ver que $r = s$.

Notamos $g_1, \ldots, g_s$ a los gráficos, a cada gráfico g_k le corresponde un conjunto de acciones $a_{j_1}, \ldots, a_{j_{N_k}}$ que no se solapan por lo que puedo suponer que estan ordenadas de menor a mayor, con respecto a los precios, entonces $c_k = ((j_1, j_2), \ldots, (j_{N_k - 1}, j_{N_k}))$ es un camino en G. De esta manera me construyo $c_1, \ldots, c_s$ caminos y cubren G pues toda acción esta algún gráfico entonces toda acción esta en algun camino. Luego $r \leq s$.

Sea $c_1, \ldots, c_r$ el menor cubrimiento por caminos del grafo. Si $a_{j_1}, \ldots, a_{j_{N_k}}$ son los elementos de camino c_k ordenados de menor a mayor entonces $P_{i,t} < P_{i+1,t}$ para todo $1 \leq t \leq D$ y para todo $1 \leq i \leq N_k - 1$, luego me construyo el grafico g_k con estas acciones, no se solapan pues la diferencia entre los precios es positiva. Luego $s \leq r$. Luego $r = s$.

1.3. Algoritmo

Por lo visto en la clase de flujo para encontrar la cantidad mínima de caminos para cubrir un DAG de n vertices $v_1, \ldots, v_n$, me construyo un grafo bipartito duplicando los vertices $u_1, \ldots, u_n$ y conecto v_i con u_j si v_i estaba conectado con v_j en el DAG y calculo matching maximo de flujo resultante m sobre este grafo. Luego la minima cantidad de caminos para cubrir el DAG es $n - m$.

Detectar cuales vertices se conectan cuesta $\mathcal{O}(A^2D)$ y como el grafo bipartito tiene $2A$ nodos y A^2 aristas la complejidad de calcular matching maximo es $\mathcal{O}(A^3)$. En total el algoritmo tiene una complejidad de $\mathcal{O}(A^2(A+D))$ como se pedia en el enunciado.

1.4. Código de la solución

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>
#include <unordered_map>
#include <experimental/optional>

using namespace std;
using namespace std::experimental;

typedef int Node;
typedef vector <vector <Node>> Graph;

// Dar el corte mínimo del grafo usando el algoritmo de Edmonds-Karp.
int MinCut( Graph &G, Node S, Node T)
{
  // Flujo que hay entre cada par de nodos <x, y>.
  // Cada elemento puede ser igual a 0 (no pasa flujo), 1 (pasa flujo
  // de x a y), o -1 (pasa flujo de y a x).
  vector <vector <int>> flow(G.size(), vector <int> (G.size(), 0));

  // Buscar caminos de augmento que mejoren el flujo máximo usando BFS.
  // La cantidad de caminos de augmento está acotada en N * M en el
  // caso general (Cormen, teorema 26.8). De hecho, como hay una nueva
  // arista dirigida con peso 1 por cada uno de los N nodos originales
  // el flujo máximo F está acotado por N, y esto también acota la
  // cantidad de caminos en N. Por lo tanto, hay O(N) caminos.
  int maxFlow = 0;
  while (true)
  {
    // El predecesor de cada nodo en el camino de augmento.
    vector <optional<Node>> pred(G.size());

    // Hacer DFS desde la fuente hasta el sumidero.
    // Esto visita una vez cada nodo, y por cada nodo recorre
    // todas sus aristas, por lo que tarda O(M) iteraciones.
    queue <Node> q;
    q.push(S);
    while (!q.empty())
    {
      Node c = q.front(); q.pop();
      for (auto &e : G[c])
      {
        // El siguiente nodo puede ser parte del camino de augmento
        // si no es parte del camino actual ni de alguno anterior.
        if (!pred[e] && e != S && flow[c][e] < 1)
        {
          pred[e] = c;
          q.push(e);
        }
      }
    }

    // Si no se encontró ningún camino de augmento hasta el sumidero,
    // el algoritmo ya tiene el corte óptimo.
    if (!pred[T])
```

```
        break;

      // Recorrer el camino de augmento
      int df = 1 - flow[T][*pred[T]];
      for (Node e = *pred[T]; pred[e]; e = *pred[e])
        df = min(df, 1 - flow[e][*pred[e]]);

      for (Node e = T; pred[e]; e = *pred[e])
      {
        flow[*pred[e]][e] += df;
        flow[e][*pred[e]] -= df;
      }

      maxFlow += df;
    }

    return maxFlow;
}

int main(){
    int a,d;cin>>a>>d;
    vector<vector<int> > p(a,vector<int>(d));
    for(int i=0;i<a;i++)
      for(int j=0;j<d;j++)
        cin>>p[i][j];

    Graph G(2*a+2);
    Node S=0;Node T=2*a+1;

    //Conectamos la fuente y el sumidero
    for (int i = 0; i < a; i++)
    {
      G[S].push_back(i+1);
      G[i+1].push_back(S);
      G[i+1+a].push_back(T);
      G[T].push_back(i+1+a);
    }


    //Conectamos la primera columna con la segunda
    //Este procesamiento tiene complejidad O(a*a*d)
    for(int i=0;i<a;i++)
      for(int j=0;j<a;j++){
        if(j==i)continue;
        bool loAgrego = true;
        for(int t=0;t<d-1;t++){
          if(p[i][t]>p[j][t] && p[i][t+1]<p[j][t+1]){
            //si se cruzan en el medio
            loAgrego = false;
            break;
          }
          else if(p[i][t]<p[j][t] && p[i][t+1]>p[j][t+1]){
            //si se cruzan en el medio
            loAgrego = false;
            break;
          }else if(p[i][t]==p[j][t] || p[i][t+1]==p[j][t+1]){
            //si se cruzan en las puntas
```

```
          loAgrego = false;
          break;
        }else if(p[i][t]>p[j][t]){
          //solo agrego si la acción i-esima esta por debajo de la acción j-esima
          loAgrego = false;
          break;
        }
      }
      if(loAgrego){
        //Agrego solo si estoy por debajo de j
        G[i+1].push_back(j+1+a);
        G[j+1+a].push_back(i+1);
      }
    }
  //Calculamos el flujo maximo, como la cantidad de vertices
  //es 2a y la cantidad de aristas como máximo es a*a entonces
  //la complejidad es O(a*a*a)
  int k = MinCut(G,S,T);
  cout <<a-k<<endl;
  return 0;
}
```