LECCION 12: Continuación

Problema de la Asignación

. Existen n personas y n trabajos

, Cada persona i puede realizar un trabajo j con más o menos rendimiento: B[i,j]. (beneficio de asigna a la persona i la tarea j).

OBJETIVO - asignar una tarea a cada trabajador de manera que se maximice la suma del rendimiento.

$$\max \sum_{i=1}^{n} B[p_i, t_i] \quad \text{sujeto } p_i \neq p_j \Rightarrow t_i \neq t_j \ \forall i \neq j.$$

TAREAS

PERSONAS

B	1	2	3
1	4	9	1
2	7	2	3
3	6	3	5

Ej 1 (P_1, T_1) (P_2, T_3) (P_3, T_2)

$B_{TOTAL} = 4 + 3 + 3 = \boxed{10}$

Ej 2 (P_1, T_2) (P_2, T_1) (P_3, T_3)

$B_{TOTAL} = 9 + 7 + 5 = \boxed{21}$

ESQUEMA GREEDY

. Candidatos → personas a las que no se les ha asignado ninguna tarea.

. $S = (t_1, t_2, \ldots t_n)$ representa la solución en la que a la persona i-th se le ha asignada la tarea t_i

Será S solución cuando se le ha asignado a todos los trabajadores una tarea aún no escogida

, FUNCION SELECCION.- Escoger para el trabajador i la tarea aún no asignada de mayor beneficio.

, FUNCION FACTIBLE.- Siempre es true.

, FUNCION OBJETIVO $\max \sum_{i=1}^{n} B[\cancel{x}, S(i)]$ sujeto $p_i \neq p_j$ ~~$S(i) \neq S(j)$~~

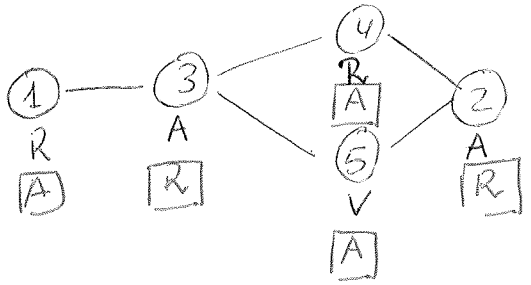
(2)

```
int Asignacion_Greedy (Matriz &B, vector<int> &S) {
    int n = S.size();
vector<bool> asignadas(n, false);

    int i = 0;
    int bene_total = 0;
    while (i < n) {
        j ⟸ Seleccion(i, B, asignadas);
                                    bene_total += B[i,j];
        asignadas[j] = true;
        S[i] = j;
        i++;
    }
    return bene_total;

}
int Seleccion (int trabajador, Matriz &B, vector<bool> asignadas){
    int n = asignadas.size();
    ~~do~~
    int max = numeric_limits<int>::min();
    int ~~xxx~~ job = -1;
    for (int i=0; i<n; i++)
        if(!asignada(i)) {
            if ( max < B[trabajador][i]) {
                max = B[trabajador][i];
                job = i;
            }
        }
    return job;

}.
```

LECCION 12.- Continuación

Problema.- COLORACIÓN de GRAFOS

Este problema consiste en asignar un color a cada nodo de forma que dos nodos unidos con un arco tenga siempre distinto color



3 colores.
2 colores

R: rojo
A: azul
V: verde.

Es un problema NP-completo

REPRESENTACIÓN de la SOLUCIÓN

$S = (c_1, c_2, \ldots, c_n)$ siendo c_i el color asignado al nodo i

La solución es valida si $\forall$ arista $a = (v_i, v_j)$ $c_{v_i} \neq c_{v_j}$

HEURISTICA TRIVIAL ←

1.- Inicialmente ningún nodo tiene color asignado
2.- Tomamos un color colorActual = 1
3.- Para cada uno de los nodos sin colorear
  3.1. Comprobar si es posible asignarle el color actual
  3.2.- Si se puede se le asigna ese color, en otro caso se deja sin colorear
4.- Si quedan nodos sin colorear, escoger otro color
  colorActual ← colorActual + 1
  ir al paso 3.

FUNCION de SELECCIÓN ← cualquier nodo sin colorear

FACTIBLE ← se puede asignar un color a x si ninguno de sus adyacentes tiene el mismo color.

- Este algoritmo no obtiene la solución óptima

```
#include <fstream>
#include <iostream>
#include "ngraph.hpp"
#include <vector>
#include <algorithm>
using namespace NGraph;
using namespace std;



struct info_vertice{
  Graph::vertex v;
  unsigned int degree;
  info_vertice(Graph::vertex vv, unsigned int d): v(vv),degree(d){}
};
bool compare(const info_vertice & v1,const info_vertice & v2){
   return v1.degree>v2.degree;
}
ostream & operator<<(ostream & os, const info_vertice &v){
  os<<v.v<< " " <<v.degree<<endl;
  return os;
}

template <class T>
ostream & operator<<(ostream & os, vector<T> & v){
  for (unsigned int i=0;i<v.size();++i)
    cout<<v[i];
  return os;
}


/**
 * @brief Obtiene el conjunto de vertices adyacente a un vertice dado
 * @param A: grafo de entrada
 * @param a: vertice sobre el que se quiere obtener sus GetAdyacentes
 * @return el conjunto de vertices adyacentes.
 */

Graph::vertex_set GetAdyacentes(Graph &A, Graph::vertex &a){
  Graph::vertex_set out;

  Graph::iterator it = A.find(a);
  if (it!=A.end()){

   out = Graph::out_neighbors(it);

  }
   return out;
}


/**
 *@brief Aplicamos el algoritmo de coloreo a un grafo usando la tecnica Voraz
 * @param A: grafo de entrada
 * @param result: un vector indicando por cada vertice el color asignado. No asi
gnado ==-1
 * @return el numero de colores necesarios
 *
 */
int Voraz_Coloreo(Graph & A,vector<info_vertice> &vertices, vector<int >&result)
{
```



```
  int ncolores=0;
  unsigned int nv=result.size();
  vector<bool>colores_usados(nv,false);
  vector<bool>libre_color(result.size(),true);

  for (unsigned int u=0; u<nv;++u){
    Graph::vertex_set out=GetAdyacentes(A,vertices[u].v);
    //Para cada uno de los adyacentes de u
    for (Graph::vertex_set::const_iterator q= out.begin(); q != out.end(); q++){
       unsigned int v= *q;

       v--;
       if (result[v]!=-1)
         libre_color[result[v]]=false;
    }
    //buscamos el primer color libre_color
    unsigned int cr;
    bool find=false;
    for (cr=0;cr<nv && !find; cr++)
       if (libre_color[cr]){
         find=true;
         result[vertices[u].v-1]=cr;
       }


    if (colores_usados[cr]==false){
      ncolores++;
      colores_usados[cr]=true;
    }
    for (Graph::vertex_set::const_iterator q= out.begin(); q != out.end(); q++){
       unsigned int v= *q;

       v--;
       if (result[v]!=-1)
         libre_color[result[v]]=true;
    }

  }
  return ncolores;
}

int main(int argc, char *argv[]){
  if (argc!=2){
     cout<<"Dime el fichero con los vertices y conexiones";
     return 0;
  }
  ifstream f (argv[1]);
  if (!f){
      cout<< "No puedo abrir el fichero"<<endl;
      return 0;
  }
  int v1,v2;
  Graph A;
  //Al ser no dirigido el grafo lo insertamos en las dos direcciones
  while (f>>v1>>v2){
      A.insert_edge(v1,v2);
      A.insert_edge(v2,v1);
  }

  vector<info_vertice> vver;
```