Manual de algoritmos del semillero de programación EAFIT

Ana Echavarría

22 de mayo de 2013

Índice

1. Plantilla

```
using namespace std;
#include <algorithm>
#include <iostream>
#include <iterator>
#include <numeric>
#include <sstream>
#include <fstream>
#include <cassert>
#include <climits>
#include <cstdlib>
#include <cstring>
#include <string>
#include <cstdio>
#include <vector>
#include <cmath>
#include <queue>
#include <stack>
#include <list>
#include <map>
#include <set>

// Template para recorrer contenedores usando iteradores
#define foreach(x, v) for (typeof (v).begin() x=(v).begin(); \
                           x !=(v).end(); ++x)
// Template que imprime valores de variables para depurar
#define D(x) cout << #x " = " << (x) << endl

// Función para comparar dos dobles sin problemas de presición
// Retorna -1 si x < y, 0 si x = y, 1 si x > y
const double EPS = 1e-9;
int cmp (double x, double y, double tol = EPS){
```

```
    return (x <= y + tol) ? (x + tol < y) ? -1 : 0 : 1;
}


int main() {
    // Entrada y salida desde / hacia archivo
    // Eliminar si la entrada es estándar
    // Cambiar in.txt / out.txt por los archivos de entrada/salida
    freopen("in.txt", "r", stdin);
    freopen("out.txt", "w", stdout);

    return 0;
}
```

2. Grafos

2.1. BFS

Algoritmo de recorrido de grafos en anchura que empieza desde una fuente s y visita todos los nodos alcanzables desde s.
El BFS también halla la distancia más corta entre s y los demás nodos si las aristas tienen todas peso 1.
Complejidad: $O(n + m)$ donde n es el número de nodos y m es el número de aristas.

```
vector <int> g[MAXN]; // La lista de adyacencia
int d[MAXN];          // Distancia de la fuente a cada nodo

void bfs(int s, int n){ // s = fuente, n = número de nodos
   for (int i = 0; i <= n; ++i) d[i] = -1;

   queue <int> q;
   q.push(s);
   d[s] = 0;
   while (q.size() > 0){
      int cur = q.front();
      q.pop();
      for (int i = 0; i < g[cur].size(); ++i){
         int next = g[cur][i];
         if (d[next] == -1){
            d[next] = d[cur] + 1;
            q.push(next);
         }
      }
   }
}
```

2.2. DFS

Algoritmo de recorrido de grafos en profundidad que empieza visita todos los nodos del grafo.
El algoritmo puede ser modificado para que retorne información de los nodos según la necesidad del problema.
El grafo tiene un ciclo $\leftrightarrow$ si en algún momento se llega a un nodo marcado como gris.
Complejidad: $O(n + m)$ donde n es el número de nodos y m es el número de aristas.

```
vector <int> g[MAXN];      // La lista de adyacencia
int color[MAXN];           // El arreglo de visitados
enum {WHITE, GRAY, BLACK}; // WHITE = 1, GRAY = 2, BLACK = 3

// Visita el nodo u y todos sus vecinos empezando por los más profundos
void dfs(int u){
   color[u] = GRAY;   // Marcar el nodo como semi-visitado
   for (int i = 0; i < g[u].size(); ++i){
     int v = g[u][i];
     if (color[v] == WHITE) dfs(v);  // Visitar los vecinos
   }
   color[u] = BLACK;  // Marcar el nodo como visitado
}

// Llama la función dfs para los nodos 0 a n-1
void call_dfs(int n){
   for (int u = 0; u < n; ++u) color[u] = WHITE;
   for (int u = 0; u < n; ++u)
     if (color[u] == WHITE) dfs(u);
}
```

2.3. Ordenamiento topológico

Dado un grafo no cíclico y dirigido (DAG), ordena los nodos linealemente de tal forma que si existe una arista entre los nodos u y v entonces u aparece antes que v en el ordenamiento.
Este ordenamiento se puede ver como una forma de poner todos los nodos en una línea recta y que las aristas vayan todas de izquierda a derecha.
Complejidad: $O(n+m)$ donde n es el número de nodos y m es el número de aristas.

```
vector <int> g[MAXN];    // La lista de adyacencia
bool seen[MAXN];         // El arreglo de visitados para el dfs
vector <int> topo_sort;  // El vector del ordemamiento

void dfs(int u){
   seen[u] = true;
   for (int i = 0; i < g[u].size(); ++i){
      int v = g[u][i];
      if (!seen[v]) dfs(v);
   }
   topo_sort.push_back(u); // Agregar el nodo al ordenamiento
}
void topological(int n){   // n = número de nodos
   topo_sort.clear();
   for (int i = 0; i < n; ++i) seen[i] = false;
   for (int i = 0; i < n; ++i) if (!seen[i]) dfs(i);
   reverse(topo_sort.begin(), topo_sort.end());
}
```

.....

2.4. Componentes fuertemente conexas

Dado un grafo dirigido, calcula la componente fuertemente conexa (SCC) a la que pertenece cada nodo.
Para cada pareja de nodos u, v que pertenecen a una misma SCC se cumple que hay un camino de u a v y de v a u.
Si se comprime el grafo dejando como nodos cada una de las componentes se quedará con un DAG.
Complejidad: $O(n+m)$ donde n es el número de nodos y m es el número de aristas.

```
vector <int> g[MAXN];    // El grafo
vector <int> grev[MAXN]; // El grafo con las aristas reversadas
vector <int> topo_sort;  // El "ordenamiento topologico" del grafo
int scc[MAXN];   // La componente a la que pertenece cada nodo
bool seen[MAXN]; // El arreglo de visitado para el primer DFS

// DFS donde se halla el ordenamiento topológico
void dfs1(int u){
   seen[u] = true;
   for (int i = 0; i < g[u].size(); ++i){
      int v = g[u][i];
      if (!seen[v]) dfs1(v);
   }
   topo_sort.push_back(u);
}
// DFS donde se hallan las componentes
void dfs2(int u, int comp){
   scc[u] = comp;
   for (int i = 0; i < grev[u].size(); ++i){
      int v = grev[u][i];
      if (scc[v] == -1) dfs2(v, comp);
   }
}

// Halla las componentes fuertemente conexas del grafo usando
// el algoritmo de Kosaraju. Retorna la cantidad de componentes
int find_scc(int n){ // n = número de nodos
   // Crear el grafo reversado
   for (int u = 0; u < n; ++u){
      for (int i = 0; i < g[u].size(); ++i){
         int v = g[u][i];
         grev[v].push_back(u);
      }
   }

   // Llamar el primer dfs
   for (int i = 0; i < n; ++i){
      if (!seen[i]) dfs1(i);
   }
   reverse(topo_sort.begin(), topo_sort.end());

   // Llamar el segundo dfs
   int comp = 0;
   for (int i = 0; i < n; ++i){
```

```
      int u = topo_sort[i];
      if (scc[u] == -1) dfs2(u, comp++);
   }
   return comp;
}
```

2.5. Algoritmo de Dijkstra

Dado un grafo con pesos **no negativos** en las aristas, halla la mínima distancia entre una fuente s y los demás nodos.
Al heap se inserta primero la distancia y luego en nodo al que se llega. Si se quieren modificar los pesos por `long long` o por `double` se debe cambiar en los tipos de dato `dist_node` y `edge`.
Complejidad $O((n+m)\log n)$ donde n es el número de nodos y m es el número de aristas.

```
typedef pair <int, int> dist_node; // Datos para el heap (dist, nodo)
typedef pair <int, int> edge; // Dato para las arista (nodo, peso)
const int MAXN = 100005;      // El máximo número de nodos
const int INF = 1 << 30;      // Usar 1LL << 60 para long long
vector <edge> g[MAXN];        // g[u] = (v = nodo, w = peso)
int d[MAXN];    // d[u] La distancia más corta de s a u
int p[MAXN];    // p[u] El predecesor de u en el camino más corto

// La función recibe la fuente s y el número total de nodos n
void dijkstra(int s, int n){
   for (int i = 0; i <= n; ++i){
      d[i] = INF;  p[i] = -1;
   }
   priority_queue < dist_node, vector <dist_node>,
                    greater<dist_node> > q;
   d[s] = 0;
   q.push(dist_node(0, s));
   while (!q.empty()){
      int dist = q.top().first;
      int cur = q.top().second;
      q.pop();
      if (dist > d[cur]) continue;
      for (int i = 0; i < g[cur].size(); ++i){
         int next = g[cur][i].first;
         int w_extra = g[cur][i].second;
         if (d[cur] + w_extra < d[next]){
            d[next] = d[cur] + w_extra;
            p[next] = cur;
            q.push(dist_node(d[next], next));
         }
      }
   }
}

// La función que retorna los nodos del camino más corto de s a t
// Primero hay que correr dijktra desde s.
// Eliminar si no se necesita hallar el camino.
vector <int> find_path (int t){
   vector <int> path;
   int cur = t;
   while(cur != -1){
      path.push_back(cur);
      cur = p[cur];
   }
   reverse(path.begin(), path.end());
   return path;
}
```