Grafos.

Material de referencia.

Índice

1. Caminos más cortos.

Consideremos un grafo $G = (V, E)$ donde cada arista (u, v) tiene una longitud $d(u, v) > 0$. El camino más corto entre dos vértices es aquel que minimiza la longitud total del camino.

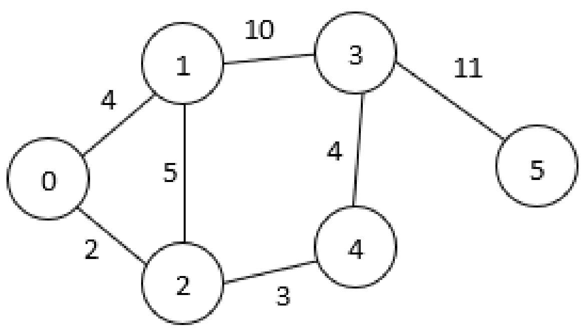
1.1. Algoritmo de Dijkstra

Complejidad: $O((|E| + |V|) \log |V|)$.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>
#include <utility>
using namespace std;

#define maxn 100000 //Maximo numero de vertices.

typedef pair<int, int> edge;
#define length first
#define to     second

int V, E;                    //Numero de vertices y aristas.
vector<edge> graph[maxn];    //Aristas.

int s;                       //Vertice inicial.
int dist[maxn], pred[maxn];  //Distancia desde s y predecesor en el camino.
bool vis[maxn];              //Visitado.

//Encuentra el camino mas corto desde un vertice a todos los demas.
void Dijkstra() {
    fill_n(dist, V, 1e9);
    fill_n(pred, V, -1);
    dist[s] = 0;

    priority_queue<edge> pq;
    pq.push(edge(dist[s], s));

    while (!pq.empty()) {
        int curr = pq.top().to;
        pq.pop();
        vis[curr] = true;

        for (edge e : graph[curr])
            if (!vis[e.to] && dist[curr] + e.length < dist[e.to]) {
                dist[e.to] = dist[curr] + e.length;
                pred[e.to] = curr;
                pq.push(edge(-dist[e.to], e.to));
            }
    }
}

int main() {
    ios_base::sync_with_stdio(0); cin.tie();
    cin >> V >> E;

```

```
    //Lee la informacion de las aristas.
    for (int i = 0; i < E; ++i) {
        int u, v, d;
        cin >> u >> v >> d;
        graph[u].push_back(edge(d, v));
        graph[v].push_back(edge(d, u));
    }

    //Imprime la configuracion.
    cin >> s;
    Dijkstra();
    for (int i = 0; i < V; ++i)
        cout << i << ": " << pred[i] << ' ' << dist[i] << '\n';

    return 0;
}
```

Entrada	Salida
6 7 0 1 4 1 3 10 3 5 11 1 2 5 2 0 2 2 4 3 4 3 4 0	0: -1 0 1: 0 4 2: 0 2 3: 4 9 4: 2 5 5: 3 20



2. Árbol de expansión mínima.

Consideremos un grafo $G = (V, E)$ donde cada arista (u, v) tiene un peso $w(u, v)$. Un árbol de expansión mínima es un subgrafo de G que es árbol y minimiza la suma de los pesos.

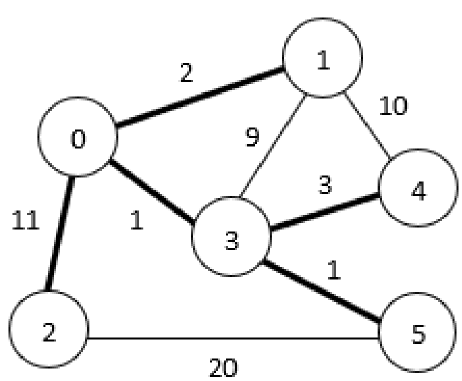
2.1. Algoritmo de Kruskal

Complejidad: $O(|E| \log |V|)$.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <utility>
using namespace std;

#define maxn 100000 //Maximo numero de vertices.

typedef pair<int, pair<int, int> > edge;
#define weight first
#define from   second.first
#define to     second.second

int V, E;          //Numero de vertices y aristas.
edge graph[maxn]; //Aristas.

int parent[maxn], Rank[maxn]; //Union-Find por rango y compresion de camino.
vector<int> MST;              //Arbol de expansion minima.

int Find(int x) {
    if (parent[x] != x)
        parent[x] = Find(parent[x]);
    return parent[x];
}

bool Union(int x, int y) {
    int a = Find(x), b = Find(y);
    if (a == b)
        return false;
    else {
        if (Rank[a] < Rank[b])
            parent[a] = b;
        else {
            parent[b] = a;
            if (Rank[a] == Rank[b])
                Rank[a]++;
        }
        return true;
    }
}

//Encuentra el arbol de expansion minima.
int Kruskal() {
    int W = 0;
    for (int i = 0; i < V; ++i) {
        parent[i] = i;
```

```
            Rank[i] = 0;
        }

        sort(graph, graph + E);
        for (int i = 0; i < E; ++i)
            if (Union(graph[i].from, graph[i].to)) {
                W += graph[i].weight;
                MST.push_back(i);
            }
        return W;
}

int main() {
        ios_base::sync_with_stdio(0); cin.tie();
        cin >> V >> E;

        //Lee la informacion de las aristas.
        for (int i = 0; i < E; ++i)
            cin >> graph[i].from >> graph[i].to >> graph[i].weight;

        //Imprime la configuracion del arbol de expansion minima.
        cout << "Peso total: " << Kruskal() << '\n';
        for (int i : MST)
            cout << graph[i].from << ' ' << graph[i].to << ' ' << graph[i].weight
                << '\n';
        return 0;
}
```

Entrada	**Salida**
6 8 0 1 2 0 3 1 3 1 9 4 1 10 3 4 3 2 0 11 2 5 20 3 5 1	Peso total: 18 0 3 1 3 5 1 0 1 2 3 4 3 2 0 11



3. Orden topológico.

Consideremos un grafo dirigido acíclico $G = (V, E)$. Un orden topológico es un ordenamiento lineal de los vértices en donde las aristas conectan solamente con vértices posteriores.

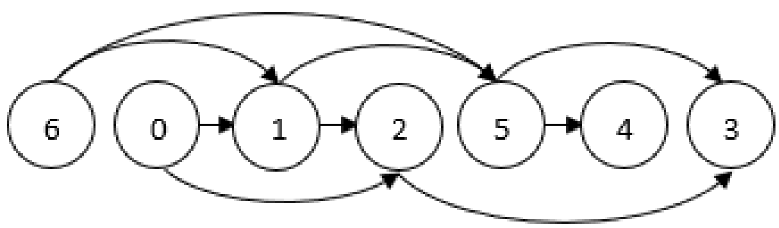
3.1. Algoritmo basado en DFS

Complejidad: $O(|V| + |E|)$.

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

#define maxn 100000 //Maximo numero de vertices.

int V, E;                  //Numero de vertices y aristas.
vector<int> graph[maxn];  //Aristas.

bool cycle;          //Verifica que no haya ciclos.
vector<int> sorted;  //Orden topologico.
int vis[maxn];       //Visitado.

//Encuentra el orden topologico iniciando en un vertice dado.
void DFS(int u) {
    if (vis[u] == 1) {
        cycle = true;
        return;
    }
    else if (!vis[u]) {
        vis[u] = 1;
        for (int v : graph[u])
            DFS(v);
        vis[u] = -1;
        sorted.push_back(u);
    }
}

//Encuentra el orden topologico.
void ToopologicalSort() {
    for (int u = 0; u < V; ++u)
        DFS(u);
    reverse(sorted.begin(), sorted.end());
}

int main() {
    ios_base::sync_with_stdio(0); cin.tie();
    cin >> V >> E;

    //Lee la informacion de las aristas.
    for (int i = 0; i < E; ++i) {
        int from, to;
        cin >> from >> to;
        graph[from].push_back(to);
    }

```

```
    //Imprime el orden topologico
    ToopologicalSort();
    if (cycle)
        cout << "No es un DAG.\n";
    else {
        for (int u : sorted)
            cout << u << ' ';
        cout << '\n';
    }

    return 0;
}
```

Entrada	Salida
7 9 6 1 6 5 0 1 1 5 0 2 1 2 2 3 5 3 5 4	6 0 1 2 5 4 3



4. Componentes fuertemente conexas.

Consideremos un grafo dirigido $G = (V, E)$. Decimos que G es fuertemente conexo si existe un camino entre cualesquiera par de vértices.

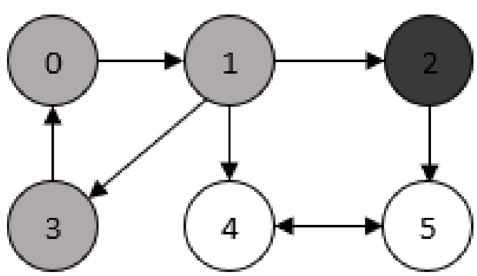
4.1. Algoritmo de Tarjan

Complejidad: $O(|V| + |E|)$.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <stack>
using namespace std;

#define maxn 100000 //Maximo numero de vertices.

int V, E;                    //Numero de vertices y aristas.
vector<int> graph[maxn]; //Aristas.

vector<vector<int>> SCC; //Componentes fuertemente conexas.

int idx[maxn], low[maxn], lst_id; //Indices, ultimo indice.
stack<int> S;                     //Vertices pendientes.
bool onStack[maxn];               //Esta en la pila.

//Encuentra la componente fuertemente conexa de u.
void StrongConnect(int u) {
    idx[u] = ++lst_id;
    low[u] = lst_id;
    S.push(u);
    onStack[u] = true;

    for (int v : graph[u]) {
        if(!idx[v]) {
            StrongConnect(v);
            low[u] = min(low[u], low[v]);
        }
        else if (onStack[v])
            low[u] = min(low[u], idx[v]);
    }

    if (low[u] == idx[u]) {
        SCC.push_back(vector<int>());
        while (S.top() != u) {
            onStack[S.top()] = false;
            SCC.back().push_back(S.top());
            S.pop();
        }

        onStack[u] = false;
        SCC.back().push_back(u);
        S.pop();
    }
}

```

```cpp
//Algoritmo de Tarjan para encontrar las componentes fuertemente conexas.
void Tarjan() {
    for (int u = 0; u < V; ++u)
        if (!idx[u])
            StrongConnect(u);
}

int main() {
    ios_base::sync_with_stdio(0); cin.tie();
    cin >> V >> E;

    //Lee la informacion de las aristas.
    for (int i = 0; i < E; ++i) {
        int from, to;
        cin >> from >> to;
        graph[from].push_back(to);
    }

    //Imprime las componentes fuertemente conexas.
    Tarjan();
    for (int i = 0; i < SCC.size(); ++i) {
        for (int u : SCC[i])
            cout << u << ' ';
        cout << '\n';
    }

    return 0;
}
```

Entrada	**Salida**
6 8 0 1 1 2 1 4 1 3 3 0 2 5 4 5 5 4	4 5 2 3 1 0



5. Puentes y puntos de articulación.

Consideremos un grafo $G = (V, E)$. Una arista (u, v) es un puente si al eliminarla el grafo queda disconexo. De igual manera, un vértice u es un punto de articulación si al eliminarlo el grafo queda disconexo.

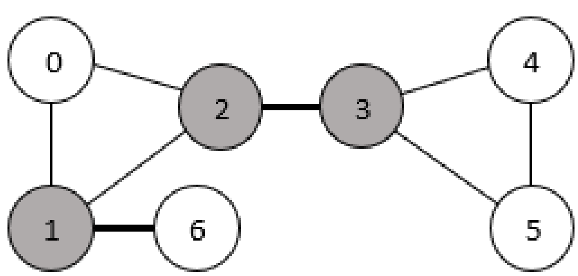
5.1. Algoritmo de Tarjan

Complejidad: $O(|V| + |E|)$.

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <utility>
using namespace std;

#define maxn 100000 //Maximo numero de vertices.

int V, E;                //Numero de vertices y aristas.
vector<int> graph[maxn]; //Aristas.

bool artpoint[maxn];          //Puntos de articulacion
vector<pair<int, int>> bridge; //Puentes

int idx[maxn], low[maxn], lst_id; //Indices, ultimo indice.

//Enumera los vertices con una DFS.
void DFS(int u, int pred) {
    idx[u] = ++lst_id;
    low[u] = lst_id;
    int children = 0;

    for (int v : graph[u]) {
        if(!idx[v]) {
            ++children;
            DFS(v, u);
            low[u] = min(low[u], low[v]);

            if ((pred == -1 && children > 1) || (pred != -1 && low[v] >= idx[u]))
                artpoint[u] = true;
            if (low[v] > idx[u])
                bridge.push_back(make_pair(u, v));
        }
        else if (v != pred)
            low[u] = min(low[u], idx[v]);
    }
}

//Algoritmo de Tarjan.
void Tarjan() {
    for (int u = 0; u < V; ++u)
        if (!idx[u])
            DFS(u, -1);
}
```

```cpp
int main() {
    ios_base::sync_with_stdio(0); cin.tie();
    cin >> V >> E;

    //Lee la informacion de las aristas.
    for (int i = 0; i < E; ++i) {
        int u, v;
        cin >> u >> v;
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    //Imprime los puentes y puntos de articulacion.
    Tarjan();
    cout << "Puntos de articulacion:\n";
    for (int i = 0; i < V; ++i)
        if (artpoint[i])
                cout << i << ' ';
    cout << "\nPuentes:\n";
    for (int i = 0; i < bridge.size(); ++i)
        cout << bridge[i].first << ' ' << bridge[i].second << '\n';

    return 0;
}
```

Entrada	**Salida**
7 8 0 2 0 1 1 2 1 6 2 3 3 4 4 5 3 5	Puntos de articulacion: 1 2 3 Puentes: 1 6 2 3



6. Flujo máximo.

Consideremos un grafo dirigido $G = (V, E)$ donde cada arista (u, v) tiene asociada una capacidad $c(u, v) > 0$. Un flujo de s a t es una función que a cada arista le asigna un número $f(u, v)$ que satisface

- $f(u, v) \leq c(u, v)$.
- Para cualquier vértice $v \neq s, t$, el flujo que entra es igual al flujo que sale; s solo tiene flujo saliente y t solo tiene flujo entrante.

El flujo total es el flujo que sale de s.

6.1. Algoritmo de Dinic

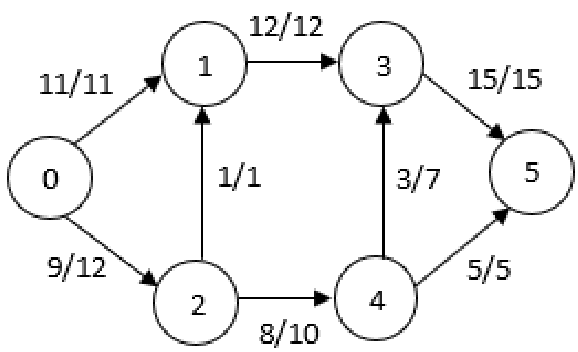
Complejidad: $O(|V|^2|E|)$.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>
using namespace std;

#define maxn 100000 //Maximo numero de vertices.
typedef int T;      //Tipo de dato del flujo.

struct edge {
    int to;             //Destino.
    T capacity, flow;   //Capacidad, flujo.
    int rev;            //Arista invertida.
};

int V, E;                     //Numero de vertices y aristas.
vector<edge> graph[maxn];     //Aristas.

int s, t;                     //Fuente y sumidero.
int level[maxn], ptr[maxn];   //Distancia a s y numero de aristas visitadas.

//Verifica si se puede enviar flujo de s a t.
bool BFS() {
    queue<int> Q;
    fill_n(level, V, -1);
    level[s] = 0;
    Q.push(s);

    while (!Q.empty()) {
        int curr = Q.front();
        Q.pop();
        for (edge e : graph[curr])
            if (level[e.to] == -1 && e.flow < e.capacity) {
                level[e.to] = level[curr] + 1;
                Q.push(e.to);
            }
    }

```

```
    return level[t] != -1;
}

//Envia flujo de s a t.
T DFS(int curr, T flow) {
    if (curr == t)
        return flow;

    for (; ptr[curr] < graph[curr].size(); ++ptr[curr]) {
        edge &e = graph[curr][ptr[curr]];

        if (level[e.to] == level[curr] + 1 && e.flow < e.capacity) {
            T currflow = DFS(e.to, min(flow, e.capacity - e.flow));
            if (currflow > 0) {
                e.flow += currflow;
                graph[e.to][e.rev].flow -= currflow;
                return currflow;
            }
        }
    }

    return 0;
}

//Calcula el maximo flujo de s a t.
T Dinic() {
    T flow = 0, currflow;
    while (BFS()) {
        fill_n(ptr, V, 0);
        do {
            currflow = DFS(s, 1e9);
            flow += currflow;
        }
        while (currflow > 0);
    }
    return flow;
}

int main() {
    ios_base::sync_with_stdio(0); cin.tie();
    cin >> V >> E >> s >> t;

    //Lee la informacion de las aristas.
    for (int i = 0; i < E; ++i) {
        int from, to;
        T capacity;
        cin >> from >> to >> capacity;

        graph[from].push_back(edge {to, capacity, 0, (int)graph[to].size()});
        graph[to].push_back(edge {from, 0, 0, (int)graph[from].size() - 1});
    }

    //Imprime la configuracion del flujo.
    cout << "Flujo maximo: " << Dinic() << '\n';
    for (int i = 0; i < V; ++i)
        for (edge e : graph[i])
```

```
            if (e.capacity > 0)
                cout << i << ' ' << e.to << ": " << e.flow << '/' << e.capacity << '\n';

    return 0;
}
```

Entrada	**Salida**
6 8 0 5 0 1 11 0 2 12 1 3 12 2 1 1 2 4 10 4 3 7 3 5 15 4 5 5	Flujo maximo: 20 0 1: 11/11 0 2: 9/12 1 3: 12/12 2 1: 1/1 2 4: 8/10 3 5: 15/15 4 3: 3/7 4 5: 5/5



7. Emparejamiento máximo.

Consideremos un grafo bipartito $G = (U \cup V, E)$. Un emparejamiento de G es un subgrafo en donde cada vértice pertenece a lo más a una arista.

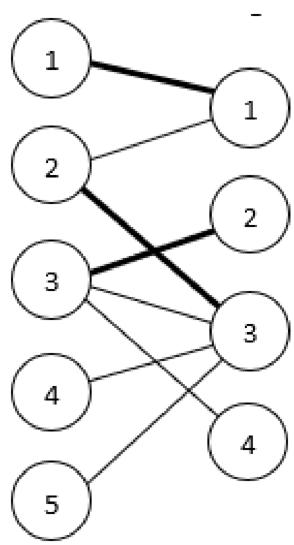
7.1. Algoritmo de Hopcroft-Karp

Complejidad: $O(|E|\sqrt{|V|})$.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>
using namespace std;

#define maxn 100000 //Maximo numero de vertices.

int U, V, E;                //Numero de vertices en cada lado y de aristas.
vector<int> graph[maxn]; //Aristas que van de U a V.

int pairU[maxn], pairV[maxn], dist[maxn]; //Pares de vertices en el
    emparejamiento.

//Verifica si existe un camino de aumento.
bool BFS() {
    queue<int> Q;
    for (int u = 1; u <= U; ++u) {
        if (!pairU[u]) {
            dist[u] = 0;
            Q.push(u);
        }
        else
            dist[u] = 1e9;
    }
    dist[0] = 1e9;

    while (!Q.empty()) {
        int u = Q.front();
        Q.pop();
        if (dist[u] < dist[0])
            for (int v : graph[u])
                if (dist[pairV[v]] == 1e9) {
                    dist[pairV[v]] = dist[u] + 1;
                    Q.push(pairV[v]);
                }
    }
    return (dist[0] != 1e9);
}

//Verifica si existe un camino de aumento que comience en u.
bool DFS(int u) {
    if (u != 0) {
        for (int v : graph[u])
            if (dist[pairV[v]] == dist[u] + 1 && DFS(pairV[v])) {
                pairV[v] = u;
                pairU[u] = v;
```

```
                    return true;
                }

            dist[u] = 1e9;
            return false;
        }
        return true;
}

//Busca un emparejamiento maximo.
int HopcroftKarp() {
        int size = 0;
        while (BFS())
            for (int u = 1; u <= U; ++u)
                if (!pairU[u] && DFS(u))
                    ++size;
        return size;
}

int main() {
        ios_base::sync_with_stdio(0); cin.tie();
        cin >> U >> V >> E;

        //Lee las aristas. Los vertices estan indexados en 1.
        for (int i = 0; i < E; ++i) {
            int u, v;
            cin >> u >> v;
            graph[u].push_back(v);
        }

        //Imprime la configuracion del emparejamiento.
        cout << "Emparejamiento: " << HopcroftKarp() << '\n';
        for (int u = 1; u <= U; ++u)
            if (pairU[u])
                cout << u << " - " << pairU[u] << '\n';

        return 0;
}
```

Entrada	**Salida**
5 4 8 1 1 2 1 2 3 3 2 3 3 3 4 4 3 5 3	Emparejamiento: 3 1 - 1 2 - 3 3 - 2


1
2
3
4
5
1
2
3
4