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 minimiza la longitud total del camino.

1.1. Algoritmo de Dijkstra

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

```cpp
#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], prevv[maxn]; //Distancia desde s y antecesor 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(prevv, V, -1);
    fill_n(vis, V, false);
    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;
                prevv[e.to] = curr;
                pq.push(edge(-dist[e.to], e.to));
            }
    }
}

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

```
48
49      //Lee la informacion de las aristas.
50      for (int i = 0; i < E; ++i) {
51          int u, v, d;
52          cin >> u >> v >> d;
53          graph[u].push_back(edge(d, v));
54          graph[v].push_back(edge(d, u));
55      }
56
57      //Imprime la configuracion.
58      cin >> s;
59      Dijkstra();
60      for (int i = 0; i < V; ++i)
61          cout << i << ": " << prevv[i] << ' ' << dist[i] << '\n';
62
63      return 0;
64  }
```

Entrada	**Salida**
6 8 0 1 11 0 2 12 1 3 12 2 1 1 2 4 11 4 3 7 3 5 19 4 5 5 0 5	Flujo maximo: 23 0 1: 11/11 0 2: 12/12 1 3: 12/12 2 1: 1/1 2 4: 11/11 3 5: 18/19 4 3: 6/7 4 5: 5/5

2. Máximo flujo.

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.

2.1. Algoritmo de Edmonds-Karp

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

```
#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.
    edge *rev;        //Arista invertida.

    edge(int _to, T _capacity, T _flow, edge *_rev) {
        to = _to; capacity = _capacity; flow = _flow; rev = _rev;
    }
};

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

//Calcula el flujo maximo de s a t.
T EdmondsKarp() {
    T flow = 0;
    edge *pred[maxn];

    do {
        //Realiza una BFS desde s hasta t.
        queue<int> Q;
        Q.push(s);
        fill_n(pred, V, nullptr);

        while (!Q.empty()) {
            int curr = Q.front();
            Q.pop();
            for (edge *e : graph[curr])
```

```
39                  if (pred[e->to] == nullptr && e->to != s && e->capacity > e->
                        flow) {
40                      pred[e->to] = e;
41                      Q.push(e->to);
42                  }
43          }
44
45          //Encontramos un camino de aumento.
46          if (pred[t] != nullptr) {
47              T df = 1e9;
48              for (edge *e = pred[t]; e != nullptr; e = pred[e->rev->to])
49                  df = min(df, e->capacity - e->flow);
50              for (edge *e = pred[t]; e != nullptr; e = pred[e->rev->to]) {
51                  e->flow += df;
52                  e->rev->flow -= df;
53              }
54              flow += df;
55          }
56      }
57      while (pred[t] != nullptr);
58
59      return flow;
60  }
61
62  int main() {
63      ios_base::sync_with_stdio(0); cin.tie();
64      cin >> V >> E;
65
66      //Lee la informacion de las aristas.
67      for (int i = 0; i < E; ++i) {
68          int from, to;
69          T capacity;
70          cin >> from >> to >> capacity;
71
72          graph[from].push_back(new edge(to, capacity, 0, nullptr));
73          graph[to].push_back(new edge(from, 0, 0, graph[from].back()));
74          graph[from].back()->rev = graph[to].back();
75      }
76
77      cin >> s >> t;
78      cout << "Flujo maximo: " << EdmondsKarp() << '\n';
79
80      //Imprime la configuracion del flujo.
81      for (int i = 0; i < V; ++i)
82          for (edge *e : graph[i]) {
83              if (e->capacity > 0)
84                  cout << i << ' ' << e->to << ": " << e->flow << '/' << e->
                        capacity <<'\n';
85              delete e;
86          }
87
88      return 0;
89  }
```

Entrada	**Salida**
6 8 0 1 11 0 2 12 1 3 12 2 1 1 2 4 11 4 3 7 3 5 19 4 5 5 0 5	Flujo maximo: 23 0 1: 11/11 0 2: 12/12 1 3: 12/12 2 1: 1/1 2 4: 11/11 3 5: 18/19 4 3: 6/7 4 5: 5/5

3. 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.

3.1. 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 numero 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) {
        for (int v : graph[u])
            if (dist[pairV[v]] == dist[u] + 1 && DFS(pairV[v])) {
                pairV[v] = u;
```

```
46                  pairU[u] = v;
47                  return true;
48              }
49
50          dist[u] = 1e9;
51          return false;
52      }
53      return true;
54  }
55
56  //Busca un emparejamiento maximo.
57  int HopcroftKarp() {
58      int size = 0;
59      fill_n(pairU, U, 0);
60      fill_n(pairV, V, 0);
61      while (BFS())
62          for (int u = 1; u <= U; ++u)
63              if (!pairU[u] && DFS(u))
64                  size++;
65      return size;
66  }
67
68  int main() {
69      ios_base::sync_with_stdio(0); cin.tie();
70      cin >> U >> V >> E;
71
72      //Lee las aristas. Los vertices estan indexados en 1.
73      for (int i = 0; i < E; ++i) {
74          int u, v;
75          cin >> u >> v;
76          graph[u].push_back(v);
77      }
78
79      //Imprime la configuracion del emparejamiento.
80      cout << "Emparejamiento: " <<  HopcroftKarp() << '\n';
81      for (int u = 1; u <= U; ++u)
82          if (pairU[u])
83              cout << u << " - " << pairU[u] << '\n';
84
85      return 0;
86  }
```

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