

Grafos.

Material de referencia.

Índice

1. Caminos más cortos.	2
1.1. Algoritmo de Dijkstra	2
2. Árbol de expansión mínima.	4
2.1. Algoritmo de Kruskal	4
3. Orden topológico.	6
3.1. Algoritmo basado en DFS	6
4. Componentes fuertemente conexas.	8
4.1. Algoritmo de Tarjan	8
5. Máximo flujo.	10
5.1. Algoritmo de Edmonds-Karp	10
6. Emparejamiento máximo.	13
6.1. Algoritmo de Hopcroft-Karp	13

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.

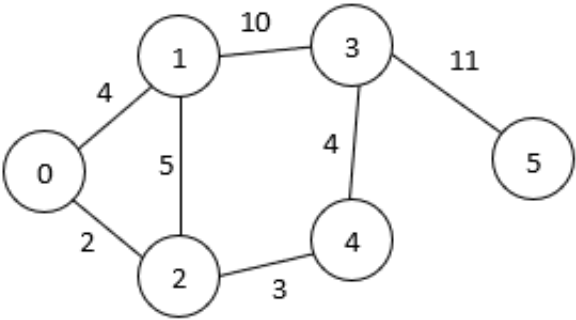
1.1. Algoritmo de Dijkstra

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

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <queue>
5  #include <utility>
6  using namespace std;
7
8  #define maxn 100000 //Maximo numero de vertices.
9
10 typedef pair<int, int> edge;
11 #define length first
12 #define to second
13
14 int V, E; //Numero de vertices y aristas.
15 vector<edge> graph[maxn]; //Aristas.
16
17 int s; //Vertice inicial.
18 int dist[maxn], pred[maxn]; //Distancia desde s y predecesor en el camino.
19 bool vis[maxn]; //Visitado.
20
21 //Encuentra el camino mas corto desde un vertice a todos los demas.
22 void Dijkstra() {
23     fill_n(dist, V, 1e9);
24     fill_n(pred, V, -1);
25     dist[s] = 0;
26
27     priority_queue<edge> pq;
28     pq.push(edge(dist[s], s));
29
30     while (!pq.empty()) {
31         int curr = pq.top().to;
32         pq.pop();
33         vis[curr] = true;
34
35         for (edge e : graph[curr])
36             if (!vis[e.to] && dist[curr] + e.length < dist[e.to]) {
37                 dist[e.to] = dist[curr] + e.length;
38                 pred[e.to] = curr;
39                 pq.push(edge(-dist[e.to], e.to));
40             }
41     }
42 }
43
44 int main() {
45     ios_base::sync_with_stdio(0); cin.tie();
46     cin >> V >> E;
47 }
```

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

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



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

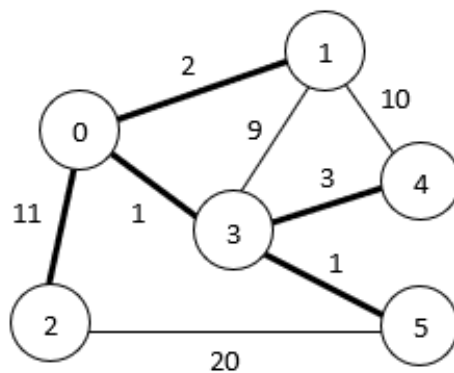
```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <utility>
5  using namespace std;
6
7  #define maxn 100000 //Maximo numero de vertices.
8
9  typedef pair<int , pair<int , int> > edge;
10 #define weight first
11 #define from second.first
12 #define to second.second
13
14 int V, E; //Numero de vertices y aristas.
15 edge graph[maxn]; //Aristas.
16
17 int parent[maxn], Rank[maxn]; //Union-Find por rango y compresion de camino.
18 vector<int> MST; //Arbol de expansion minima.
19
20 int Find(int x) {
21     if (parent[x] != x)
22         parent[x] = Find(parent[x]);
23     return parent[x];
24 }
25
26 bool Union(int x, int y) {
27     int a = Find(x), b = Find(y);
28     if (a == b)
29         return false;
30     else {
31         if (Rank[a] < Rank[b])
32             parent[a] = b;
33         else {
34             parent[b] = a;
35             if (Rank[a] == Rank[b])
36                 Rank[a]++;
37         }
38     }
39     return true;
40 }
41
42 //Encuentra el arbol de expansion minima.
43 int Kruskal() {
44     int W = 0;
45     for (int i = 0; i < V; ++i) {
46         parent[i] = i;
```

```

47     Rank[i] = 0;
48 }
49
50 sort(graph, graph + E);
51 for (int i = 0; i < E; ++i)
52     if (Union(graph[i].from, graph[i].to)) {
53         W += graph[i].weight;
54         MST.push_back(i);
55     }
56 return W;
57 }
58
59 int main() {
60     ios_base::sync_with_stdio(0); cin.tie();
61     cin >> V >> E;
62
63     //Lee la informacion de las aristas.
64     for (int i = 0; i < E; ++i)
65         cin >> graph[i].from >> graph[i].to >> graph[i].weight;
66
67     //Imprime la configuracion del arbol de expansion minima.
68     cout << "Peso total: " << Kruskal() << '\n';
69     for (int i : MST)
70         cout << graph[i].from << ' ' << graph[i].to << ' ' << graph[i].weight
71             << '\n';
72     return 0;

```

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



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

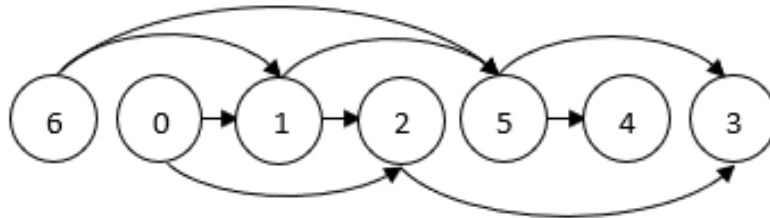
```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  using namespace std;
5
6  #define maxn 100000 //Maximo numero de vertices.
7
8  int V, E;           //Numero de vertices y aristas.
9  vector<int> graph[maxn]; //Aristas.
10
11 bool cycle;         //Verifica que no haya ciclos.
12 vector<int> sorted; //Orden topologico.
13 int vis[maxn];      //Visitado.
14
15 //Encuentra el orden topologico iniciando en un vertice dado.
16 void DFS(int u) {
17     if (vis[u] == 1) {
18         cycle = true;
19         return;
20     }
21     else if (!vis[u]) {
22         vis[u] = 1;
23         for (int v : graph[u])
24             DFS(v);
25         vis[u] = -1;
26         sorted.push_back(u);
27     }
28 }
29
30 //Encuentra el orden topologico.
31 void ToopologicalSort() {
32     for (int u = 0; u < V; ++u)
33         DFS(u);
34     reverse(sorted.begin(), sorted.end());
35 }
36
37 int main() {
38     ios_base::sync_with_stdio(0); cin.tie();
39     cin >> V >> E;
40
41     //Lee la informacion de las aristas.
42     for (int i = 0; i < E; ++i) {
43         int from, to;
44         cin >> from >> to;
45         graph[from].push_back(to);
46     }
47 }
```

```

48 //Imprime el orden topologico
49 ToopologicalSort();
50 if (cycle)
51     cout << "No es un DAG.\n";
52 else {
53     for (int u : sorted)
54         cout << u << ' ';
55     cout << '\n';
56 }
57
58 return 0;
59 }

```

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



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

```

1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <stack>
5  using namespace std;
6
7  #define maxn 100000 //Maximo numero de vertices.
8
9  int V, E;           //Numero de vertices y aristas.
10 vector<int> graph[maxn]; //Aristas.
11
12 vector<vector<int>> SCC; //Componentes fuertemente conexas.
13
14 int idx[maxn], low[maxn], lst_id; //Indices, ultimo indice.
15 stack<int> S;                    //Vertices pendientes.
16 bool onStack[maxn];              //Esta en la pila.
17
18 //Encuentra la componente fuertemente conexa de u.
19 void StrongConnect(int u) {
20     idx[u] = ++lst_id;
21     low[u] = lst_id;
22     S.push(u);
23     onStack[u] = true;
24
25     for (int v : graph[u]) {
26         if (!idx[v]) {
27             StrongConnect(v);
28             low[u] = min(low[u], low[v]);
29         }
30         else if (onStack[v])
31             low[u] = min(low[u], idx[v]);
32     }
33
34     if (low[u] == idx[u]) {
35         SCC.push_back(vector<int>());
36         while (S.top() != u) {
37             onStack[S.top()] = false;
38             SCC.back().push_back(S.top());
39             S.pop();
40         }
41
42         onStack[u] = false;
43         SCC.back().push_back(u);
44         S.pop();
45     }
46 }
47

```

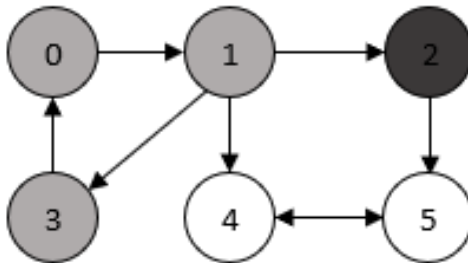


```

48 //Algoritmo de Tarjan para encontrar las componentes fuertemente conexas.
49 void Tarjan() {
50     for (int u = 0; u < V; ++u)
51         if (!idx[u])
52             StrongConnect(u);
53 }
54
55 int main() {
56     ios_base::sync_with_stdio(0); cin.tie();
57     cin >> V >> E;
58
59     //Lee la informacion de las aristas.
60     for (int i = 0; i < E; ++i) {
61         int from, to;
62         cin >> from >> to;
63         graph[from].push_back(to);
64     }
65
66     //Imprime las componentes fuertemente conexas.
67     Tarjan();
68     for (int i = 0; i < SCC.size(); ++i) {
69         for (int u : SCC[i])
70             cout << u << ' ';
71         cout << '\n';
72     }
73
74     return 0;
75 }

```

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



5. 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 .

5.1. Algoritmo de Edmonds-Karp

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

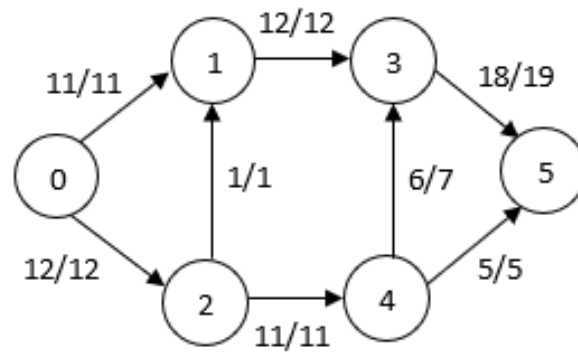
```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <queue>
5  using namespace std;
6
7  #define maxn 100000 //Maximo numero de vertices.
8  typedef int T;      //Tipo de dato del flujo.
9
10 struct edge {
11     int to;           //Destino.
12     T capacity, flow; //Capacidad, flujo.
13     edge *rev;        //Arista invertida.
14
15     edge(int _to, T _capacity, T _flow, edge *_rev) {
16         to = _to; capacity = _capacity; flow = _flow; rev = _rev;
17     }
18 };
19
20 int V, E;           //Numero de vertices y aristas.
21 int s, t;           //Fuente y sumidero.
22 vector<edge*> graph[maxn]; //Aristas.
23
24 //Calcula el flujo maximo de s a t.
25 T EdmondsKarp() {
26     T flow = 0;
27     edge *pred[maxn];
28
29     do {
30         //Realiza una BFS desde s hasta t.
31         queue<int> Q;
32         Q.push(s);
33         fill_n(pred, V, nullptr);
34
35         while (!Q.empty()) {
36             int curr = Q.front();
37             Q.pop();
38             for (edge *e : graph[curr])
```

```

39         if (pred[e->to] == nullptr && e->to != s && e->capacity > e->
40             flow) {
41             pred[e->to] = e;
42             Q.push(e->to);
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->
85                     capacity << '\n';
86             delete e;
87         }
88     return 0;
89 }

```

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



6. 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.

6.1. Algoritmo de Hopcroft-Karp

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

```

1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <queue>
5  using namespace std;
6
7  #define maxn 100000 //Maximo numero de vertices.
8
9  int U, V, E;          //Numero de vertices en cada lado y de aristas.
10 vector<int> graph[maxn]; //Aristas que van de U a V.
11
12 int pairU[maxn], pairV[maxn], dist[maxn]; //Pares de vertices en el
    emparejamiento.
13
14 //Verifica si existe un camino de aumento.
15 bool BFS() {
16     queue<int> Q;
17     for (int u = 1; u <= U; ++u) {
18         if (!pairU[u]) {
19             dist[u] = 0;
20             Q.push(u);
21         }
22         else
23             dist[u] = 1e9;
24     }
25     dist[0] = 1e9;
26
27     while (!Q.empty()) {
28         int u = Q.front();
29         Q.pop();
30         if (dist[u] < dist[0])
31             for (int v : graph[u])
32                 if (dist[pairV[v]] == 1e9) {
33                     dist[pairV[v]] = dist[u] + 1;
34                     Q.push(pairV[v]);
35                 }
36     }
37     return (dist[0] != 1e9);
38 }
39
40 //Verifica si existe un camino de aumento que comience en u.
41 bool DFS(int u) {
42     if (u) {
43         for (int v : graph[u])
44             if (dist[pairV[v]] == dist[u] + 1 && DFS(pairV[v])) {
45                 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     while (BFS())
60         for (int u = 1; u <= U; ++u)
61             if (!pairU[u] && DFS(u))
62                 size++;
63     return size;
64 }
65
66 int main() {
67     ios_base::sync_with_stdio(0); cin.tie();
68     cin >> U >> V >> E;
69
70     //Lee las aristas. Los vertices estan indexados en 1.
71     for (int i = 0; i < E; ++i) {
72         int u, v;
73         cin >> u >> v;
74         graph[u].push_back(v);
75     }
76
77     //Imprime la configuracion del emparejamiento.
78     cout << "Emparejamiento: " << HopcroftKarp() << '\n';
79     for (int u = 1; u <= U; ++u)
80         if (pairU[u])
81             cout << u << " - " << pairU[u] << '\n';
82
83     return 0;
84 }

```

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

