Algorithm 1 Flujo Maximo

1: **Ford-Fulkerson(G,s,t)**
2: **for** (cada arista (u,v) de E) **do**
3: $f[u, v] = 0$;
4: $f[v, u] = 0$;
5: **end for**
6: **while** (exista un camino p desde s a t en la red residual Gf) **do**
7: cf(p) = $min\{cf(u, v) : (u, v)\ esta\ sobre\ p\}$;
8: **for** (cada arista (u,v) en p) **do**
9: $f[u, v] = f[u, v] + cf(p)$;
10: $f[v, u] = -f[u, v]$;
11: **end for**
12: **end while**

Algorithm 2 Algoritmo de Dijkstra - Cola de prioridad

1: **DIJKSTRA**(Grafo G, nodo_fuente s)
2: **for** $u \in V[G]$ **do**
3: distancia $[u]$ = INFINITO
4: padre $[u]$ = NULL
5: distancia $[s] = 0$
6: adicionar (cola, (s,distancia[s]))
7: **end for**
8: **while** cola no es vacia **do**
9: u = extraer_mínimo(cola)
10: **end while**
11: **for** cada $v \in$ adyacencia $[u]$ **do**
12: **if** distancia $[v]$ > distancia $[u]$ + peso (u, v) **then**
13: distancia $[v]$ = distancia $[u]$ + peso (u, v)
14: padre $[v] = u$
15: adicionar(cola,(v, adicionar $[v]$))
16: **end if**
17: **end for**

Algorithm 3 Algoritmo de Dijkstra - Sin cola de prioridad

```
1: función Dijkstra (Grafo G, nodo_salida s)
2: //Usaremos un vector para guardar ñas distancias del nodo salida al resto entero distancia[n]
3: //Inicializamos el vector con distancias iniciales boolenao visto [n]
4: //Vector de boleanos para controlar los vértices de los que ya tenemos la distancia mínima
5: for cada w ∈ V[G] do
6:     if (no existe arista entre s y w) then
7:         distancia [w] = Infinito //puedes marcar la casilla con un −1 por ejemplo
8:     else if  then
9:         distancia [w] = peso (s, w)
10:    end if
11: end for
12: distancia[s] = 0
13: visto[s] = cierto
14: //n es el número de vértices que tiene el Grafo
15: while (no_esten_vistos_todos) do
16:    vértice = coger_el_mínimo_del_vector distancia y que no este vísto;
17:    vísto [vértice] = cierto;
18:    for cada w ∈ sucesores (G, vértice) do
19:        if distancia[w] > distancia[vértice]+peso (vértice, w) then
20:            distancia[w] = distancia[vértice]+peso (vértice, w)
21:        end if
22:    end for
23: end while
```

Algorithm 4 Algoritmo de Kruskal

```
1: función Kruskal(G)
2: for cada vértice v en G do
3:     Nuevo conjunto C(v) ← {v}
4:     Nuevo heap Q que contiene todas las aristas de G, ordenando por su peso.
5:     Defino un árbol T ← ∅
6:     //n es el número total de vértices
7: end for
8: while T tenga menos de n − 1 aristas y !Q.vacío() do
9:     (u, v) ← Q.sacarMín()
10:    //previene ciclos en T. agrega (u, v) si u y v están diferentes componentes en el conjunto.
11:    //Nótese que C(u) devuelve la componente a la que pertenece u.
12:    if C(v) ≠ C(u) then
13:        Agregar arista (v, u) a T.
14:        Merge C(v) y C(u) en el conjunto
15:    end if
16: end while
17: return árbol T
```

Algorithm 5 Algoritmo de Kruskal

```
1: Floyd-Warshall (G)
2: D = A ' matriz de distancias = matriz de arcos
3: if i=j o Dij= infinito then
4:     Pij= nulo
5: else if Pi,j =i ' matriz de caminos then
6: end if
7: for k=1 → V do
8:     for i=1 → V do
9:         for j=1 → V do
10:            Di,j=mín(Di,j, Di,k + Dk,j)
11:            if mín = Di,k + Dk,j then
12:                Pi,j = Pk,j
13:            end if
14:        end for
15:    end for
16: end for
```