Bellman-Ford

```
Bellman_Ford (G, weights, initial)
    for every vertex ∈ V
        λ [vertice] ← ∞
        π [vertice] ← null

    λ [initial] ← 0

    for i from 1 to | V | -1
        for every edge = (u, v) ∈ A
            if λ [v]> λ [u] + weights (u, v) # relaxation
                λ [v] ← λ [u] + weights (u, v)
                π [v] ← u
```

O algoritmo de Bellman-Ford executa em tempo O(V x E) onde V é o número de vértices e E o número de arestas.

Floyd-Warshall

```
ROTINA fw(Inteiro[1..n,1..n] grafo)
   # Inicialização
   VAR Inteiro[1..n,1..n] dist := grafo
   VAR Inteiro[1..n,1..n] pred
   PARA i DE 1 A n
      PARA j DE 1 A n
         SE dist[i,j] < Infinito ENTÃO
            pred[i,j] := i
   # Laço principal do algoritmo
   PARA k DE 1 A n
      PARA i DE 1 A n
         PARA j DE 1 A n
            SE dist[i,j] > dist[i,k] + dist[k,j] ENTÃO
               dist[i,j] = dist[i,k] + dist[k,j]
               pred[i,j] = pred[k,j]
   RETORNE dist
```

Complexidade de O(V^3).

Dijkstra

```
para todo v ∈ V[G]
    d[v] ← ∞
    π[v] ← -1
d[s] ← 0
Q ← V[G]
enquanto Q ≠ ø
        u ← extrair-mín(Q)                    //Q ← Q - {u}
```

```
para cada v adjacente a u
    se d[v] > d[u] + peso(u, v)          //relaxe (u, v)
      então d[v] ← d[u] + peso(u, v)
          π[v] ← u
```

Grafos sem ciclos com peso negativo em uma complexidade de tempo $O(V^3)$.

Ford-Fulkerson

```
função Atualiza-Grafo-Residual(G, f)
   Para cada aresta a(u,v) em G, , com u,v∈N
        Se f(a) < ca então
           insira aR(u,v) com caR=(ca - f(a))
        Se f(a) > 0 então
           insira aR(v,u) com caR=f(a)
    Retorna(GR)

função Ford-Fulkerson(G, s, t)
  Inicia f(a)=0 para cada aresta a de G
  Defina GR = Atualiza-Grafo-Residual(G, f)
  Enquanto existir caminho de aumento de s para t em GR
      Seja P um caminho de aumento s-t em GR
      Defina cP = min{caR : aR∈P}
      Para cada aresta aR em P
         Se aR tem direção s-t então
             faça [f(a) → f(a) + cP] em G
        Caso contrário
            faça [f(a) → f(a) - cP] em G
      GR = Atualiza-Grafo-Residual(G, f)
 Retorna (f)
```

A complexidade do algoritmo é $O(mf)$, em que m representa o número de arestas presentes no grafo G e f o fluxo máximo encontrado.

Push–relabel

```
push(f, h, v, w) .                              e(v) > 0, h(w) < h(v) e (v, w) ∈ Gf
se (v, w) é uma aresta direta de Gf então
        e ← (v, w)
        ε ← min(e(v), u(e) − f(e))
        f(e) ← f(e) + ε
se (v, w) é uma aresta inversa de Gf então
        e ← (w, v)
        ε ← min(e(v), f(e))
        f(e) ← f(e) − ε
devolva (f, h)
```

```
relabel(f, h, v) .          e(v) > 0 e h(w) ≥ h(v) para toda aresta (v, w) ∈ Gf
        h(v) ← h(v) + 1
        devolva (f, h)

PushRelabel(G, s, t)
para cada v em V faça h(v) ← 0
h(s) ← |V |
para cada e ← (v, w) em Gf faça
        se v = s então f(e) ← u(e)
        senão f(e) ← 0
enquanto existe vértice v 6= t com e(v) > 0 faça
        seja v um vértice com excesso positivo
        se existe aresta (v, w) com h(w) < h(v) então push(f, h, v, w)
        senão relabel(f, h, v)
devolva f
```

O algoritmo push-relabel é um dos algoritmos de fluxo máximo mais eficientes. O algoritmo genérico tem uma complexidade de tempo O (V^2 E).

Busca em profundidade

```
Busca-em-Profundidade (n, Adj, r)
para  u ← 1  até  n  faça
        cor[u] ← branco
cor[r] ← cinza
P ← Cria-Pilha (r)
enquanto  P  não estiver vazia faça
        u ← Copia-Topo-da-Pilha (P)
        v ← Próximo (Adj[u])
        se  v ≠ nil
                então  se cor[v] = branco
                        então  cor[v] ← cinza
                                Coloca-na-Pilha (v, P)
                senão  cor[u] ← preto
                        Tira-da-Pilha (P)
devolva  cor[1..n]
```

Complexidade: O(m+n)

Shortest-Path

```
Shortest-Path-Main (W)
L = W
Para i = 2 até n
        L = Shortest-Path(L,W)
Retorne L
```

```
Shortest-Path (L(m), W)
Cria L(m+1) = inf
Para todo i pertencente a V
        Para todo j pertencente a V
                c = 0
                Para todo k pertencente a V
                        c += L(m)[i,k] + W[k,i]
                L(m+1)[i,j] = c
Retorne L(m+1)
```

Complexidade: O(n^n)