UFC – UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS SOBRAL

Curso de Engenharia da Computação

Disciplina: Algoritmos em Grafos

Pseudocódigos

Equipe:

Gerônimo Pereira Aguiar - 385145

Pedro Renoir Silveira Sampaio - 389113

Samuel Hericles Souza Silveira - 389118

Sobral - CE 2019.2

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).

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)

Extend-Shortest-Path

```
EXTEND-SHORTEST-PATH-MOD(G, L, W)
n = L.row
L' = l'[i, j] é uma nova matriz nxn
G' = π'[i, j] se é uma nova matriz nxn
for i = 1 to n
    for j = 1 to n
      l'[i, j] = ∞
      π'[i, j] = null
      for k = 1 to n
        if l[i, k] + l[k, j] < l[i, j]
          l[i, j] = l[i, k] + l[k, j]
          if k != j
            π'[i, j] = k
          else
            π'[i, j] = π[i, j]
return (G', L')
```

```
SLOW-ALL-PAIRS-SHORTEST-PATHS-MOD(W)
n = W.rows
L(1) = W
G(1) = π[i, j](1) onde π[i, j](1) = i se houver uma aresta de i a j, e null caso contrário
for m = 2 to n - 1
    G(m), L(m) = EXTEND-SHORTEST-PATH-MOD(G(m - 1), L(m - 1), W)
return (G(n - 1), L(n - 1))
```

(livro)

```
EXTEND-SHORTEST-PATH(L,W)
n = linhas[L]
seja L’ = (l’) uma matriz nxn
for i to n
     do for j = 1 to n
          do l’ = inf
               for k = 1 to n
                    do l = min(l’, l + w)
```

retorne L’

Execução: $O(n^3)$

```
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
n = linhas[W]
L¹ = W0
for m = 2 to n - 1
        do L(m) = EXTEND-SHORTEST-PATH(L(m-1),W)
retorne L(n-1)
```

```
FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
n = linhas[W]
L¹ = W
 m = 1
while m < n - 1
        do L(2m) = EXTEND-SHORTEST-PATH(L(m),L(m))
                m = 2m
retorne L(m)
```