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
```

Funcionamento:
Como o algoritmo Dijkstra, o algoritmo Bellman-Ford usa a técnica de relaxamento, ou seja, realiza sucessivas aproximações das distâncias até finalmente chegar na solução. A principal diferença entre Dijkstra e Bellman-Ford é que no algoritmo de Dijkstra é utilizada uma fila de prioridades para selecionar os vértices a serem relaxados, enquanto o algoritmo de Bellman-Ford simplesmente relaxa todas as arestas. G é um grafo no formato G = (V,A), π[v] indica o predecessor de v, λ[v] é o custo de sair da origem e chegar até o vértice v e inicial é o vértice inicial. Após o término do algoritmo, para cada v pertencente aos vértices de G, π[v] → y representa uma aresta selecionada para a árvore geradora mínima (se y ≠ nulo) e, pensando em árvore, λ[v] representa a distância da raiz até o vértice v.

Complexidade:
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
```

Funcionamento:
O algoritmo de Floyd-Warshall recebe como entrada uma matriz de adjacência que representa um grafo (V,E) orientado e valorado. O valor de um caminho entre dois vértices é a soma dos valores de todas as arestas ao longo desse caminho. As arestas E do grafo podem ter valores negativos, mas o grafo não pode conter nenhum ciclo de valor negativo. O algoritmo calcula, para cada par de vértices, o menor de todos os caminhos entre os vértices. Por exemplo, o caminho de menor custo.Assumindo que os vértices de um grafo orientado G são V= 1,2,3... n, considere um subconjunto 1,2,3, ... ,k. Para qualquer par de vértices(i,j) em V, considere todos os caminhos de i a j cujos vértices intermédios pertencem ao subconjunto 1,2,3, ... ,k, e p como o mais curto de todos eles. O algoritmo explora um relacionamento entre o caminho p e os caminhos mais curtos de i a j com todos os vértices intermédios em 1,2,3, ... ,k-1. O relacionamento depende de k ser ou não um vértice intermédio do caminho p.

Complexidade:
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)
```

Funcionamento:
O algoritmo de Ford-Fulkerson, também conhecido como algoritmo dos pseudocaminhos aumentadores, resolve o problema do fluxo máximo. Cada iteração começa com um fluxo f que respeita as capacidades dos arcos. A primeira iteração começa com o fluxo nulo. O processo iterativo consiste no seguinte: enquanto existe pseudocaminho aumentador, encontra um pseudocaminho aumentador P, calcule a capacidade residual δ de P, envie δ unidades de fluxo ao longo de P e atualiza f .

Complexidade:
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
```

Funcionamento:

Para calcular o fluxo descrito em 2, o algoritmo utiliza duas funções principais: push e relabel. Tais funções mantém os seguintes dados:

Fluxo de u para v e f(u,v). A capacidade disponível é dada por c(u,v) - f(u,v). Height(u). Nós chamamos a função push de u para v apenas se height(u) > height(v). Para todos u, height(u) é um inteiro positivo. excess(u). Soma do fluxo de e para u.

Após cada passo do algoritmo, teremos um pré-fluxo que deve satisfazer as seguintes condições:

f(u,v) < c(u,v). O fluxo entre u e v não excede a capacidade.

f(u,v) = - f(v,u). Mantemos o fluxo da rede.

Σf(v,u) = express(u) >= 0 para todos os nós u ≠ s. Somente a fonte pode produzir fluxo.

Complexidade:

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]
```

Funcionamento:
Formalmente, um algoritmo de busca em profundidade realiza uma busca não-informada que progride através da expansão do primeiro nó filho da árvore de busca, e se aprofunda cada vez mais, até que o alvo da busca seja encontrado ou até que ele se depare com um nó que não possui filhos (nó folha). Então a busca retrocede (backtrack) e começa no próximo nó. Numa implementação não-recursiva, todos os nós expandidos recentemente são adicionados a uma pilha, para realizar a exploração. Quando ocorrem buscas em grafos muito grandes, que não podem ser armazenadas completamente na memória, a busca em profundidade não termina, em casos onde o comprimento de um caminho numa árvore de busca é infinito. O simples artifício de “lembrar quais nós já foram visitados” não funciona, porque pode não haver memória suficiente. Isso pode ser resolvido estabelecendo-se um limite de aumento na profundidade da árvore.

Complexidade:
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)
```

Funcionamento:
Defina todas as distâncias dos vértices = infinito, exceto o vértice da fonte, defina a distância da fonte = 0. Empurre o vértice de origem em uma fila de prioridade mínima no formato (distância, vértice), pois a comparação na fila de prioridade mínima será de acordo com as distâncias dos vértices. Define o vértice com a distância mínima da fila de prioridade (primeiro o vértice estourado = origem). Atualize as distâncias dos vértices conectados ao vértice estourado no caso de "distância atual do vértice + peso da borda <distância do próximo vértice" e empurre o vértice com a nova distância da fila de prioridade. Se o vértice popped já tiver sido

visitado antes, continue sem usá-lo. Aplique o mesmo algoritmo novamente até que a fila de prioridade esteja vazia.

Complexidade:
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)
```