UNIVERSIDADE FEDERAL DO CEARÁ
ENGENHARIA DA COMPUTAÇÃO

ALGORITMOS EM GRAFOS

TRABALHO FINAL

HUGO SOUSA SILVEIRA
378998

LUCINARA FERNANDES
385127

RENAN HENRIQUE CARDOSO
379013

SOBRAL
2019

SUMÁRIO

INTRODUÇÃO

Para a implementação dos algoritmos, foi utilizada a linguagem python com funções e bibliotecas básicas. As bibliotecas usadas foram:

- *Numpy* - Manipulação de matrizes
- *Math* - Atribuição do valor equivalente a “infinito”.

Algumas funções foram implementadas a priori para serem usadas no decorrer do código:

- calcArestas - Recebe um arquivo no formato de .txt que segue o modelo de entrada proposto, e a partir do texto lido, é selecionado apenas os índices das arestas e seus respectivos pesos, ou seja, a matriz inicial sem as duas primeiras linhas.

```
calcArestas()
arquivo <- Recebe arquivo .txt com grafo
texto <- Vetor de zeros de tamanho (n)
arestas <- texto[2 até fim]
retorna arestas
```

- matrizPesos - Seleciona o arquivo de entrada e transforma em uma matriz de adjacência

```
matrizPesos()
arquivo <- Recebe arquivo .txt com grafo
nVertices <- Valor do elemento das primeiras
                                 linha e coluna
arestas <- texto[2 até fim]
w <- Matriz de zeros de tamanho (nVertices)
Para cada elemento com índices i != j:
  w[i,j] <- ∞
Para cada linha i em arestas:
  j <- Cada elemento da linha
  w[j[0],j[1]] <- Valor dos pesos
retorna w
```

- raiz - Utilizada nos Problemas de Caminhos Mínimos onde retorna um único vértice explicitando ao qual será impresso os menores caminhos;

```
raiz()
arquivo <- Recebe arquivo .txt com grafo
vPai <- Valor do elemento da primeira coluna e
                                 segunda linha
retorna vPai
```

- fonte_sorvedouro - Utilizada nos Problemas de Fluxo onde retorna dois valores referentes ao par de vértices, separados por espaço onde o primeiro vértice é a fonte e o segundo o sorvedouro, presentes na entrada;

```
fonte_sorvedouro()
 arquivo <- Recebe arquivo .txt com grafo
 f <- Valor do elemento da primeira coluna e
                                        segunda linha (fonte)
 s <- Valor do elemento da segunda coluna e
                                   segunda linha (sorvedouro)
retorna f, s
```

DESENVOLVIMENTO

Parte 1 - Caminhos Mínimos

1. Bellman-Ford

1.1. Resumo de funcionamento

O algoritmo passa por três fases principais. A primeira é inicializar as distâncias, ou seja, toda distância do vértice *raiz* a outro seja infinito e para ele mesmo seja 0 e o pai de cada vértice é configurado sendo -1, à exceção de *raiz*, pois seu pai é a própria *raiz*. A segunda fase é o relaxamento das arestas, que tem como objetivo testar e substituir um caminho mais curto entre o vértice v_1 e outro vértice v_2 do grafo. A última fase é feito novamente o relaxamento de todos os vértices, se ainda assim houver um caminho melhor entre algum par de vértices, então é constatado que este grafo possui um ciclo negativo.

1.2. Pseudo Código

```
bellmanFord( W , raiz )
g2 <- inicializa( W )
arestas <- calcArestas()
n <- | W |
pai <- Matriz de zeros com tamanho (n,n)
pai_v <- pai
para i <- 0 até n faça
  pai[i] <- -1
pai[ raiz ] <- raiz
para v <- 0 até n faça
  para i <- 0 até n-1 faça
    para row em arestas faça
      j <- split(row)*
      relax(v, j[0], j[1], g2, W , pai)
  para row em arestas faça
    j <- split(row)*
    se relax(v, j[0], j[1], g2, W , pai) faça
      exibe("Têm ciclo negativo")
      retorna Falso
  se v = raiz faça
    pai_v <- pai
  pai <- Vetor de zeros de tamanho (n)
  para i <- 0 até n faça
     pai[i] <- -1
  pai[ raiz ] <- raiz
exibe(g2)
exibe("Caminhos Mínimos")
exibeCaminhos(g2, pai_v, raiz)
retorna True
```

*A função split(obj) retorna o parâmetro objeto em forma de lista, onde cada posição é um conjunto de caracteres do tipo string e que não possui espaços vazios. Exemplo:

```
split("Hoje é um belo dia!") =>
[[Hoje], [é], [um], [belo], [dia]]
```

Funções auxiliares do algoritmo de Bellman-Ford:

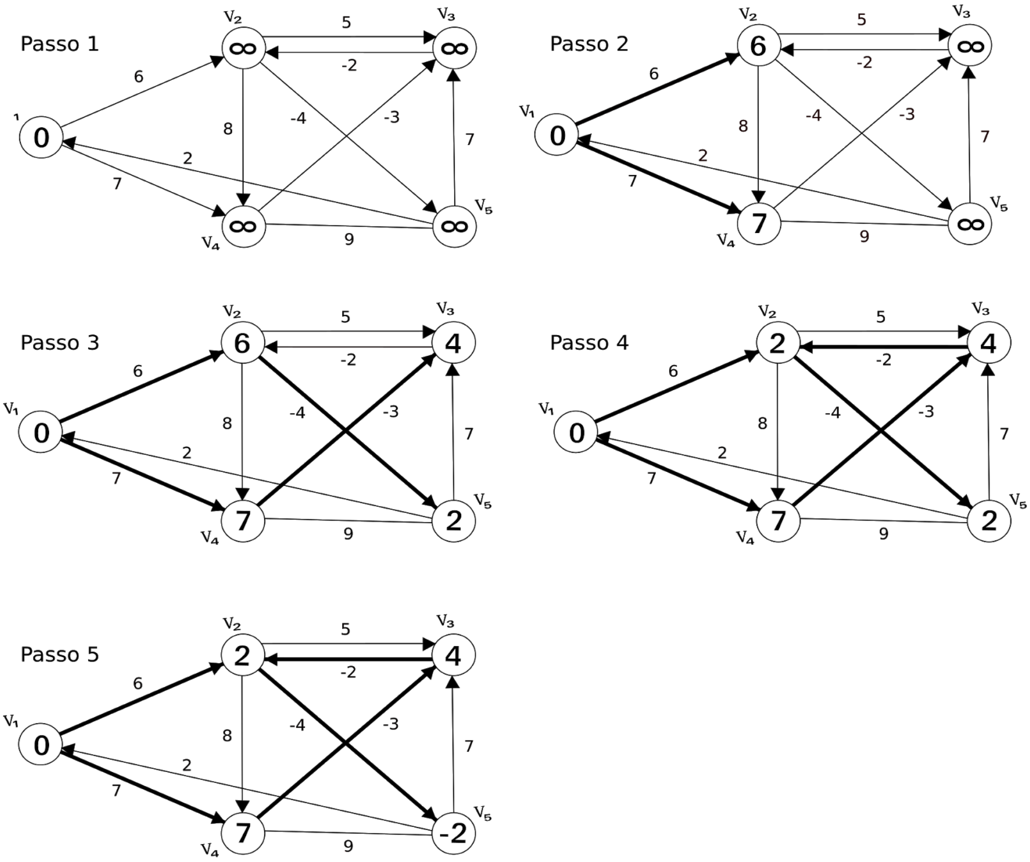
```
inicializa(W)
G <- W
n <- |G|
para i <- 0 até n faça
  para j <- 0 até n faça
    se i ≠ j faça:
      G[i,j] <- ∞
retorna G
```

```
exibeCaminhos(g2, pai_v, raiz)
n <- |g2|
path <- Vetor vazio
path_inv <- Vetor vazio
para v <- 1 até n faça
  path <- Esvazia vetor
  pai_atual <- pai_v
  Adiciona v ao final de path
  enquanto (pai_atual ≠ raiz) faça
    Adiciona pai_path[v] ao final de path
    pai_atual <- pai_v[v]
  v <- pai_atual
  path_inv <- path com sequência invertida
  exibe(path_inv[0])
  m <- |path_inv|
  para i <- 1 até m faça
    exibe("->")
    exibe(path_inv[i])
```

```
relax(s, u, v, g2, W, pai)
se g2[s,v] > (g2[s,u] + W[u,v]) faça:
  g2[s,v] <- g2[s,u] + W[u,v]
  pai[v] = u
  retorna 1
retorna 0
```

1.3. Complexidade: O (n^2 . m)

1.4. Exemplo:



Fonte: Elaborada pelos autores.

2. Floyd-Warshall

2.1. Resumo de funcionamento

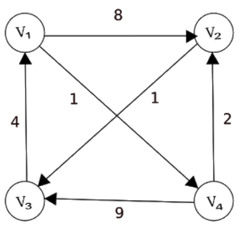
O algoritmo recebe como entrada uma matriz de adjacência e conta com três *loops* de repetição, o grafo não pode conter nenhum ciclo de valor negativo. Seu objetivo é calcular, para cada par de vértices, o caminho de menor custo dentre todos os possíveis.

2.2. Pseudo Código

```
floydWarshall( W )
d <- W
n <- | W |
pai <- Matriz de zeros com tamanho (n,n)
pai_v <- pai
para k <- 0 até n faça
  para i <- 0 até n faça
    para j <- 0 até n faça
      se (d[i,j] > (d[i,k] + d[k,j]) faça
        d[i,j] = d[i,k] + d[k,j]
exibe(d)
```

2.3. Complexidade: O (n^3)

2.4. Exemplo:



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & inf & 1 \\ 2 & inf & 0 & 1 & inf \\ 3 & 4 & inf & 0 & inf \\ 4 & inf & 2 & 9 & 0 \end{array} \Rightarrow D_0$$

$$D_1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & inf & 1 \\ 2 & inf & 0 & 1 & inf \\ 3 & 4 & 12 & 0 & 5 \\ 4 & inf & 2 & 9 & 0 \end{array}$$

$$D_2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & 9 & 1 \\ 2 & inf & 0 & 1 & inf \\ 3 & 4 & 12 & 0 & 5 \\ 4 & inf & 2 & 9 & 0 \end{array}$$

$$D_3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & 9 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 12 & 0 & 5 \\ 4 & inf & 2 & 9 & 0 \end{array}$$

$$D_4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 4 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 7 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{array}$$

Fonte: Elaborada pelos autores.

3. Shortest Path (Menor Rec)

3.1. Resumo de funcionamento: idem *Shortest Path* (4.1).

3.2. Pseudo Código

```
menorRecSTP( W )
l <- W
n <- |W|
para i <- 0 até n faça
  para j <- 0 até n faça
    l2 <- W
    l[i,j] = calcSTP(l2, i, j, n)
exibe(l)
```

```
calcSTP( W , i, j, n)
se i = j
  retorna 0
se m = i
  retorna w[i,j]
c <- ∞
n <- |W|
para k <- 0 até n faça
  se (c > calcSTP( W , i, k, m-1)) faça
    c = calcSTP(w, i, k, m-1) + w[k,j]
retorna c
```

3.3. Complexidade: O(n^3 . m).

3.4. Exemplo: idem *Shortest Path* (4.4).

4. Shortest Path

4.1. Resumo de funcionamento

O algoritmo se baseia na criação de matrizes intermediárias L, onde cada matriz L é calculada com a matriz L anterior, cada nova matriz apresenta menores caminhos do que a anterior.

4.2. Pseudo Código

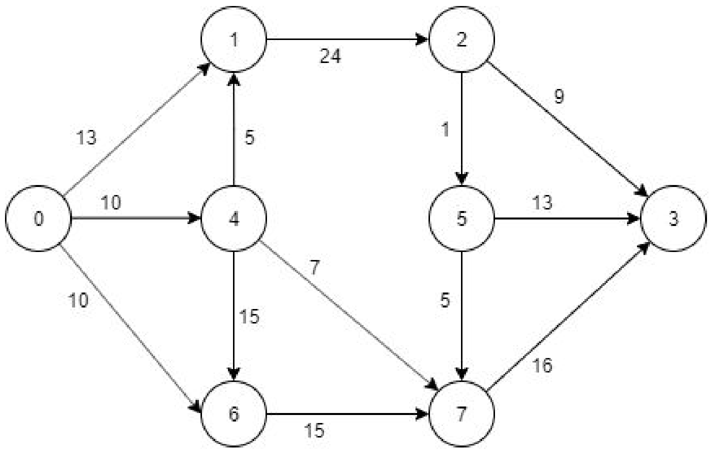
```
STP(l)
nVertices <- |l|
l2 <- Matriz de zeros com tamanho
(nVertices,nVertices)
para i <- 0 até nVertices faça
  para j <- 0 até nVertices faça
    se i ≠ j faça
      l2[i,j] <- ∞
para i <- 0 até nVertices faça
  para j <- 0 até nVertices faça
```

```
        c = l[i,j]
        para k até nVertices faça
          se c > (l[i,k] + l[k,j]) faça
             c = l[i,k] + l[k,j]
        l[i,j] = c
    retorna l

mainSTP( W )
    l <- W
    para i <- 1 até |W| faça
      l = STP(l)
    exibe(l)
```

4.3. Complexidade: O (n^3. log(n))

4.4. Exemplo:

Dado o grafo a seguir, e usando o vértice 0 como raiz, obtemos as seguintes matrizes com o algoritmo de Shortest Path.



Matriz Intermediária 1:

[[0 13 37 46 10 38 10 17]
 [∞ 0 24 33 ∞ 25 ∞ 31]
 [∞ ∞ 0 9 ∞ 1 ∞ 7]
 [∞ ∞ ∞ 0 ∞ ∞ ∞ ∞]
 [∞ 5 29 23 0 30 15 7]
 [∞ ∞ ∞ 13 ∞ 0 ∞ 6]
 [∞ ∞ ∞ 31 ∞ ∞ 0 15]
 [∞ ∞ ∞ 16 ∞ ∞ ∞ 0]]

Matriz Intermediária 2:

[[0 13 37 33 10 38 10 17]
 [∞ 0 24 33 ∞ 25 ∞ 31]
 [∞ ∞ 0 9 ∞ 1 ∞ 7]
 [∞ ∞ ∞ 0 ∞ ∞ ∞ ∞]
 [∞ 5 29 23 0 30 15 7]
 [∞ ∞ ∞ 13 ∞ 0 ∞ 6]
 [∞ ∞ ∞ 31 ∞ ∞ 0 15]
 [∞ ∞ ∞ 16 ∞ ∞ ∞ 0]]

Matriz Intermediária 3:

```
[[ 0  13  37  33  10  38  10  17 ]
 [ ∞  0  24  33  ∞  25  ∞  31 ]
 [ ∞  ∞  0  9  ∞  1  ∞  7 ]
 [ ∞  ∞  ∞  0  ∞  ∞  ∞  ∞ ]
 [ ∞  5  29  23  0  30  15  7 ]
 [ ∞  ∞  ∞  13  ∞  0  ∞  6 ]
 [ ∞  ∞  ∞  31  ∞  ∞  0  15 ]
 [ ∞  ∞  ∞  16  ∞  ∞  ∞  0 ]]
```

Matriz Intermediária 4:

```
[[ 0  13  37  33  10  38  10  17 ]
 [ ∞  0  24  33  ∞  25  ∞  31 ]
 [ ∞  ∞  0  9  ∞  1  ∞  7 ]
 [ ∞  ∞  ∞  0  ∞  ∞  ∞  ∞ ]
 [ ∞  5  29  23  0  30  15  7 ]
 [ ∞  ∞  ∞  13  ∞  0  ∞  6 ]
 [ ∞  ∞  ∞  31  ∞  ∞  0  15 ]
 [ ∞  ∞  ∞  16  ∞  ∞  ∞  0 ]]
```

Matriz Intermediária 5:

```
[[ 0  13  37  33  10  38  10  17 ]
 [ ∞  0  24  33  ∞  25  ∞  31 ]
 [ ∞  ∞  0  9  ∞  1  ∞  7 ]
 [ ∞  ∞  ∞  0  ∞  ∞  ∞  ∞ ]
 [ ∞  5  29  23  0  30  15  7 ]
 [ ∞  ∞  ∞  13  ∞  0  ∞  6 ]
 [ ∞  ∞  ∞  31  ∞  ∞  0  15 ]
 [ ∞  ∞  ∞  16  ∞  ∞  ∞  0 ]]
```

Matriz Intermediária 6:

```
[[ 0  13  37  33  10  38  10  17 ]
 [ ∞  0  24  33  ∞  25  ∞  31 ]
 [ ∞  ∞  0  9  ∞  1  ∞  7 ]
 [ ∞  ∞  ∞  0  ∞  ∞  ∞  ∞ ]
 [ ∞  5  29  23  0  30  15  7 ]
 [ ∞  ∞  ∞  13  ∞  0  ∞  6 ]
 [ ∞  ∞  ∞  31  ∞  ∞  0  15 ]
 [ ∞  ∞  ∞  16  ∞  ∞  ∞  0 ]]
```

Matriz de Menores Caminhos:

```
[[ 0  13  37  33  10  38  10  17 ]
 [ ∞  0  24  33  ∞  25  ∞  31 ]
 [ ∞  ∞  0  9  ∞  1  ∞  7 ]
 [ ∞  ∞  ∞  0  ∞  ∞  ∞  ∞ ]
 [ ∞  5  29  23  0  30  15  7 ]
 [ ∞  ∞  ∞  13  ∞  0  ∞  6 ]
 [ ∞  ∞  ∞  31  ∞  ∞  0  15 ]
 [ ∞  ∞  ∞  16  ∞  ∞  ∞  0 ]]
```

Parte 2 - Problema de Fluxo Máximo

1. Ford-Fulkerson

1.1. Resumo de funcionamento

O algoritmo de *Ford-Fulkerson* busca encontrar o fluxo máximo entre uma fonte *f* e um sorvedouro *s*. Enquanto houver um caminho que conecta *f* a *s*, o algoritmo pega o maior desses caminhos. Nesta implementação foi utilizada uma variação do algoritmo de *Bellman-Ford*, que busca o maior caminho de uma raiz, que é *f*, até *s*. Ao encontrá-lo, é necessário buscar a menor aresta desse caminho. Depois, de cada aresta do caminho é subtraído o valor da menor aresta, gerando uma nova matriz de pesos, o valor da menor aresta é acumulado em um contador; esse processo é repetido até não haver mais caminhos de *f* a *s*. Então, o contador terá o valor do fluxo máximo.

1.2. Pseudo Código

```
inicializa_max( W )
G <- W
n <- |G|
para i <- 0 até n faça
```

```
  para j <- 0 até n faça
    se i ≠ j faça:
      G[i,j] <- -1
retorna G
```

```
relax_inv(s, u, v, g2, W, pai)
se g2[s,v] < (g2[s,u] + W[u,v]) faça:
  g2[s,v] <- g2[s,u] + W[u,v]
  pai[v] = u
  retorna 1
retorna 0
```

```
exibeCaminhos_max(g2, pai_v, raiz, s)
n <- |g2|
path <- Vetor vazio
path_inv <- Vetor vazio
para v <- 1 até n faça
  path <- Esvazia vetor
  pai_atual <- -1
  Adiciona v ao final de path
  enquanto (pai_atual ≠ raiz) faça
    Adiciona pai_v[v] ao final de path
    pai_atual <- pai_v[v]
    v <- pai_atual
  path_inv <- path com sequência invertida
  m <- |path_inv|
  se última posição de path_inv = s faça
    retorna path_inv
```

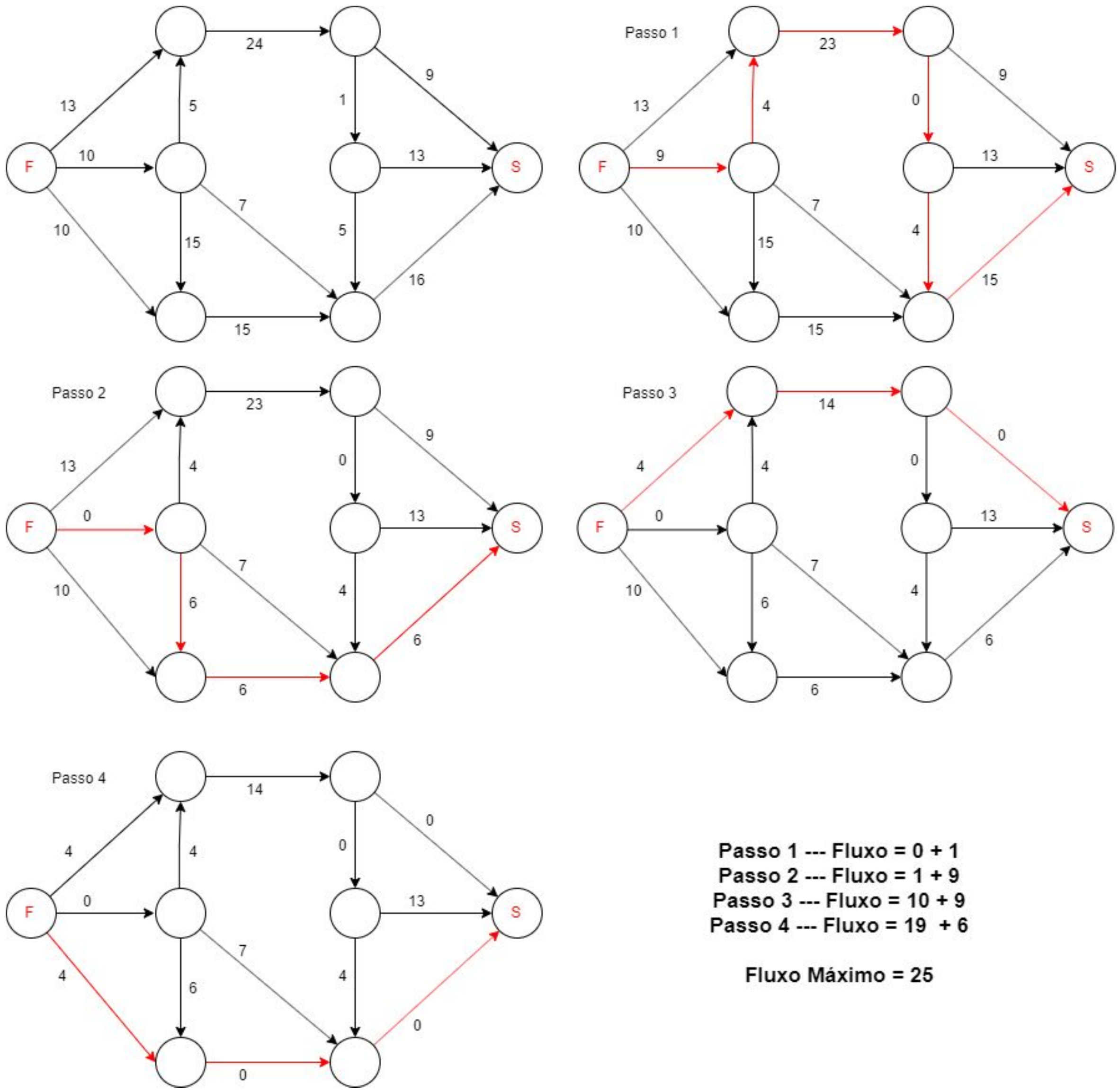
```
bellmanFord_max(W, raiz, s)
g2 <- inicializa(W)
arestas <- calcArestas()
n <- |W|
pai <- Matriz de zeros com tamanho (n,n)
pai_v <- pai
para i <- 0 até n faça
  pai[i] <- -1
pai[raiz] <- raiz
para v <- 0 até n faça
  para l <- 0 até n faça
    para c <- 0 até n faça
      se W[l,c] != infinito e != 0 faça
        relax(v, l, c, g2, W, pai)
  se v = raiz faça
    pai_v <- pai
```

```
  pai <- Vetor de zeros de tamanho (n)
  para i <- 0 até n faça
     pai[i] <- -1
  pai[raiz] <- raiz
 p = exibeCaminhos_max(g2, pai_v,raiz,s)
 retorna p
```

```
fordFurkelson(W,f,s):
w2 <- w
fluxo = 0
enquanto Verdadeiro
  path = bellmanFord_Max(w2, f, s)
  arestas_path = Vetor vazio
  para i<-0 até |path| - 1 faça
    add w2[path[i],path[i+1]] em arestas_path
  minimo = min(arestas_path)
  fluxo += minimo
  para i<-0 até |path| - 1 faça
    w2[path[i],path[i+1]] -= minimo
  se existir -1 em path
    para
  exiba(Grafo de Resíduo = w2)
  exiba(Fluxo Max = fluxo)
```

1.3. Complexidade: O (m . |flu|)

1.4. Exemplo: (próxima página)



2. Push–relabel maximum flow algorithm

2.1. Resumo de funcionamento

O algoritmo apresenta quatro funções principais: *inicializaPR*, *push*, *relabel* e *push_relabel*. A função *inicializaPR* cria um pré-fluxo inicial, coloca tanto como 0 o excesso de todos os vértices quanto todas as alturas para 0, menos do vértice fonte que recebe o número de vértices do grafo como sua altura. A função *push* é aplicada quando o vértice está transbordando, quando o fluxo é maior que 0 e altura do vértice é igual à altura do próximo vértice mais 1; essa função altera os fluxos e os excessos dos vértices. A função *relabel* é aplicada sempre que o vértice esteja transbordando e a altura do vértice seja maior ou igual a altura do próximo vértice; tem como objetivo aumentar a altura do vértice atual. A função *push_relabel* é a função que chama as outras funções enquanto houver a possibilidade de realizar as operações de *push* e *relabel*.

2.2. Pseudo Código

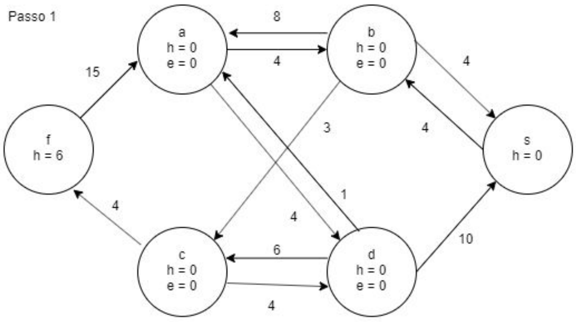
```
inicializaPR(w,f):
h <- vetor de zeros de tamanho |w|
e <- vetor de zeros de tamanho |w|
c <- w
flu <- w
para l<-0 até |w| faça
  para j<-0 até |w| faça
    flu[l,j] <- 0
para l<-0 até |w| faça
  para j<-0 até |w| faça
    se w[l,j] = infinito faça
      c[l,j] <- 0
h[f] = |w|
e[f] = infinito
para l<-0 até |w| faça
  se c[f,l] != 0 faça
    flu[f,l] = c[f,l]
    flu[l,f] = -c[f,l]
    e[l] = c[f,l]
    e[f] = e[f] - c[f,l]
retorna h, e, flu, c

push(u,v,e,c,flu):
cf = c[u,v] - flu[u,v]
d = min(e[u],cf)
flu[u,v] = flu[u,v] + d
flu[v,u] = -flu[u,v]
e[u] = e[u] - d
e[v] = e[v] + d

relabel(u,c,flu,h,w):
m = infinito
para i<-0 até |w| faça
  cf = c[u,i] - flu[u,i]
  se cf>0 faça
    se h[i]<m:
      m = h[i]
h[u] = 1 + m

push_relabel(w,f,s):
h, e, flu, c = inicializaPR(w,f)
para j<-0 até |w|*|w| faça
  para u<-0 até |w| faça
    se e[u]>0 e u!=f e u!=s faça
      relabel(u,c,flu,h,w)
      para v<-0 até |w| faça
```

```
    cf = c[u,v] - flu[u,v]
    se cf != 0 faça
     se h[u] = h[v]+1 faça
      push(u,v,e,c,flu)
exiba(Resíduo)
exiba(c - flu)
fluxo = 0
para i<-0 até |w| faça
 fluxo += flu[i,s]
exiba(Fluxo Máximo: fluxo)
```

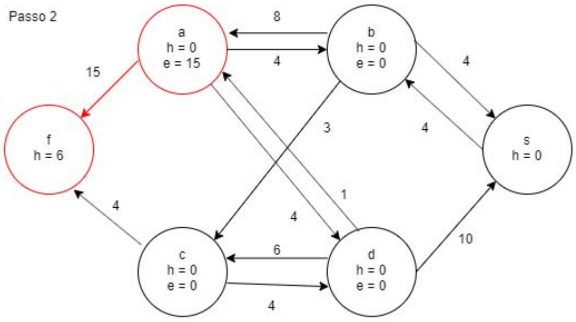
2.3. Complexidade: O (n² . m)

2.4. Exemplo: (próxima página)

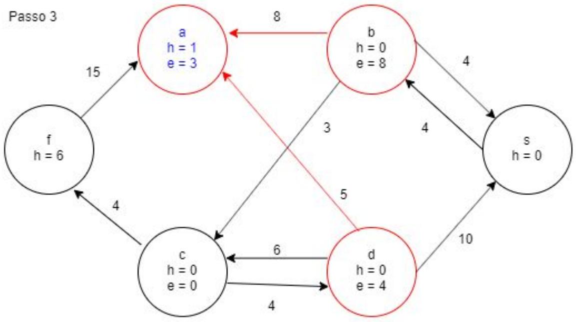
Passo 1



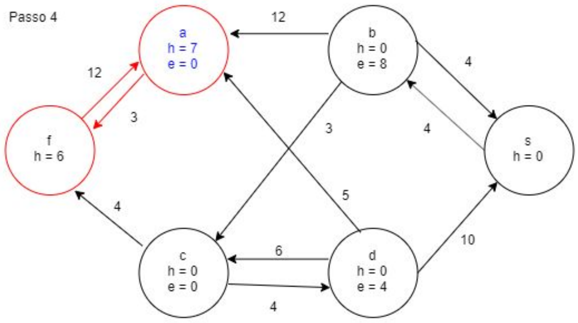
Passo 2



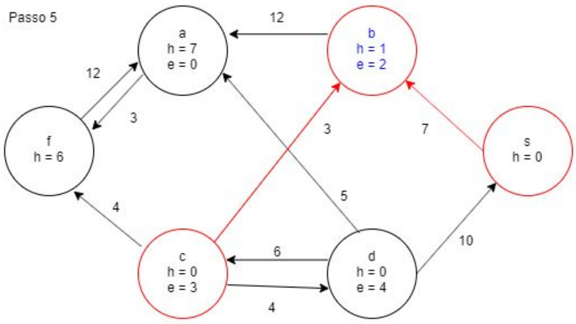
Passo 3



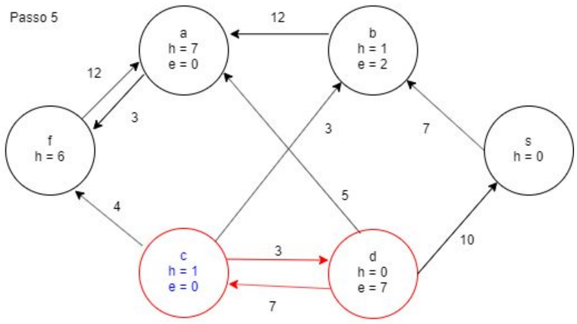
Passo 4



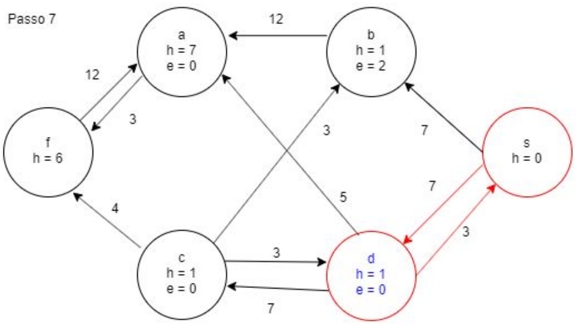
Passo 5



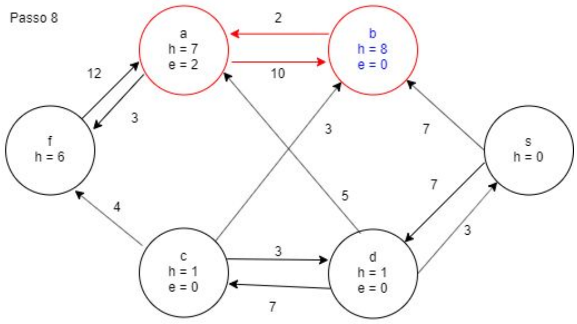
Passo 5



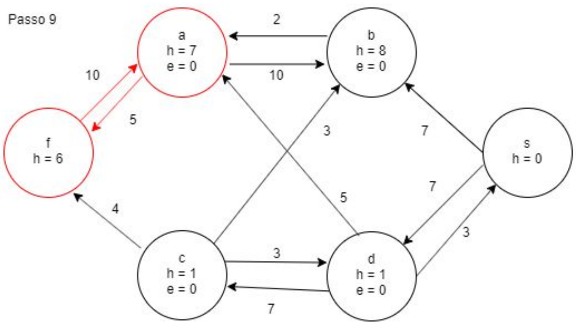
Passo 7



Passo 8



Passo 9



3. Teorema de Fluxo Máximo Corte Mínimo

3.1. Enunciado:

"O valor máximo de um fluxo s-t, onde s é o ponto destino e t o sorvedouro, é igual a capacidade mínima de um corte s-t."

max intensidade (fluxo) = min capacidade (corte)

3.2. Prova:

Considere que se um dado fluxo f não tem pseudo-caminho aumentador, ou seja, não possui um caminho em que vai do vértice inicial ao final do grafo e tem como propriedades nenhum arco direto estar cheio e nenhum arco reverso estar vazio, então f é um fluxo de intensidade máxima. Tal fato é provado a partir da seguinte análise: Tome a afirmação, por contradição, de que pseudo-caminho aumentador não termine em t. Agora considere o conjunto S de todos os vértices que são término de um pseudo-caminho aumentador. Nota-se que s está em S, porém t está fora de S pois, por hipótese, não existe pseudo-caminho aumentador que termine em t. Seja C o corte cuja margem superior é S; todos os arcos diretos de C estão completos e todos os arcos reversos estão vazios. Portanto, o saldo de f em S é igual à capacidade de C. Logo, a intensidade de f é igual à capacidade de C e portanto f é um fluxo máximo.

NOTAS SOBRE O CÓDIGO

1. Arquivo:

Todos os algoritmos foram implementados em um mesmo arquivo em python (main.py) que se encontra dentro da pasta (algoritmos) com todos os arquivos necessários para sua execução.

2. Instalação das dependências:
    2.1. Instalar o python3 e o pip:
    Sessão "Como Instalar Python no Ubuntu" do site: <https://www.hostinger.com.br/tutoriais/como-instalar-python-ubuntu/>
    2.2. Instalar biblioteca Numpy com o comando: sudo pip install numpy

3. Demais instruções:

- Abrir arquivo "grafos.txt" e adicionar o grafo desejado respeitando o modelo.
- Para executar utilize os comandos:
  - cd "caminho dos arquivos"
  - python main.py

4. Observações:

- Caso tenha algum problema durante a execução, o trabalho pode ser acessado e executado através da ferramenta Repl.it pelo link <https://repl.it/@RenanCardoso/trabFinalGrafos>.
- Se o grafo analisado for grande (por exemplo: n>8), o algoritmo menorRecSTP(w) apresenta uma demora maior para executar, visto que é um algoritmo com métodos recursivos. Caso haja problemas, basta comentar a linha 396 do arquivo "main.py" (utilize # para comentar).

REFERÊNCIAS

N, I., Mello, M. Bellman-Ford. *UnB*. Disponível em: <http://flaviomoura.mat.br/files/PAA/2019-1BellmanFord.pdf>. Acesso em: 23/11/2019.

Algoritmo de Floyd-Warshall. Disponível em: <https://pt.wikipedia.org/wiki/Algoritmo_de_Floyd-Warshall>. Acesso em: 23/11/2019.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2002). Algoritmos: teoria e prática. *Editora Campus*, *2*, 2.