

Relatório de Implementação dos Algoritmos

Heron Sanches Marino Honenheim
Nilton Vasques

Disciplina Teoria dos Grafos,
Professor Steffen Lewitzka,
Ciência da Computação,
Universidade Federal da Bahia

13 de Fevereiro de 2014

1 Árvore Geradora Mínima

Definição: Seja $G=(V,E)$ um grafo não direcionado e conexo, $G'=(V,E')$ é chamado de subgrafo gerador se possui os mesmos vértices de G . Portanto se tivermos em G' uma árvore, então o subgrafo é uma árvore geradora. Quando G é um grafo conexo, em que cada aresta possui um valor ou peso $p(e)$, o peso total da árvore geradora é

$$\sum_{e \in E'} p(e)$$

onde $p(e)$ é uma função que retorna o peso da aresta e . A árvore geradora mínima é a árvore G' que possui o menor peso total dentre todas as árvores possíveis do grafo G [2]. Podemos enunciar a função para encontrar a árvore geradora mínima como

$$\min \sum_{e \in E'} p(e)$$

. A partir dessa noção podemos visualizar que encontrar a árvore geradora mínima não é tão trivial assim. Se propormos uma solução pela força bruta, ou seja, encontrar todas as árvores geradoras e assim então verificar qual a que possui o menor peso total. No pior caso quando temos um grafo completo(em que todos os vértices se ligam uns aos outros) teríamos n^{n-2} árvores geradoras onde n é o número de nós, sendo assim teríamos uma solução em tempo

exponencial $O(n^n)$ e inviável [3]. Diante deste cenário alguns matemáticos elaboram soluções para o problema das Árvore Geradoras Mínimas, se utilizando de heurísticas gulosas para encontrar a solução ótima. No presente artigo abordaremos o Algoritmo de Kruskal e o de Prim, como estudo de caso.

1.1 Projeto

A implementação dos algoritmos referente a árvore geradora mínima, consistiu no desenvolvimento de um applet para java, que facilitasse a visualização das etapas realizadas pelos algoritmos de Kruskal e Prim, de maneira bastante interativa. Todo o material produzido na implementação esta disponível em [4].

1.2 Algoritmo de Kruskal

O algoritmo de Kruskal é um algoritmo guloso, que tem por objetivo encontrar uma árvore geradora mínima para um grafo conexo e valorado (com pesos nas arestas). Vale ressaltar que para árvores não conexas, o algoritmo encontra floresta geradora mínima, ou seja uma árvore geradora mínima para cada componente conexo do grafo. O algoritmo pode ser enunciado nos seguintes passos:

Data: Um grafo Conexo

Result: Uma árvore geradora mínima a partir de um grafo conexo

Criar uma floresta F , onde cada vértice do grafo é uma árvore separada;

Criar um conjunto S contendo todas as arestas do grafo;

while S *é não vazio* **do**

 Remova uma aresta e com peso mínimo de S ;

 Se e conecta duas diferentes árvores, então adicione e para floresta F ;

 Caso contrário, descarte e , ou seja se a escolha de e gera um circuito em F , descarte-a;

end

Algoritmo 1: Pseudo Código do algoritmo de Kruskal

1.2.1 Implementação

A implementação foi realizada em Java com uso da estrutura de dados de listas encadeadas para manipular os conjuntos disjuntos. O código fonte está disponível no Apêndice A. A seguir o pseudo código da implementação com

as manipulações representadas pelas operações Union-Find :

```

Data:  V, E
Result: A, W
1  W  $\leftarrow$  0; A  $\leftarrow$  vazio;
2  for  v  $\in$  V do
3    |  a[v]  $\leftarrow$  make-set(v);
4  end
5  L  $\leftarrow$  ordene(E, w);
6  k  $\leftarrow$  0;
7  while  k  $\neq$  | V | - 1 do
8    |  remove(L, (u, v ));
9    |  a[u]  $\leftarrow$  find-set(u);
10   |  a[v]  $\leftarrow$  find-set(v);
11   |  if  a[u]  $\neq$  a[v] then
12     |  |  aceita(u, v);
13     |  |  A  $\leftarrow$  A  $\cup$  {(u, v)};
14     |  |  W  $\leftarrow$  W + w(u, v);
15     |  |  k  $\leftarrow$  k + 1;
16   |  end
17   |  union(a[u], a[v]);
18 end
19 retorne(A, W);

```

Algoritmo 2: Pseudo Código do algoritmo de Kruskal com UnionFind

1.2.2 Análise de Complexidade

A estrutura de dados UnionFind mantém um conjunto de elementos particionados em vários subconjuntos não sobrepostos. O algoritmo que controla essa estrutura possui duas operações principais:

- Find: Determina de qual subconjunto um elemento pertence.
- Union: Faz a união de dois subconjuntos em um só subconjunto.

A ordenação na linha 5 do algoritmo 2, tem complexidade $O(|E| \log |E|)$ e domina a complexidade das demais operações. A repetição das linhas 7-17 será executado $O(|E|)$ no pior caso. Logo, a complexidade total das linhas 9-10 será $O(|E| f(|V|))$, onde $f(|V|)$ é complexidade da função **find-set**. As linhas de 12 a 15 serão executados $|V| - 1$ vezes no total, pois para um grafo contendo N vértices, precisamos de apenas N-1 arestas para interligar todos os nós e gerar uma árvore geradora mínima. Assim, a complexidade total de execução destas linhas será $O(|V| \cdot g(|V|))$ onde

$g(|V|)$ é a complexidade de realizar **union**. A complexidade do algoritmo de Kruskal será então:

$$O(|E| \log |E| + |E| \cdot f(|V|) + |V| \cdot g(|V|))$$

A estrutura de dados UnionFind foi implementada na sua forma simples, com o uso de uma lista encadeada. Sendo assim a complexidade da função find é $\omega(n)$, e union tem complexidade $O(n)$ [1]. A complexidade final da implementação foi:

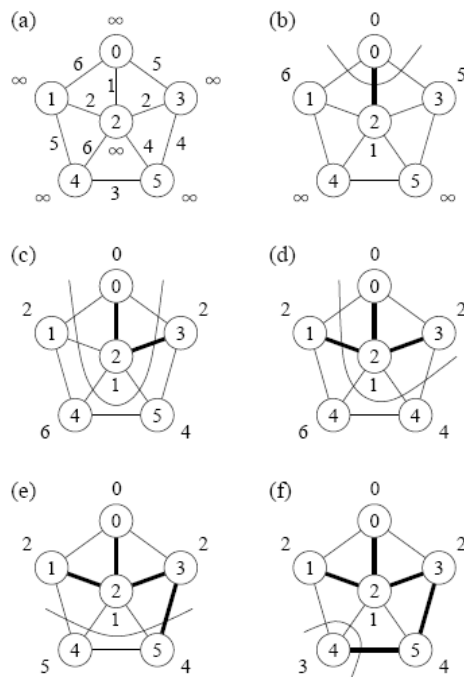
$$O(|E| \log |E| + |E| \cdot \Omega(|V|) + |V| \cdot O(|V|))$$

A complexidade pode ser reduzida utilizando de uma estrutura de dados mais refinada para implementar a manipulação dos conjuntos disjuntos, como por exemplo usar uma lista encadeada e *weighted-union heuristic* [1], consegue-se uma complexidade de $O(m + n \log n)$ para realizar as m operações de **make-set**, **find-set** e **union** [1].

1.3 Algoritmo de Prim

Assim como o algoritmo de Kruskal's o algoritmo de Prim utilizada também uma heurística gulosa para solucionar o problema da Àrvore Geradora Mínima. A heurística utilizada é procurar o caminho mais curto, dentre todos os possíveis, de maneira similar ao algoritmo de Dijkstra. O algoritmo de Prim's tem uma propriedade de que as arestas em A sempre forma uma árvore simples *Fig.1*.

Fig. 1: Ilustração do Algoritmo de Prim



O algoritmo genérico de Prim procura encontrar o caminho mais curto de um vértice para os vértices vizinhos, até que todos os vértices estejam ligados uns aos outros. O pseudo-código pode ser visualizado logo abaixo:

Data: Um grafo Conexo

Result: Uma árvore geradora mínima a partir de um grafo conexo

Escolha um vértice S para iniciar o subgrafo;

while *há vértices que não estão no subgrafo* **do**

 selecione uma aresta segura;

 insira a aresta segura e seu vértice no subgrafo;

end

Algoritmo 3: Pseudo Código do algoritmo de Prim

1.3.1 Implementação

Assim como a implementação anterior o algoritmo de Prim's foi implementado em Java e fez uso da Matriz de Adjacências Binária como estrutura de dados para armazenar os vértices adjacentes de um dado vértice u . O código fonte está disponível no Apêndice A. O pseudo código que foi utilizado para a implementação pode ser visualizado em *Algorithm 4*.

```
Data:  $V, E$   
Result:  $A, W$   
1  $dist[r] \leftarrow 0; Q \leftarrow V;$   
2 for  $v \in V - \{r\}$  do  
3    $dist[v] \leftarrow \infty;$   
4 end  
5  $pred[r] \leftarrow NULL;$   
6  $A \leftarrow \emptyset;$   
7  $W \leftarrow 0;$   
8 while  $Q$  não for vazio do  
9   remover de  $Q$  o vértice  $u$  com menor valor em  $dist$ ;  
10   $W \leftarrow W + dist[u];$   
11  if  $pred[u] \neq null$  then  
12     $A \leftarrow A \cup \{(pred[u], u)\};$   
13  end  
14  for  $v \in Adj[u]$  do  
15    if  $v \in Q$  and  $dist[v] > w[u, v]$  then  
16       $dist[v] \leftarrow w[u, v];$   
17       $pred[v] \leftarrow u;$   
18    end  
19  end  
20 end  
21 retorne( $A, W$ );
```

Algoritmo 4: Pseudo Código do algoritmo de Prim

1.3.2 Análise de Complexidade

A complexidade do algoritmo de Prim está diretamente ligada a maneira de como é implementada a estrutura de dados em Q . Uma simples implementação utilizando matrizes de adjacência vai requerer complexidade $O(|V|^2)$ e foi a estrutura de dados utilizada neste trabalho. Utilizando de uma heap binária a complexidade cai para $O(|E| \log |V|)$, ainda assim é possível decrescer ainda mais a complexidade utilizando como estrutura de dados uma Fibonacci Heap com complexidade $O(|E| + |V| \log |V|)$ [1].

2 Caminho Mínimo em Grafos Orientados

2.1 Algoritmo de Dijkstra

O algoritmo de Dijkstra, proposto pelo cientista E. W. Dijkstra, resolve o problema de encontrar o menor caminho entre dois vertices num grafo orientado ou não com arestas de peso não negativo. O algoritmo de Dijkstra é um metodo guloso para resolver o problema do caminho mais curto. A ideia por trás do método guloso é efetuar uma BFS ponderada sobre um dado grafo, a partir de um nó n . Dijkstra é comumente implementado com uma fila de prioridade como uma heap, de modo que em cada iteração, quando precisamos de obter o próximo nó a ser visitado, então este nó escolhido será o nó mais próximo ao nó n .

O algoritmo de Dijkstra mantém um conjunto S de vertices cujo o peso do menor caminho a partir de s já foi determinado. O algoritmo seleciona repetidamente o vértice $u \in V - S$ com o menor caminho estimado, acrescenta u em S , e relaxa todas as arestas que incidem em u . O seguinte pseudo-código usa uma fila de prioridade Q de vértices, introduzidos pelos seus valores d .

```
Data:  $G, w, s$ 
1 INITIALIZE-SINGLE-SOURCE( $G, s$ );
2  $S \leftarrow \emptyset$ ;
3  $Q \leftarrow V[G]$ ;
4 while  $Q \neq \emptyset$  do
5    $u \leftarrow EXTRACT\_MIN(Q)$ ;
6    $S \leftarrow S \cup \{u\}$ ;
7   foreach vertex  $v \in Adj[u]$  do
8     RELAX( $u, v, w$ );
9   end
10 end
```

Algoritmo 5: Pseudo Código do Algoritmo de Dijkstra

2.1.1 Análise de Complexidade

Seja n o número de nós e m o número de arestas.

Uma vez que implementamos nossa fila de prioridade como uma heap, então o tempo de complexidade para remover o elemento minimo da heap ou adicionar um novo elemento é $O(\log n)$. Agora precisamos considerar o fato de quando atualizamos as menores distância dos nós, ou a fase de relaxação das arestas obtemos $O(n + m)$.

Com isso, o tempo que o algoritmo de Dijkstra gasta em cada nó é $O(m \log n)$, enquanto que se precisamos visitar todos os nós, então a com-

plexidade de tempo do algoritmo de Dijkstra seria $O((n + m)\log n)$.

Até agora, consideramos apenas a computação de um único nó para todos os outros nós. Contudo a complexidade para computar a menor distância partindo de todos os nós para todos os outros nós é $O(n(n + m)\log n)$. Assim, se temos um grafo completo a complexidade seria $O(n^2\log n)$.

2.1.2 Implementação

O algoritmo foi implementado na linguagem Java. Fez uso da matriz de adjacências para verificar se um vértice é vizinho do outro, assim como utilizou filas e conjuntos para as estruturas internas do algoritmo. O código da classe Dijkstra está disponível no Apêndice, assim como também em [4].

2.2 Algoritmo de Floyd-Warshall

O algoritmo de Floyd-Warshall usa uma formulação de programação dinâmica para resolver o problema de caminhos mais curtos de todos os pares em grafo orientado $G = (V, E)$. O algoritmo é executado no tempo $O(V^3)$. O algoritmo de Floyd-Warshall se baseia na observação a seguir. Sejam $V = 1, 2, \dots, n$ os vértices de G , e considere um subconjunto $1, 2, \dots, k$ de vertices para algum k . Para qualquer par de vértices $i, j \in V$, considere todos os caminhos desde i até j cujos vértices intermediários são todos traçados a partir de $1, 2, \dots, k$, e seja p um caminho de peso mínimo dentre eles. O algoritmo de Floyd-Warshall explora um relacionamento entre o caminho p e caminhos mais curtos desde i até j com todos vértices intermediários no conjunto $1, 2, \dots, k-1$. O Relacionamento depende do fato de k ser ou não um vértice intermediário do caminho p .

Se k não é um vértice intermediário do caminho p , então todos os vértices intermediários do caminho p estão no conjunto $1, 2, \dots, k-1$. Desse modo, um caminho mais curto desde o vértice i até o j com todos os intermediários no conjunto $1, 2, \dots, k-1$ também é um caminho mais curto desde i até j com todos os vértices intermediários no conjunto $1, 2, \dots, k$.

Se k é um vértice intermediário do caminho p , então desmembramos p em ip_1kp_2j . p_1 é um caminho mais curto desde i até k com todos os vértices intermediários no conjunto $1, 2, \dots, k$. Como o vértice k não é um vértice intermediário do caminho p_1 , vemos que p_1 é um caminho mais curto desde i até k com todos os vértices intermediários no conjunto $1, 2, \dots, k-1$. De modo semelhante, p_2 é um caminho mais curto até o vértice j com todos os vértices intermediários no conjunto $1, 2, \dots, k-1$.

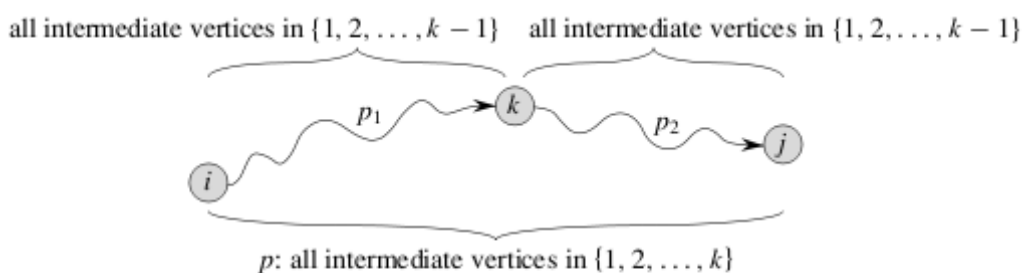


Figure 25.3 Path p is a shortest path from vertex i to vertex j , and k is the highest-numbered intermediate vertex of p . Path p_1 , the portion of path p from vertex i to vertex k , has all intermediate vertices in the set $\{1, 2, \dots, k-1\}$. The same holds for path p_2 from vertex k to vertex j .

A formulação recursiva seguindo a discussão acima é dada por:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1. \end{cases}$$

Com base na recorrência acima, o seguinte procedimento bottom-up pode ser usado para calcular o $d_{ij}(k)$, a fim de aumentar os valores de k . A sua entrada é uma matriz $n \times n$ W definido como na equação. O procedimento retorna a matriz $D(n)$ com os pesos dos menores caminhos.

```

FLOYD-WARSHALL( $W$ )
1   $n \leftarrow \text{rows}[W]$ 
2   $D^{(0)} \leftarrow W$ 
3  for  $k \leftarrow 1$  to  $n$ 
4      do for  $i \leftarrow 1$  to  $n$ 
5          do for  $j \leftarrow 1$  to  $n$ 
6               $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ 
7  return  $D^{(n)}$ 

```

Fig. 2: Pseudo Código do algoritmo de Floyd-Warshall

2.2.1 Implementação

O algoritmo foi implementado na linguagem python no arquivo floyd.py.

A função main cria um grafo predefinido e depois chama a função floydWarshall com o grafo sendo passado como parâmetro.

```

1  if __name__ == '__main__':
2
3      #Cria o grafo
4
5      grafo = {'A':{'A':0,'B':INF,'C':12,'D':6,'E':4},
6
7              'B':{'A':INF,'B':0,'C':INF,'D':1,'E':INF},
8
9              'C':{'A':INF,'B':INF,'C':0,'D':15,'E':3},
10
11             'D':{'A':INF,'B':INF,'C':15,'D':0,'E':4},
12
13             'E':{'A':INF,'B':7,'C':INF,'D':INF,'E':0}
14
15         }

```

A função `floydWarshall` primeiro armazena a distancia dos vertices em seguida executa o algoritmo de Floyd-Warshal para obter o menor caminho para todos os pares por fim exibe a tabela com os resultados.

```
1 def floydWarshall(grafo):
2
3     nos = grafo.keys()
4
5     distancia = {}
6
7     #armezena a distancia do no n para o no k
8     for n in nos:
9
10        distancia[n] = {}
11
12        for k in nos:
13
14            distancia[n][k] = grafo[n][k]
15        # Floyd-warshal - programacao dinamica
16        for k in nos:
17
18            for i in nos:
19
20                for j in nos:
21
22                    if distancia[i][k] + distancia[k][j] <
23                        distancia[i][j]:
24
25                        distancia[i][j] = distancia[i][k]+
26                            distancia[k][j]
27
28        # imprime a tabela com resultado
29        printSolution(distancia)
```

3 Busca em Profundidade ou Depth-First-Search (DFS) para Ordenação Topológica

3.1 Definição

Busca em profundidade (ou busca em profundidade-primeiro, também usada a sigla em inglês DFS) é um algoritmo usado para realizar uma busca ou travessia numa árvore, estrutura de árvore ou grafo. Intuitivamente, o algoritmo começa num nó raiz (selecioneando algum nó como sendo o raiz, no caso de um grafo) e explora tanto quanto possível cada um dos seus ramos, antes de retroceder(backtracking).

A estratégia seguida pela busca em profundidade é, procurar “mais fundo” no grafo sempre que possível. As arestas são exploradas a partir do vértice v mais recentemente descoberto que ainda tem arestas inexploradas saindo dele. Quando todas as arestas de v são exploradas, a busca “regressa” para explorar as arestas que deixam o vértice a partir do qual v foi descoberto. Esse processo continua até descobrirmos todos os vértices acessíveis a partir do vértice de origem inicial. Se restarem quaisquer vértices não descobertos, então um deles será selecionado como nova origem, e a busca se repetirá a partir daquela origem. Esse processo inteiro será repetido até que todos os vértices sejam descobertos.

Sempre que um vértice v é descoberto durante uma varredura da lista de adjacências de um vértice já descoberto u , a busca em profundidade registra esse evento definindo um campo predecessor de v , um campo $r[u]$, como u . O subgrafo predecessor produzido por uma busca em profundidade pode ser composto por várias árvores, ele forma uma floresta primeiro na profundidade composta por várias árvores primeiro na profundidade. O algoritmo genérico para busca em profundidade realiza passos que já foram descritos mais acima. O pseudo-código pode ser visualizado no Algoritmo 6:

Data: Inicio, Alvo

```

1 function BuscaProfundidade;
2 empilha(Pilha, Inicio);
3 while Pilha is not empty do
4   |  $varNodo \leftarrow desempilha(Pilha)$ ;
5   | Colore(Nodo, Cinza);
6   | if Nodo = Alvo then
7   |   | return Nodo;
8   | end
9   | for Filho in Expande(Nodo) do
10  |   | if Filho.cor = Branco then
11  |   |   | empilha(Pilha, Filho);
12  |   | end
13  | end
14  | Colore(Nodo, Preto);
15 end

```

Algoritmo 6: Pseudo-código do algoritmo de Busca em Profundidade

3.2 Implementação

A implementação deste algoritmo de busca em profundidade foi feito em Java, e fez uso de uma matriz de adjacências binária como estrutura de dados para armazenar os vértices adjacentes de um dado vértice. O código fonte está disponível no Apêndice. O pseudo-código que foi utilizado para a implementação da busca em profundidade pode ser visualizado no Algoritmo 7.

Data: Inicio, Alvo

```

1 Coloque o nó inicial no topo da pilha;
2 if pilha estiver vazia then
3   | retorne falha e pare;
4 end
5 if o elemento na pilha é o nó alvo g then
6   | retorne sucesso e pare;
7 end
8 else
9   | Remova e expanda o primeiro elemebto e coloque o filho no topo
   | da pilha;
10  | Volte ao passo 2;
11 end

```

Algoritmo 7: Pseudo-código para a implementação de Busca em Profundidade

3.3 Análise de Complexidade

O tempo de execução do nosso algoritmo de Busca em Profundidade é de tempo $O(V^2)$, para V igual à quantidade de vértices. Mas, a complexidade do problema pode ser de tempo $O(V+E)$, sendo E a quantidade de arestas. No nosso algoritmo, primeiro entra-se com um inteiro não negativo da quantidade de vértices do grafo, após isso diz-se qual o vértice-origem do grafo e o programa gera o vetor ordenado por Busca em Profundidade.

3.4 Ordenação Topológica (usando Busca em Profundidade)

Uma ordenação topológica de um grafo acíclico orientado $G = (V, E)$ é uma ordenação linear de todos os seus vértices, tal que se G contém uma aresta (u, v) , então u aparece antes de v na ordenação. A ordenação topológica de um grafo pode ser vista como uma ordenação de seus vértices ao longo de uma linha horizontal de tal forma que todas as arestas orientadas sigam da esquerda para a direita. Grafos acíclicos orientados (gao) são usados em muitas aplicações para indicar precedências entre eventos.

TOPOLOGICAL-SORT(G);

chamar DepthFirstSearch (DFS) para cada vértice v ;

à medida que cada vértice é terminado, inserir o vértice à frente de uma lista ligada;

return a lista ligada de vértices.;

Algoritmo 8: Pseudo Código para ordenar topologicamente um gao

Executamos uma ordem topológica no tempo $O(V+E)$, pois a busca em profundidade demora o tempo $O(V+E)$ e leva tempo $O(1)$ para inserir cada um dos $|V|$ vértices à frente da lista ligada.

Para o exemplo a seguir, a matriz adjacência seria a que é mostrada na Fig 4.

Uma ordenação da busca profundidade para o exemplo seria $7 \rightarrow 11 \rightarrow 2 \rightarrow 9 \rightarrow 10 \rightarrow 5 \rightarrow 8 \rightarrow 3 \rightarrow 10$.

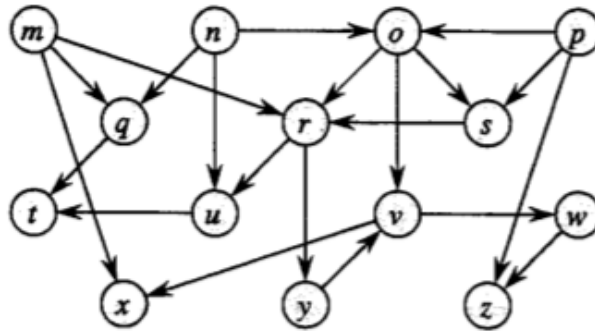


Fig. 3: Um grafo para ordenação topológica

	2	3	5	7	8	9	10	11
2	0	0	0	0	0	0	0	0
3	0	0	0	0	1	0	0	0
5	0	0	0	0	0	0	0	1
7	0	0	0	0	1	0	0	1
8	0	0	0	0	0	1	0	0
9	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0
11	1	0	0	0	0	1	0	0

Fig. 4: Matriz Adjacência para o grafo ao lado

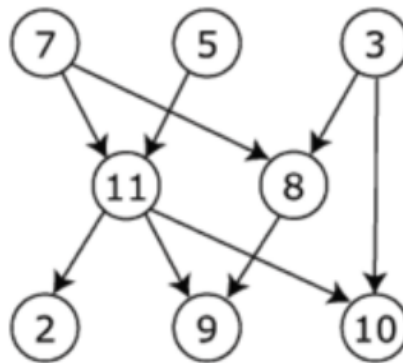


Fig. 5: Exemplo de grafo direcionado a ser ordenado por profundidade

4 Fechamento Transitivo

4.1 Definição

Suponha que temos um grafo direcionado $G = (V, E)$. É útil saber que, dado um par de vértices u e w , onde há um caminho de u para w no grafo. Uma

boa maneira de guardar esta informação é construir um outro grafo, chamá-lo de $G^* = (V, E^*)$, tal que existe uma aresta (u, w) em G^* , se e somente se, existe um caminho de u para w em G . Esse grafo é chamado de fechamento transitivo.

O nome “fechamento transitivo” significa que: Ter uma propriedade transitiva significa que se a se relaciona com b de alguma forma especial, e b se relaciona com c , então a se relaciona com c . Somos familiares com várias formas de transitividade. No caso de grafos, dizemos que um grafo é transitivo se, para todo trio de vértices a , b e c , se (a,b) é uma aresta, e (b,c) é uma aresta, então (a,c) também é uma aresta. Alguns grafos são transitivos, outros não. Algebricamente: “ R é transitiva sse $R^n \subseteq R$ para todo $n \geq 1$ ”. Um conjunto A^* é um fechamento de um conjunto A com alguma propriedade especial (como transitividade) é resultado da soma de A com somente os elementos que causam A satisfazer esta propriedade especial, e não outros elementos. O fechamento transitivo de um grafo é resultado da adição das menores arestas possíveis ao grafo tal que é transitivo. (Podemos facilmente adicionar uma porção de arestas a um grafo para fazê-lo transitivo, mas a parte do fechamento transitivo significa que queremos preservar as relações de caminhos existentes anteriormente, i.e., não adicionaremos arestas que não representam caminhos no grafo. Representaremos grafos usando uma matriz de adjacência de valores booleanos e usaremos esta matriz de adjacência para construir a matriz de fechamento transitivo.

4.2 Algoritmo de Warshall

O algoritmo de Warshall é um algoritmo de força bruta, tem por objetivo encontrar para cada elemento do grafo, as arestas que chegam e saem dele. É um eficiente método de computar a transitividade de uma relação. O algoritmo de Warshall tem como entrada uma matriz M_r representando a relação R e tem como saída a matriz M_r^* da relação R^* , o fechamento transitivo de

R. Abaixo está um pseudo-código para Warshall.

```

Transitive-Closure (G);
n = | V |;
t(0) = the adjacency matrix for G;
// there is always an empty path from a vertex to itself, // make sure
the adjacency matrix reflects this
for i in 1..n do
    | t(0)[i,i] = True;
end
// step through the t(k)'s;
for k in 1..n do
    | for i in 1..n do
        | for j in 1..n do
            | t(k)[i, j] = t(k-1)[i, j] OR (t(k-1)[i, k] AND t(k-1)[k, j]);
            end
        end
    end
end
return t(n);

```

Algoritmo 9: Pseudo-código do algoritmo de Warshall

4.3 Implementação

A implementação foi feita em Python. O código fonte está disponível no Apêndice. A seguir o pseudo código da implementação com a função principal do algoritmo de Warshall.

```

Data: MR: n x n 0-1 matrix
W := MR(W = [wi,j]) for k=1 to n do
    | for i=1 to n do
        | for j=1 to n do
            | wi,j = wi,j ∨ (wi,k ∧ wk,j);
            end
        end
    end
end
return W;

```

Algoritmo 10: Pseudo-código da função principal do algoritmo de Warshall

4.4 Análise de Complexidade

Ao final do algoritmo, a simples análise dele nos diz que a complexidade dos 3 loops levam um tempo $O(n^3)$. Com relação ao armazenamento de da-

dos, notamos que no algoritmo só precisamos de duas matrizes computadas, então podemos reusar o armazenamento para outras matrizes, nos dando uma complexidade de armazenamento de $O(n^2)$.

Appendices

Algoritmos Implementados no Projeto

```
1 package br.ufba.graph.algorithm.minimumspanningtree;
2
3 import br.ufba.datastructures.UnionFind;
4 import br.ufba.datastructures.UnionFind.UnionElement;
5 import br.ufba.graph.Aresta.Status;
6 import br.ufba.graph.Aresta;
7 import br.ufba.graph.Graph;
8 import br.ufba.graph.Vertice;
9 import br.ufba.graph.algorithm.GraphAlgorithm;
10
11
12 /**
13  * @author niltonvasques
14  * http://en.wikipedia.org/wiki/Kruskal's\_algorithm
15  */
16 public class Kruskal implements GraphAlgorithm{
17
18     private Graph mGraph;
19
20     private int edges[];
21
22     private int edgeIndex = 0;
23
24     public Kruskal(Graph grafo) {
25         mGraph = grafo;
26     }
27
28     @Override
29     public void init(){
30         edgeIndex = 0;
31         // crie uma floresta F (um conjunto de árvores), onde cada vértice
           no grafo é uma árvore separada
32
33         for (int i = 0; i < mGraph.getVerticesCount() ;
           i++) {
34             Vertice vertice = mGraph.getVertices()[i];
35             UnionFind.makeSet(vertice);
36         }
37     }
```

```

38 // create a set S containing all the edges in the graph ordered by
    weight
39     edges = new int[mGraph.getArestasCount()];
40     for( int i = 0; i < edges.length; edges[i] = i
        ++);
41     qsort(0, edges.length -1);
42 }
43
44
45
46 @Override
47 public boolean performStep() {
48 /*     enquanto S for nao-vazio, faca:
49 *         remova uma aresta com peso minimo de S
50 *         se essa aresta conecta duas arvores
diferentes, adicione-a a floresta, combinando duas
arvores numa unica arvore parcial
51 *         do contrario, descarte a aresta
52 */
53     if( edgeIndex >= edges.length )
54         return true;
55
56     Aresta aresta = mGraph.getArestas()[ edges[
        edgeIndex]];
57     if( aresta.status == Status.WAITING ){
58         aresta.status = Status.PROCESSING;
59         return false;
60     }
61     edgeIndex++;
62
63     UnionElement u = aresta.u;
64     UnionElement v = aresta.v;
65     if( !UnionFind.find(u).equals(UnionFind.find(v))
        ){
66         aresta.status = Status.TAKED;
67         UnionFind.union(u, v);
68     }else{
69         aresta.status = Status.DISCARDED;
70     }
71     return false;
72 }
73
74 private int partition(int left, int right) {

```

```

75     int pivot = mGraph.getArestas()[edges[(left +
76         right) / 2]].weight;
77     while (left <= right) {
78         while (mGraph.getArestas()[edges[left]].
79             weight < pivot)
80             left++;
81         while (mGraph.getArestas()[edges[right]].
82             weight > pivot)
83             right--;
84
85         if (left <= right) {
86             int i = left++;
87             int j = right--;
88             int k = edges[i];
89             edges[i] = edges[j];
90             edges[j] = k;
91         }
92     }
93     return left;
94 }
95
96 protected void qsort(int left, int right) {
97     if (left >= right)
98         return;
99
100     int i = partition(left, right);
101     qsort(left, i - 1);
102     qsort(i, right);
103 }
104 }

```

Implementação do Algoritmo de Kruskal em Java

```

1 package br.ufba.graph.algorithm.minimumspanningtree;
2
3 import br.ufba.datastructures.AdjacencyMatrix;
4 import br.ufba.graph.Aresta;
5 import br.ufba.graph.Aresta.Status;
6 import br.ufba.graph.Graph;
7 import br.ufba.graph.algorithm.GraphAlgorithm;
8
9 /**
10  * @author niltonvasques
11  * http://en.wikipedia.org/wiki/Prim%27s\_algorithm

```

```

12  */
13  public class Prim implements GraphAlgorithm{
14      private Graph mGraph;
15      private AdjacencyMatrix verticesMatrix;
16      private AdjacencyMatrix matrix;
17      private boolean processing = false;
18
19      public Prim( Graph graph) {
20          mGraph = graph;
21      }
22      @Override
23      public void init() {
24          processing = false;
25          verticesMatrix = new AdjacencyMatrix(mGraph.
26              getVerticesCount());
27          verticesMatrix.makeAdjacency(0, 0);
28          matrix = mGraph.createAdjacencyMatrix();
29      }
30
31      @Override
32      public boolean performStep() {
33          Aresta bestChoice = null;
34          int bestVertex = -1;
35          int vertices[] = verticesMatrix.getAdjacencys(0)
36              ;
37          for( int v = 0; v < vertices.length; v++){
38              if( vertices[v] == 1){
39                  int adjacencys[] = matrix.getAdjacencys(
40                      v);
41                  for( int i = 0; i < adjacencys.length; i
42                      ++){
43                      if(adjacencys[i] == 1){
44                          Aresta aresta = mGraph.getAresta
45                              (i, v);
46                          if( aresta != null){
47                              if( !processing ){
48                                  aresta.status = Status.
49                                      PROCESSING;
50                              }else if ( bestChoice !=
51                                  null ){
52                                  if( bestChoice.weight >
53                                      aresta.weight ){

```

```

46         bestChoice = aresta
47         ;
48         bestVertice = i;
49     }
50     }else{
51         bestChoice = aresta;
52         bestVertice = i;
53     }
54 }
55 }
56 }
57 }
58 if(!processing){
59     processing = true;
60     return false;
61 }
62 boolean finish = true;
63 if( bestVertice != -1 ){
64     bestChoice.status = Status.TAKED;
65     for(int i = 0; i < vertices.length; i++){
66         if( vertices[i] == 1){
67             matrix.removeAdjacency(i,
68                 bestVertice);
69         }else{
70             finish = false;
71         }
72     }
73     processing = false;
74     verticesMatrix.makeAdjacency(0, bestVertice)
75     ;
76 }
77 return finish;
78 }
79 }

```

Implementação do Algoritmo de Prims em Java

```

1 package br.ufba.graph.algorithm.mininumpath;
2
3 import java.util.ArrayList;
4 import java.util.List;

```

```

5
6 import br.ufba.datastructures.AdjacencyMatrix;
7 import br.ufba.graph.Aresta;
8 import br.ufba.graph.Aresta.Status;
9 import br.ufba.graph.algorithm.GraphAlgorithm;
10 import br.ufba.graph.Graph;
11 import br.ufba.graph.Vertice;
12
13
14 /**
15  * @author niltonvasques
16  * Link: http://en.wikipedia.org/wiki/Dijkstra%27s\_algorithm
17  */
18 public class Dijkstra implements GraphAlgorithm{
19
20     private static final int INFINITY = Integer.
        MAX_VALUE;
21     private static final int UNDEFINED = -1;
22
23     private Graph graph;
24     private AdjacencyMatrix matrix;
25     private Vertice source;
26     private Vertice target;
27
28     public Dijkstra(Graph graph) {
29         this.graph = graph;
30     }
31
32     @Override
33     public void init() {
34         matrix = graph.createAdjacencyMatrix();
35     }
36
37     @Override
38     public boolean performStep() {
39         execute(source);
40         return false;
41     }
42
43     public void setSource(int source){
44         this.source = graph.getVertices()[source];
45     }

```



```

46
47     public void setTarget(int target){
48         try{
49             this.target = graph.getVertices()[target];
50         }catch(Exception e){
51             this.target = null;
52         }
53     }
54
55     private void execute(Vertex source){
56         int[] dist = new int[graph.getVerticesCount()];
57         int[] previous = new int[graph.getVerticesCount()];
58
59         for(int i = 0; i < graph.getVerticesCount(); i
60             ++){
61             dist[i] = INFINITY;
62
63             previous[i] = UNDEFINED;
64         }
65
66         dist[source.index] = 0;
67
68         List<Vertex> q = new ArrayList<Vertex>();
69         for (int i = 0; i < graph.getVerticesCount(); i
70             ++){
71             q.add(graph.getVertices()[i]);
72         }
73
74         while(!q.isEmpty()){
75             Vertex u = menorDistancia(dist, q);
76             q.remove(u);
77
78             if(dist[u.index] == INFINITY)
79                 break;
80
81             int[] adj = matrix.getAdjacencys(u.index);
82             for(int v = 0; v < adj.length; v++){
83                 if(adj[v] == 1 && q.contains(graph.
getVertices()[v])){

```

```

84         int alt = dist[u.index] + graph.
            getAresta(u.index, v).weight;
85         if (alt < dist[v]){
86             dist[v] = alt;
87             previous[v] = u.index;
88             q.remove(graph.getVertices()[v])
                ;
89             q.add(graph.getVertices()[v]);
90         }
91     }
92 }
93 }
94 }
95
96 if(target == null){
97     for(int i = 0; i < dist.length; i++){
98         System.out.println(source.nome+" ->
                "+(i+1)+" = "+(dist[i] ==
                INFINITY ? "INFINITO": dist[i]));
99     }
100 }else{
101     System.out.println(source.nome+" -> "+target
        .nome+" = "+(dist[target.index] ==
        INFINITY ? "INFINITO": dist[target.index
        ]));
102     clearPaths();
103     paintPath(target.index, previous);
104 }
105 }
106
107 private Vertice menorDistancia(int[] dist, List<
    Vertice> q) {
108     Vertice menor = q.get(0);
109
110     int menorDist = INFINITY;
111     for(Vertice v : q){
112         if(dist[v.index] < menorDist){
113             menor = v;
114             menorDist = dist[v.index];
115         }
116     }
117     return menor;
118 }

```

```

119
120     private void paintPath(int origem, int[] previous){
121         if(previous[origem] != UNDEFINED){
122             paintPath(previous[origem], previous);
123             graph.getAresta(origem, previous[origem] ).
                status = Status.TAKED;
124         }
125     }
126
127     private void clearPaths(){
128         for (Aresta aresta : graph.getArestas()) {
129             if(aresta != null)
130                 aresta.status = Status.WAITING;
131         }
132     }
133 }

```

Implementação do Algoritmo de Dijkstra em Java

```

1  INF = 999999999
2
3  def printSolution(distGraph):
4
5      string = "inf"
6
7      nodes =distGraph.keys()
8
9      for n in nodes:
10
11          print "\t%6s"%(n),
12
13      print " "
14
15      for i in nodes:
16
17          print"%s"%(i),
18
19          for j in nodes:
20
21              if distGraph[i][j] == INF:
22
23                  print "%10s"%(string),
24
25              else:

```

```

26         print "%10s"%(distGraph[i][j]),
27
28     print " "
29
30
31 def floydWarshall(grafo):
32
33     nos = grafo.keys()
34
35     distancia = {}
36
37     #armezena a distancia do no n para o no k
38     for n in nos:
39
40         distancia[n] = {}
41
42         for k in nos:
43
44             distancia[n][k] = grafo[n][k]
45     # Floyd-warshal - programacao dinamica
46     for k in nos:
47
48         for i in nos:
49
50             for j in nos:
51
52                 if distancia[i][k] + distancia[k][j] <
                    distancia[i][j]:
53
54                     distancia[i][j] = distancia[i][k]+
                        distancia[k][j]
55     # imprime a tabela com resultado
56     printSolution(distancia)
57
58 if __name__ == '__main__':
59
60     #Cria o grafo
61
62     grafo = {'A':{'A':0,'B':INF,'C':12,'D':6,'E':4},
63
64             'B':{'A':INF,'B':0,'C':INF,'D':1,'E':INF},
65
66             'C':{'A':INF,'B':INF,'C':0,'D':15,'E':3},

```

```

67
68         'D': {'A': INF, 'B': INF, 'C': 15, 'D': 0, 'E': 4},
69
70         'E': {'A': INF, 'B': 7, 'C': INF, 'D': INF, 'E': 0}
71
72     }
73
74     floydWarshall(grafo)

```

Implementação do Algoritmo de Floyd em Python

```

1  #!/usr/bin/python
2  # vim: set fileencoding: utf-8 :
3
4  #Encontra para cada elemento do grafo, as arestas que chegam e saem
5  #Para cada par de aresta de chegada e saida, coloca 1 na matriz de
   saída
6
7  def warshall(p, n):
8      for k in range(n):
9          for i in range(n):
10             for j in range(n):
11                 p[i][j]=max(p[i][j],(p[i][k]&p[k][j]))
12
13     return
14
15 def max(a,b):
16     if(a>b):
17         return(a)
18     else:
19         return(b)
20
21 print("\n Quantidade de vertices e arestas,
   respectivamente:")
22 n= int(input(""))
23 e= int(input(""))
24
25 p = [[0 for i in range(n)] for i in range(n)]
26 for i in range(e):
27     print("\n Vertices que a aresta incide %d:" % (i+1))
28     u=int(input(""))
29     v=int(input(""))
30     p[(u-1)][(v-1)]=1
31

```

```

32 print("\n Matrix adjacencia:")
33 for i in range(n):
34     print(p[i])
35
36 warshall(p,n)
37
38 for i in range(n):
39     for j in range(n):
40         if(i==j):
41             p[i][j]=1
42
43 print("\n Fechamento transitivo: \n")
44 for i in range(n):
45     print(p[i])

```

Implementação do Algoritmo de Warshall em Python

```

1 package br.ufba.graph.algorithm.search;
2
3 import java.util.Stack;
4
5 import br.ufba.datastructures.Adjacencymatrix;
6 import br.ufba.graph.Graph;
7 import br.ufba.graph.Vertice;
8 import br.ufba.graph.Aresta.Status;
9 import br.ufba.graph.algorithm.GraphAlgorithm;
10
11 /**
12  * @author niltonvasques
13  * @author marinofull
14  * link: http://www.ime.usp.br/~pf/analise\_de\_algoritmos/aulas/dfs.html
15  */
16 public class DFS implements GraphAlgorithm{
17
18     private Graph graph;
19     private Adjacencymatrix matrix;
20     private Stack<Vertice> stack = new Stack<Vertice>();
21
22     private int ordemBusca[];
23     private int index = 0;
24
25     public DFS(Graph graph) {
26         this.graph = graph;

```

```

27     }
28
29     @Override
30     public void init() {
31         this.matrix = this.graph.createAdjacencyMatrix()
32             ;
33
34         for(int i = 0; i < graph.getVerticesCount(); i
35             ++){
36             graph.getVertices()[i].status = Vertice.
37                 Status.BRANCO;
38         }
39         stack.push(graph.getVertices()[0]);
40         graph.getVertices()[0].status = Vertice.Status.
41             CINZA;
42
43         index = 0;
44         ordemBusca = new int[graph.getVerticesCount()];
45         ordemBusca[index++] = next;
46     }
47
48     int next = 0;
49     @Override
50     public boolean performStep() {
51
52         if(stack.isEmpty()) return true;
53
54         boolean step = false;
55
56         while(!step){
57             Vertice u = stack.peek();
58
59             if(next >= graph.getVerticesCount()){
60                 next = 0;
61                 stack.pop();
62                 u.status = Vertice.Status.PRETO;
63                 return false;
64             }
65
66             Vertice vertexV = graph.getVertices()[next];

```

```

65         if(matrix.checkAdjacency(u.index, next) &&
66            vertexV != null
67            && vertexV.status == Vertice.Status.
68               BRANCO){
69
70             graph.getAresta(u.index, next).status =
71               Status.TAKED;
72             vertexV.status = Vertice.Status.CINZA;
73             stack.push(vertexV);
74             ordemBusca[index++] = next;
75             next = 0;
76             step = true;
77         }else{
78             next++;
79         }
80     }
81
82     @Override
83     public String toString() {
84         String result = "";
85         for(int i = 0; i< ordemBusca.length; i++){
86             result += " -> " + (ordemBusca[i]+1);
87         }
88         return result;
89     }
90 }
91 }

```

Implementação da busca profundidade em Java

Código Fonte das Estruturas de Dados Implementadas no Projeto

```

1 package br.ufba.datastructures;
2
3
4 /**
5  * @author niltonvasques
6  * UnionFind data structure
7  * http://en.wikipedia.org/wiki/Disjoint-
8  \*   set\_data\_structure
9  */
10 public class UnionFind {

```



```

10
11     public interface UnionElement{
12
13         public UnionElement getRoot();
14         public UnionElement getParent();
15         public void setRoot(UnionElement x);
16         public void setParent(UnionElement x);
17
18     }
19
20     public static void makeSet( UnionElement x ){
21         x.setParent(x);
22     }
23
24     public static UnionElement find( UnionElement x){
25         if ( x.getParent() == x )
26             return x;
27         else
28             return find(x.getParent());
29     }
30
31     public static void union( UnionElement x,
32                             UnionElement y){
33         x.setRoot( find(x) );
34         y.setRoot( find(y) );
35         x.getRoot().setParent( y.getRoot() );
36     }
37 }

```

Interface Para Operações Union-Find com Lista Encadeada

```

1 package br.ufba.datastructures;
2
3 /**
4  * @author niltonvasques
5  * http://pt.wikipedia.org/wiki/Matriz\_de\_adjac%C3%A2ncia
6  */
7 public class AdjacencyMatrix {
8
9     private static final int MEM_BLOCK_SIZE = 32;
10     int matrix[];
11     int stride;

```

```

12     public AdjacencyMatrix(int n) {
13         stride = n;
14         int size = (int)(stride * stride);
15         int lenght = 1 + (size/32);
16         matrix = new int[ lenght ];
17     }
18
19     public void makeAdjacency(int element, int adjacency
20     ){
21         makeAdjacencyInternal(element, adjacency);
22         makeAdjacencyInternal(adjacency, element);
23     }
24
25     public void removeAdjacency(int element, int
26     adjacency ){
27         removeAdjacencyInternal(element, adjacency);
28         removeAdjacencyInternal(adjacency, element);
29     }
30
31     private void makeAdjacencyInternal(int element, int
32     adjacency) {
33         int bitIndex          = (element*stride+
34         adjacency);
35         int index              = (int) (bitIndex/MEM_BLOCK_SIZE
36         );
37         int shift              = bitIndex % MEM_BLOCK_SIZE;
38         matrix[index]          |= (0x01 << shift);
39     }
40
41     private void removeAdjacencyInternal(int element,
42     int adjacency) {
43         int bitIndex          = (element*stride+
44         adjacency);
45         int index              = (int) (bitIndex/MEM_BLOCK_SIZE
46         );
47         int shift              = bitIndex % MEM_BLOCK_SIZE;
48         matrix[index]          &= ~(0x01 << shift);
49     }
50
51     public int[] getAdjacencys(int element){
52         int adjacencys[] = new int[stride];
53
54         int bitIndex          = (element*stride);

```

```

47         int bitRemainder      = (bitIndex % MEM_BLOCK_SIZE
48             );
49
50         for( int i = 0; i < stride; i++){
51             int x = ((bitIndex+i)/MEM_BLOCK_SIZE);
52             int y = bitRemainder + i;
53
54             int ret = (matrix[x] >> y) & 0x01;
55             adjacencys[i] = ret;
56         }
57
58         return adjacencys;
59     }
60
61     public boolean checkAdjacency(int u, int v){
62
63         int bitIndex          = (u*stride) + v;
64         int index              = (int) (bitIndex/
65             MEM_BLOCK_SIZE);
66
67         int y = (bitIndex % MEM_BLOCK_SIZE);
68
69         return ((matrix[index] >> y) & 0x01) == 0x01;
70     }
71
72     @Override
73     public String toString() {
74         String result = "";
75         for(int i = 0; i < matrix.length; i++){
76             result += "["+i+": "+Integer.toBinaryString
77                 (matrix[i]);
78         }
79     }
80 }

```

Implementação da Matriz de Adjacências Binária

Referências

- [1] Thomas H. Cormem, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, volume ISBN 0-262-03293-7, chapter

- 21: Data structures for Disjoint Sets, pages 498–524. MIT Press, second edition, 2001.
- [2] Fernando Nogueira. Problema da Árvore Gerador Mínima. *UFJF*.
- [3] F. Prado, T. Almeida, and V. N. Souza. Introdução ao Estudo sobre Árvore Geradora Mínima em Grafos com Parâmetros Fuzzy. *UNICAMP - Faculdade de Engenharia Elétrica*.
- [4] Nilton Vasques. Árvore geradora mínima - teoria dos grafos. <https://github.com/niltonvasques/teoria-dos-grafos>, 2013.