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

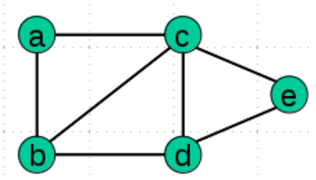
1 Representação do Grafo

1.1 Matriz de Adjacências

Um grafo simples G = (V,E), orientado ou não, pode ser representado internamente por uma estrutura de dados denominada matriz de adjacências. A matriz de adjacências é uma matriz quadrada A de ordem $\mid V \mid$, cujas as linhas e colunas são indexadas pelos vértices em V, da seguinte forma abaixo:

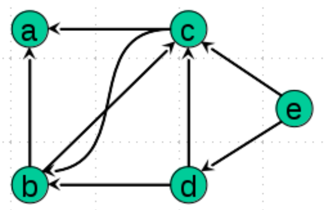
$$A(i,j) = \begin{cases} 1 \; se(i,j) \in E \\ 0 \; caso \; contrário \end{cases}$$

Fig. 1: Exemplo da representação de um grafo simples não orientado em sua correspondente matriz de adjacências.[6]



	a	b	c	d	e
a	0	1	1	0	0
b	1	0	1	1	0
c	1	1	0	1	1
d	0	1	1	0	1
e	0	0	1	1	0

Fig. 2: Exemplo de um grafo orientado simples e a matriz de adjacências correspondente.[6]



	a	b	c	d	e
a	0	0	0	0	0
b	1	0	1	0	0
c	1	1	0	0	0
d	0	1	1	0	0
e	0	0	1	1	0

2 Árvore Geradora Mínima

Definição: Seja G=(V,E) um grafo não direcionado e conexo, G'=(V,E') é chamado de subgrafo gerador se possuí 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*. Á árvore geradora mínima é a árvore G' que possui o menor peso total dentre todas as árvores possíveis do grafo G[4]. 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 [5]. Diante deste cenário alguns matemáticos elaboram soluções para o problema das Árvores 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.

2.1 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 [7].

2.2 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 todos as arestas do grafo;
while S *é não vazio* **do**
    Remova um aresta e com peso mínimo de S;
    Se e conecta duas diferentes árvores, então adicione e para floresta F;
    Caso contrário, discarte e, ou seja se a escolha de e gera um circuito em F, discarte-a;
end

Algoritmo 1: Pseudo Código do algoritmo de Kruskal

2.2.1 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 ← 0; A ← vazio;
2  for v ∈ V do
3  |  a[v] ← make-set(v);
4  end
5  L ← ordene(E, w);
6  k ← 0;
7  while k ≠| V | −1 do
8  |  remove(L, (u, v ));
9  |  a[u] ← find-set(u);
10 |  a[v] ← find-set(v);
11 |  if a[u] ≠ a[v] then
12 |  |  aceita(u, v);
13 |  |  A ← A ∪ {(u, v)};
14 |  |  W ← W + w(u, v);
15 |  |  k ← 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

2.2.2 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 $O(\mid E \mid log \mid E \mid)$ e domina a complexidade das demais operações. A repetição das linhas 7-17 será executado $O(\mid E \mid)$ no pior caso. Logo, a complexidade total das linhas 9-10 será $O(\mid E \mid f(\mid V \mid))$, onde $f(\mid V \mid))$ é complexidade da função **find-set**. As linhas de 12 a 15 serão executados $\mid V \mid -1$ 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 $O(\mid V \mid .g(\mid V \mid)$ onde

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

$$O(\mid E \mid log \mid E \mid + \mid E \mid .f(\mid V \mid) + \mid V \mid .g(\mid V \mid))$$

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 $O(n)$ [3]. A complexidade final da implementação foi:

$$O(\mid E \mid log \mid E \mid + \mid E \mid .\Omega(\mid V \mid) + \mid V \mid .O(\mid V \mid))$$

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*[3], consegue-se uma complexidade de $O(m+nlogn)$ para realizar as m operações de **make-set**, **find-set** e **union** [3].

3 Caminho Mínimo em Grafos Orientados

3.1 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 algoritimo 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 mantem 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
INITIALIZE-SINGLE-SOURCE(G,s);
S ← ∅;
Q ← V[G];
while Q = ∅ do
    u ← EXTRACTING − MIN(Q);
    S ← S ∪ {u};
    foreach vertex v ∈ Adj[u] do
        RELAX(u,v,w);
    end
end
```

Algoritmo 3: Pseudo Código do Algoritmo de Dijkstra

3.1.1 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 $O(logn)$. 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 $O(mlogn)$, enquanto que se precisamos visitar todos os nós, então a complexidade de tempo do algoritmo de Dijkstra seria $O((n + m)logn)$.

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 $O(n(n+m)logn)$. Assim, se temos um grafo completo a complexidade seria $O(n^2logn)$.

3.1.2 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 [7].

3.2 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 $O(V^3)$. É importante ressaltar que este algoritmo também pode ser usado para determinar se um grafo tem fecho transitivo, ou seja, se há caminho entre todos os vértices do grafo[2].

O algoritmo de Floyd-Warshall se baseia na observação a seguir. Sejan V = 1,2,...,n os vértices de G, e considere um subconjunto 1,2,...,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,...,k, e seja p um caminho de peso minimo dentre eles. O algoritmo de Floyd-Warshal explora um relacionamento entre o caminho p e caminhos mais curtos desde i até j com todos vértices intermediários no conjunto1,2,...,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,...,k-1. Desse modo, um caminho mais curto desde o vértice i até o j com todos os intermediários no conjunto 1,2,...,k-1 também é um caminho mais curto desde i até j com todos os vértices intermediários no conjunto 1,2,...,k.

Se k é um vertice intermediário do caminho p, então desmembramos p em ip_1kp_2j. P1 é um caminho mais curto desde i até k com todos os vértices intermediários no conjunto 1,2,...,k. Como o vértice k não é um vertice intermediário do caminho p1, vemos que p1 é um caminho mais curto desde i até k com todos os vértices intermediários no conjunto 1,2,...,k-1. De modo semelhante, p2 é um caminho mais curto até o vértice j com todos os vértices intermediários no conjunto 1,2,...,k-1.

A formulação recursiva seguindo a discussão acima é dada por:

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

3.2.1 Implementação

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

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

```
if __name__ == '__main__':
```

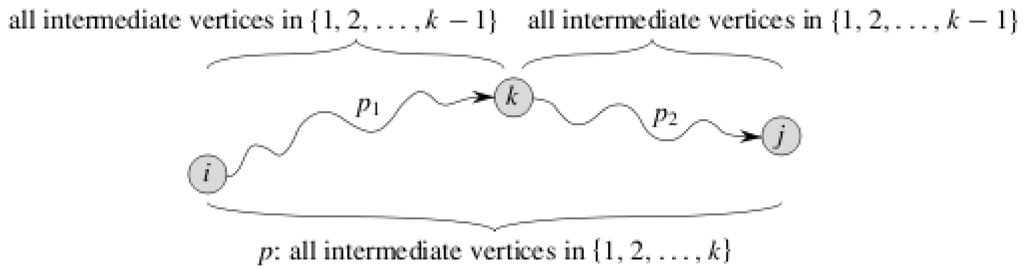


Figure 25.3 Path p is a shortest path from vertex i to vertex j, and k is the highest-numbered intermediate vertex of p. Path p_1, the portion of path p from vertex i to vertex k, has all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The same holds for path p_2 from vertex k to vertex j.

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

FLOYD-WARSHALL(W)

```
n ← rows[W]
D^(0) ← W
for k ← 1 to n
     do for i ← 1 to n
            do for j ← 1 to n
                   do d_ij^(k) ← min(d_ij^(k−1), d_ik^(k−1) + d_kj^(k−1))
return D^(n)
```

Fig. 3: Pseudo Código do algoritmo de Floyd-Warshall

```

    #Cria o grafo

    grafo = {'A':{'A':0,'B':INF,'C':12,'D':6,'E':4},

             'B':{'A':INF,'B':0,'C':INF,'D':1,'E':INF},

             'C':{'A':INF,'B':INF,'C':0,'D':15,'E':3},

             'D':{'A':INF,'B':INF,'C':15,'D':0,'E':4},

             'E':{'A':INF,'B':7,'C':INF,'D':INF,'E':0}

             }
```

A funcão 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.

```python
def floydWarshall(grafo):

    nos = grafo.keys()

    distancia = {}

    #armezena a distancia do no n para o no k
    for n in nos:

        distancia[n] = {}

        for k in nos:

            distancia[n][k] = grafo[n][k]
    # Floyd-warshal - programacao dinamica
    for k in nos:

        for i in nos:

            for j in nos:

                if distancia[i][k] + distancia[k][j] <
                    distancia[i][j]:

                     distancia[i][j] = distancia[i][k]+
                        distancia[k][j]
    # imprime a tabela com resultado
    printSolution(distancia)
```

4 Busca em Largura ou Breadth-First-Search

4.1 Definição

O algoritmo para busca em largura é essencialmente o algoritmo de Dijkstra onde todas as arestas possuem peso igual a 1. Quando percorremos um vértice salvamos ele para posteriormente percorrer os seus vizinhos que ainda não foram visitados, até terminar o grafo. Usamos desta vez uma lista FIFO para saber quem já foi visitado e quem não foi, e um vetor d[] contendo as distâncias do vértice para o vértice raiz.

4.2 Implementação

A implementação deste algoritmo de busca em largura, foi feito em Java, fez uso de matriz de adjacências binárias 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 8.

```
Data: G,w,s
BFS(G,s);
foreach todos os vértices v do
    d[v] = ∞;
end
d[raiz] = 0;
Qfifo = raiz;
while Qfifo != ∅ do
    u = Q.removeTopoDaFila;
    foreach v visinhos de u do
        if d[v] == infinito then
            d[v] = d[u] + 1;
            Q.enfileira(v);
        end
    end
end
```

Algoritmo 4: Pseudo Código do Algoritmo de Dijkstra

4.3 Análise de Complexidade

Como cada vértices e cada aresta são visitados uma única vez a complexidade é O(V+E), onde V é a quantidade de vértices e E a quantidade de arestas. Se o grafo for um grafo conexo, então teremos nosso $E >= V - 1$. Neste caso, a complexidade do algoritmo de busca em largura será O(E), para E a quantidade de arestas.

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

5.1 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 (selecionando 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 5:

```
Data: Inicio, Alvo
function BuscaProfundidade;
empilha(Pilha,Inicio);
while Pilha is not empty do
    varNodo ← desempilha(Pilha);
    Colore(Nodo, Cinza);
    if Nodo = Alvo then
        return Nodo;
    end
    for Filho in Expande(Nodo) do
        if Filho.cor = Branco then
            empilha(Pilha, Filho);
        end
    end
    Colore(Nodo, Preto);
end
```

Algoritmo 5: Pseudo-código do algoritmo de Busca em Profundidade

5.2 Implementação

A implementação deste algoritmo de busca em profundidade foi feito em Jave, 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 6.

```
   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 6: Pseudo-código para a implementação de Busca em Profundidade

5.3 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.

5.4 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), etnão 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 7: 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 $7 \rightarrow 11 \rightarrow 2 \rightarrow 9 \rightarrow 10 \rightarrow 5 \rightarrow 8 \rightarrow 3 \rightarrow 10$.

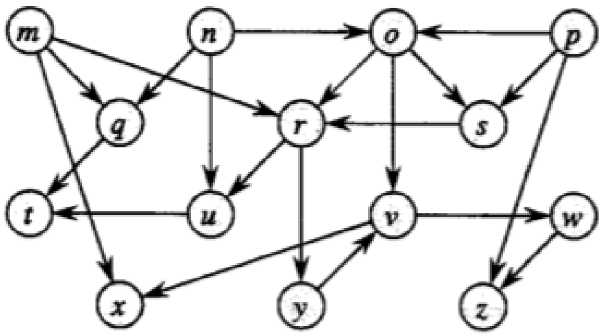


Fig. 4: Um gao para ordenação topológica

	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. 5: Matriz Adjacência para o grafo ao lado

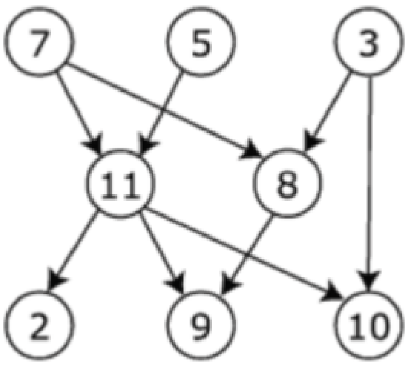
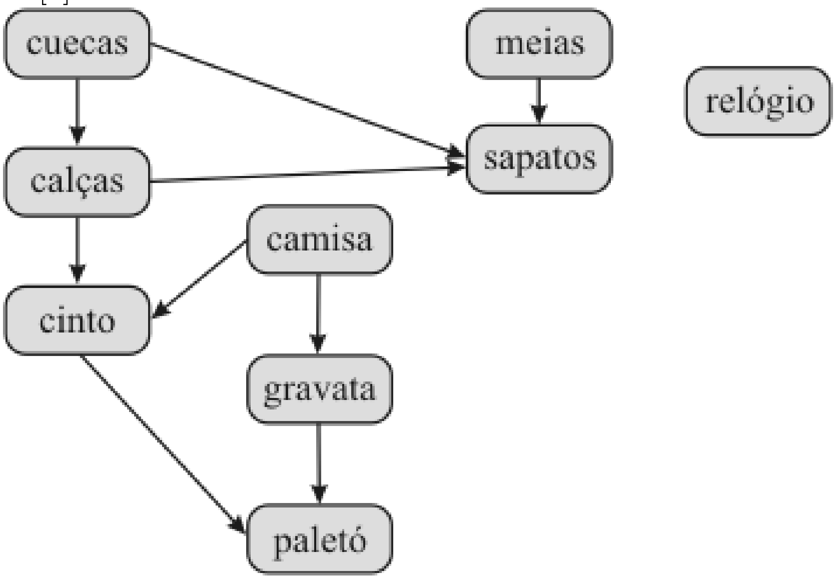


Fig. 6: Exemplo de grafo direcionado a ser ordenado por profundidade

Fig. 7: Exemplo de ordenação topológica que aplicamos sem perceber no cotidiano.[1]



6 Fechamento Transitivo

6.1 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 ares. 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.

6.2 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 Mr representando a relação R e tem como saída a matriz Mr* 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
return t(n);
```

Algoritmo 8: Pseudo-código do algoritmo de Warshall

6.3 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 = [w_{i,j}])i for k=1 to n do
    for i=1 to n do
        for j=1 to n do
            w_{i,j} = w_{i,j} ∨ (w_{i,k} ∧ w_{k,j});
        end
    end
end
return W;
```

Algoritmo 9: Pseudo-código da função principal do algoritmo de Warshall

6.4 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 dados, 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

```java
package br.ufba.graph.algorithm.minimumspanningtree;

import br.ufba.datastructures.UnionFind;
import br.ufba.datastructures.UnionFind.UnionElement;
import br.ufba.graph.Aresta.Status;
import br.ufba.graph.Aresta;
import br.ufba.graph.Graph;
import br.ufba.graph.Vertice;
import br.ufba.graph.algorithm.GraphAlgorithm;


/**
 * @author niltonvasques
 * http://en.wikipedia.org/wiki/Kruskal's_algorithm
 */
public class Kruskal implements GraphAlgorithm{

    private Graph mGraph;

    private int edges[];

    private int edgeIndex = 0;

    public Kruskal(Graph grafo) {
        mGraph = grafo;
    }

    @Override
    public void init(){
        edgeIndex = 0;
// crie uma floresta F (um conjunto de árvores), onde cada vértice
   no grafo é uma árvore separada

        for (int i = 0; i < mGraph.getVerticesCount() ;
           i++) {
            Vertice vertice = mGraph.getVertices()[i];
            UnionFind.makeSet(vertice);
        }

```

```java
// create a set S containing all the edges in the graph ordered by weight
        edges = new int[mGraph.getArestasCount()];
        for( int i = 0; i < edges.length; edges[i] = i++);
        qsort(0, edges.length -1);
    }



    @Override
    public boolean performStep() {
/*      enquanto S for nao-vazio, faca:
*           remova uma aresta com peso minimo de S
*           se essa aresta conecta duas arvores diferentes, adicione-a a floresta, combinando duas arvores numa unica arvore parcial
*           do contrario, descarte a aresta
*/
        if( edgeIndex >= edges.length )
            return true;

        Aresta aresta   = mGraph.getArestas()[ edges[edgeIndex]];
        if( aresta.status == Status.WAITING ){
            aresta.status = Status.PROCESSING;
            return false;
        }
        edgeIndex++;

        UnionElement u  = aresta.u;
        UnionElement v  = aresta.v;
        if( !UnionFind.find(u).equals(UnionFind.find(v)) ){
            aresta.status = Status.TAKED;
            UnionFind.union(u, v);
        }else{
            aresta.status = Status.DISCARDED;
        }
        return false;
    }

    private int partition(int left, int right) {
```

```java
        int pivot = mGraph.getArestas()[edges[(left +
            right) / 2]].weight;
        while (left <= right) {
            while (mGraph.getArestas()[edges[left]].
                weight < pivot)
                left++;
            while (mGraph.getArestas()[edges[right]].
                weight > pivot)
                right--;

            if (left <= right) {
                int i = left++;
                int j = right--;
                int k = edges[i];
                edges[i] = edges[j];
                edges[j] = k;
            }
        }
        return left;
    }

    protected void qsort(int left, int right) {
        if (left >= right)
            return;

        int i = partition(left, right);
        qsort(left, i - 1);
        qsort(i, right);
    }
}
```

Implementação do Algoritmo de Kruskal em Java

```java
package br.ufba.graph.algorithm.minimumspanningtree;

import br.ufba.datastructures.AdjacencyMatrix;
import br.ufba.graph.Aresta;
import br.ufba.graph.Aresta.Status;
import br.ufba.graph.Graph;
import br.ufba.graph.algorithm.GraphAlgorithm;

/**
 * @author niltonvasques
 * http://en.wikipedia.org/wiki/Prim%27s_algorithm
```

```java
 */
public class Prim implements GraphAlgorithm{
    private Graph mGraph;
    private AdjacencyMatrix verticesMatrix;
    private AdjacencyMatrix matrix;
    private boolean processing = false;

    public Prim( Graph graph) {
        mGraph = graph;
    }
    @Override
    public void init() {
        processing = false;
        verticesMatrix = new AdjacencyMatrix(mGraph.
           getVerticesCount());
        verticesMatrix.makeAdjacency(0, 0);
        matrix = mGraph.createAdjacencyMatrix();
    }

    @Override
    public boolean performStep() {
        Aresta bestChoice   = null;
        int bestVertice     = -1;
        int vertices[] = verticesMatrix.getAdjacencys(0)
           ;
        for( int v = 0; v < vertices.length; v++){
            if( vertices[v] == 1){
                int adjacencys[] = matrix.getAdjacencys(
                   v);
                for( int i = 0; i < adjacencys.length; i
                   ++){
                    if(adjacencys[i] == 1){
                        Aresta aresta = mGraph.getAresta
                           (i, v);
                        if( aresta != null){
                            if( !processing ){
                                aresta.status = Status.
                                   PROCESSING;
                            }else if ( bestChoice !=
                               null ){
                                if( bestChoice.weight >
                                   aresta.weight ){
```

```
                                    bestChoice  = aresta;
                                    bestVertice = i;
                                }
                            }else{
                                bestChoice = aresta;
                                bestVertice = i;
                            }
                        }
                    }
                }
            }
        }
        if(!processing){
            processing = true;
            return false;
        }
        boolean finish = true;
        if( bestVertice != -1 ){
            bestChoice.status = Status.TAKED;
            for(int i = 0; i < vertices.length; i++){
                if( vertices[i] == 1){
                    matrix.removeAdjacency(i, bestVertice);
                }else{
                    finish = false;
                }
            }
            processing = false;
            verticesMatrix.makeAdjacency(0, bestVertice);
        }
        return finish;
    }


}
```

Implementação do Algoritmo de Prims em Java

```
package br.ufba.graph.algorithm.mininumpath;

import java.util.ArrayList;
import java.util.List;
```

```java

import br.ufba.datastructures.AdjacencyMatrix;
import br.ufba.graph.Aresta;
import br.ufba.graph.Aresta.Status;
import br.ufba.graph.algorithm.GraphAlgorithm;
import br.ufba.graph.Graph;
import br.ufba.graph.Vertice;


/**
 * @author niltonvasques
 * Link: http://en.wikipedia.org/wiki/Dijkstra%27
    s_algorithm
 */
public class Dijkstra implements GraphAlgorithm{

    private static final int INFINITY = Integer.
       MAX_VALUE;
    private static final int UNDEFINED = -1;

    private Graph graph;
    private AdjacencyMatrix matrix;
    private Vertice source;
    private Vertice target;

    public Dijkstra(Graph graph) {
        this.graph = graph;
    }

    @Override
    public void init() {
        matrix = graph.createAdjacencyMatrix();
    }

    @Override
    public boolean performStep() {
        execute(source);
        return false;
    }

    public void setSource(int source){
        this.source = graph.getVertices()[source];
    }
```

```java

    public void setTarget(int target){
        try{
            this.target = graph.getVertices()[target];
        }catch(Exception e){
            this.target = null;
        }
    }

    private void execute(Vertice source){
        int[] dist = new int[graph.getVerticesCount()];
        int[] previous = new int[graph.getVerticesCount
            ()];

        for(int i = 0; i < graph.getVerticesCount(); i
            ++){
            dist[i] = INFINITY;

            previous[i] = UNDEFINED;
        }

        dist[source.index] = 0;

        List<Vertice> q = new ArrayList<Vertice>();
        for (int i = 0; i < graph.getVerticesCount(); i
            ++) {
            q.add(graph.getVertices()[i]);
        }


        while(!q.isEmpty()){

            Vertice u = menorDistancia(dist, q);
            q.remove(u);

            if(dist[u.index] == INFINITY)
                break;

            int[] adj = matrix.getAdjacencys(u.index);
            for(int v = 0; v < adj.length; v++){
                if(adj[v] == 1 && q.contains(graph.
                    getVertices()[v])){
```

```
                    int alt = dist[u.index] + graph.getAresta(u.index, v).weight;
                    if(alt < dist[v]){
                        dist[v] = alt;
                        previous[v] = u.index;
                        q.remove(graph.getVertices()[v]);
                        q.add(graph.getVertices()[v]);
                    }

                }
            }
        }

        if(target == null){
            for(int i = 0; i < dist.length; i++){
                    System.out.println(source.nome+" -> "+(i+1)+" = "+(dist[i] == INFINITY ? "INFINITO": dist[i]));
            }
        }else{
            System.out.println(source.nome+" -> "+target.nome+" = "+(dist[target.index] == INFINITY ? "INFINITO": dist[target.index]));
            clearPaths();
            paintPath(target.index, previous);
        }
    }

    private Vertice menorDistancia(int[] dist, List<Vertice> q) {
        Vertice menor = q.get(0);

        int menorDist = INFINITY;
        for(Vertice v : q){
            if(dist[v.index] < menorDist){
                menor = v;
                menorDist = dist[v.index];
            }
        }
        return menor;
    }
```

```java

    private void paintPath(int origem, int[] previous){
        if(previous[origem] != UNDEFINED){
            paintPath(previous[origem], previous);
            graph.getAresta(origem, previous[origem] ).
               status = Status.TAKED;
        }
    }

    private void clearPaths(){
        for (Aresta aresta : graph.getArestas()) {
            if(aresta != null)
                aresta.status = Status.WAITING;
        }
    }
}
```

Implementação do Algoritmo de Dijkstra em Java

```python
INF = 999999999

def printSolution(distGraph):

    string = "inf"

    nodes =distGraph.keys()

    for n in nodes:

        print "\t%6s"%(n),

    print " "

    for i in nodes:

        print"%s"%(i),

        for j in nodes:

            if distGraph[i][j] == INF:

                print "%10s"%(string),

            else:
```

```

                print "%10s"%(distGraph[i][j]),

        print" "

def floydWarshall(grafo):

    nos = grafo.keys()

    distancia = {}

    #armezena a distancia do no n para o no k
    for n in nos:

        distancia[n] = {}

        for k in nos:

            distancia[n][k] = grafo[n][k]
    # Floyd-warshal - programacao dinamica
    for k in nos:

        for i in nos:

            for j in nos:

                if distancia[i][k] + distancia[k][j] < distancia[i][j]:

                    distancia[i][j] = distancia[i][k]+ distancia[k][j]
    # imprime a tabela com resultado
    printSolution(distancia)

if __name__ == '__main__':

    #Cria o grafo

    grafo = {'A':{'A':0,'B':INF,'C':12,'D':6,'E':4},

             'B':{'A':INF,'B':0,'C':INF,'D':1,'E':INF},

             'C':{'A':INF,'B':INF,'C':0,'D':15,'E':3},
```

```
                'D':{'A':INF,'B':INF,'C':15,'D':0,'E':4},

                'E':{'A':INF,'B':7,'C':INF,'D':INF,'E':0}

                }

    floydWarshall(grafo)
```

Implementação do Algoritmo de Floyd em Python

```
#!/usr/bin/python
# vim: set fileencoding: utf-8 :

#Encontra para cada elemento do grafo, as arestas que chegam e saem
#Para cada par de aresta de chegada e saida, coloca 1 na matriz de
    saída

def warshall(p, n):
    for k in range(n):
        for i in range(n):
            for j in range(n):
                p[i][j]=max(p[i][j],(p[i][k]&p[k][j]))
    return

def max(a,b):
    if(a>b):
        return(a)
    else:
        return(b)


print("\n Quantidade de vertices e arestas,
    respectivamente:")
n= int(input(""))
e= int(input(""))

p = [[0 for i in range(n)] for i in range(n)]
for i in range(e):
    print("\n Vertices que a aresta incide %d:" % (i+1))
    u=int(input(""))
    v=int(input(""))
    p[(u-1)][(v-1)]=1

```

```python
print("\n Matrix adjacencia:")
for i in range(n):
    print(p[i])

warshall(p,n)

for i in range(n):
    for j in range(n):
        if(i==j):
            p[i][j]=1

print("\n Fechamento transitivo: \n")
for i in range(n):
    print(p[i])
```

Implementação do Algoritmo de Warshall em Python

```java
package br.ufba.graph.algorithm.search;

import java.util.Stack;

import br.ufba.datastructures.AdjacencyMatrix;
import br.ufba.graph.Graph;
import br.ufba.graph.Vertice;
import br.ufba.graph.Aresta.Status;
import br.ufba.graph.algorithm.GraphAlgorithm;

/**
 * @author niltonvasques
 * @author marinofull
 * link: http://www.ime.usp.br/~pf/analise_de_algoritmos
    /aulas/dfs.html
 */
public class DFS implements GraphAlgorithm{

    private Graph graph;
    private AdjacencyMatrix matrix;
    private Stack<Vertice> stack = new Stack<Vertice>();

    private int ordemBusca[];
    private int index = 0;

    public DFS(Graph graph) {
        this.graph = graph;
```

```
    }

    @Override
    public void init() {
        this.matrix = this.graph.createAdjacencyMatrix();

        for(int i = 0; i < graph.getVerticesCount(); i++){
            graph.getVertices()[i].status = Vertice.Status.BRANCO;
        }
        stack.push(graph.getVertices()[0]);
        graph.getVertices()[0].status = Vertice.Status.CINZA;

        index = 0;
        ordemBusca = new int[graph.getVerticesCount()];
        ordemBusca[index++] = next;
    }



    int next = 0;
    @Override
    public boolean performStep() {

        if(stack.isEmpty()) return true;

        boolean step = false;

        while(!step){
            Vertice u = stack.peek();

            if(next >= graph.getVerticesCount()){
                next = 0;
                stack.pop();
                u.status = Vertice.Status.PRETO;
                return false;
            }

            Vertice vertexV = graph.getVertices()[next];
```

```
            if(matrix.checkAdjacency(u.index, next) &&
               vertexV != null
                    && vertexV.status == Vertice.Status.
                       BRANCO){

                graph.getAresta(u.index, next).status =
                   Status.TAKED;
                vertexV.status = Vertice.Status.CINZA;
                stack.push(vertexV);
                ordemBusca[index++] = next;
                next = 0;
                step = true;
            }else{
                next++;
            }
        }

        return false;
    }

    @Override
    public String toString() {
        String result = "";
        for(int i = 0; i< ordemBusca.length; i++){
            result += " -> "+(ordemBusca[i]+1);
        }
        return result;
    }

}
```

Implementação da busca profundidade em Java

Código Fonte das Estruturas de Dados Implementadas no Projeto

```
package br.ufba.datastructures;


/**
 * @author niltonvasques
 * UnionFind data structure
 * http://en.wikipedia.org/wiki/Disjoint-
   set_data_structure
 */
public class UnionFind {
```

```java

    public interface UnionElement{

        public UnionElement getRoot();
        public UnionElement getParent();
        public void setRoot(UnionElement x);
        public void setParent(UnionElement x);

    }

    public static void makeSet( UnionElement x ){
        x.setParent(x);
    }

    public static UnionElement find( UnionElement x){
        if ( x.getParent() == x )
            return x;
        else
            return find(x.getParent());
    }

    public static void union( UnionElement x,
       UnionElement y){
        x.setRoot( find(x) );
        y.setRoot( find(y) );
        x.getRoot().setParent( y.getRoot() );
    }

}
```

Interface Para Operações Union-Find com Lista Encadeada

```java
package br.ufba.datastructures;

/**
 * @author niltonvasques
 * http://pt.wikipedia.org/wiki/Matriz_de_adjac%C3%
    AAncia
 */
public class AdjacencyMatrix {

    private static final int MEM_BLOCK_SIZE = 32;
    int matrix[];
    int stride;
```

```
    public AdjacencyMatrix(int n) {
        stride = n;
        int size = (int)(stride * stride);
        int lenght = 1 + (size/32);
        matrix = new int[ lenght ];
    }

    public void makeAdjacency(int element, int adjacency
        ){
        makeAdjacencyInternal(element, adjacency);
        makeAdjacencyInternal(adjacency, element);
    }

    public void removeAdjacency(int element, int
       adjacency ){
        removeAdjacencyInternal(element, adjacency);
        removeAdjacencyInternal(adjacency, element);
    }

    private void makeAdjacencyInternal(int element, int
       adjacency) {
        int bitIndex            = (element*stride+
           adjacency);
        int index       = (int) (bitIndex/MEM_BLOCK_SIZE
           );
        int shift       = bitIndex % MEM_BLOCK_SIZE;
        matrix[index]   |= (0x01 << shift);
    }

    private void removeAdjacencyInternal(int element,
       int adjacency) {
        int bitIndex            = (element*stride+
           adjacency);
        int index       = (int) (bitIndex/MEM_BLOCK_SIZE
           );
        int shift       = bitIndex % MEM_BLOCK_SIZE;
        matrix[index]   &= ~(0x01 << shift);
    }

    public int[] getAdjacencys(int element){
        int adjacencys[] = new int[stride];

        int bitIndex        = (element*stride);
```

```
        int bitRemainder    = (bitIndex % MEM_BLOCK_SIZE);

        for( int i = 0; i < stride; i++){
            int x = ((bitIndex+i)/MEM_BLOCK_SIZE);
            int y =  bitRemainder + i;

            int ret = (matrix[x] >> y) & 0x01;
            adjacencys[i] = ret;
        }

        return adjacencys;
    }

    public boolean checkAdjacency(int u, int v){

        int bitIndex         = (u*stride) + v;
        int index            = (int) (bitIndex/MEM_BLOCK_SIZE);

        int y = (bitIndex % MEM_BLOCK_SIZE);

        return ((matrix[index] >> y) & 0x01) == 0x01;
    }

    @Override
    public String toString() {
        String result = "";
        for(int i = 0; i < matrix.length; i++){
            result += "["+i+"]: "+Integer.toBinaryString(matrix[i]);
        }
        return result;
    }
}
```

Implementação da Matriz de Adjacências Binária

Referências

[1] J. Amgarten, P. Brito, and C. Rocha. Notas de aulas unicamp. http://www.ic.unicamp.br/~meidanis/courses/mo417/2003s1/aulas/2003-05-14.html, 2014.

[2] York College. Site das notas de aulas da york college os pa york. https://disciplinas.dcc.ufba.br/pub/MATA53/SemestreCorrente/Aula_04_TG_Isomorfismo.ppt, 2014.

[3] 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.

[4] Fernando Nogueira. Problema da Árvore Gerador Mínima. *UFJF*.

[5] 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*.

[6] UFBA. Site disciplinas dcc ufba. https://disciplinas.dcc.ufba.br/pub/MATA53/SemestreCorrente/Aula_04_TG_Isomorfismo.ppt, 2014.

[7] Nilton Vasques. Árvore geradora mínima - teoria dos grafos. https://github.com/niltonvasques/teoria-dos-grafos, 2013.