Apostila Resumida - Teoria dos Grafos

Fonte: http://faculty.ycp.edu/~dbabcock/PastCourses/cs360/lectures/

Adaptação: **Prof. Acauan C. Ribeiro**

Alunos: Kaio Guilherme Ferraz e Wandressa Reis

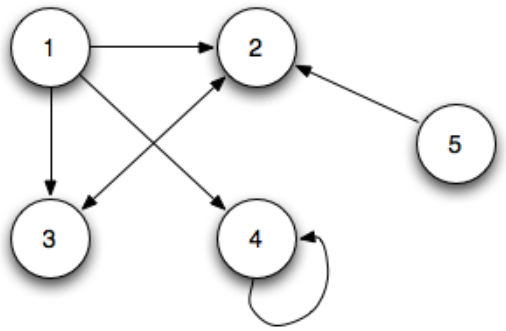
Este trabalho se propõe a fazer uma breve revisão sobre os principais assuntos relacionados a Grafo vistos na disciplina DCC 405 – Estrutura de Dados II.

1. Definição de Grafo

Assumimos que um **grafo G(V, E)** é representado como um par de conjuntos finitos onde **V é o conjunto de vértices e E é o conjunto de arestas**.

1.1 Gráfico dirigido (dígrafo)

Em um *grafo direcionado*, as arestas são representadas **por pares ordenados de vértices (u,v)** e mostrados de maneira gráfica como setas direcionadas (um vértice pode ser conectado a si mesmo por meio de um auto-loop).



Uma aresta *(u,v)* é **incidente** de(**sai**) *u* e é incidente de(**entra**) v. Se um grafo contém uma aresta (u,v), então v é adjacente a u e é representado notadamente como u→v. Note que v ser adjacente a u não implica que u seja adjacente a v, a menos que a aresta (v,u) ∈ E. Assim (u,v) e (v,u) são arestas distintas em um grafo direcionado.

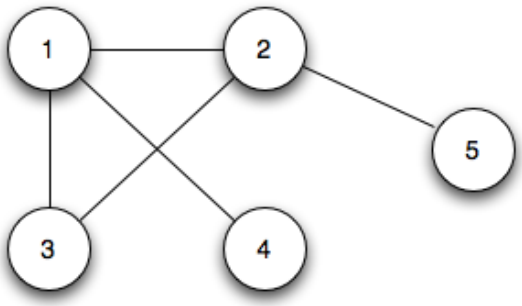
Dizemos que u e v são vizinhos se (u,v) ∈ E ou (v,u) ∈ E.

Para cada vértice definimos o **grau** de saída como o número de arestas que saem do vértice, o grau de entrada como o número de arestas que entram no vértice e o grau como o grau de saída + o grau de entrada (ou seja, o número total de arestas no vértice). Se um vértice tem grau= 0, então o vértice é isolado.

Se o grafo direcionado não tem auto-loops, então é um **grafo direcionado simples**.

1.2 Gráfico não direcionado

Em um grafo não direcionado, as arestas são representadas por pares não ordenados de vértices. Assim (u,v) e (v,u) representam a mesma aresta e são mostrados de maneira gráfica como simplesmente uma linha de conexão (observe que grafos não direcionados podem não conter auto-loops).



Uma aresta (u,v) é incidente em u e v, e u e v são adjacentes um ao outro.

O grau é o número de arestas incidentes em um vértice.

Para converter um grafo não direcionado em um direcionado, basta substituir cada aresta (u,v) por (u,v) e (v,u). Por outro lado, para converter um grafo direcionado em um não direcionado, substitua cada aresta (u,v) ou (v,u) por (u,v) e remova todos os auto-loops.

1.3 Caminhos

Um caminho de comprimento k de **u** para **u'** é uma sequência de vértices $< v_0 , v_1 , ..., v_k >$ com **u** = v_0 , u' = v_k , e (v_{i-1} , v_i) ∈ E .

Se existe um caminho p de u para u', então u' é alcançável a partir de u (denotado u ⇝ u' se G for um grafo direcionado).

O **caminho** é simples se todos os vértices forem distintos.

Um subcaminho é uma subsequência contígua $< v_i , v_{i+1} , ..., v_j >$ com $0 \le i \le j \le k$.

Um **ciclo** é um caminho com $v_0 = v_k$ (e também é simples se todos os vértices, exceto os pontos finais, forem distintos). Um grafo **acíclico** é um grafo sem ciclos.

1.4 Componentes conectados

Em um grafo não direcionado, um **componente conectado** é um *subconjunto* de vértices que são todos alcançáveis uns pelos outros. O **grafo é conexo** se contiver exatamente um componente conexo, ou seja, todos os vértices são alcançáveis a partir de todos os outros.

Em um grafo direcionado, um componente fortemente conectado é um subconjunto de vértices mutuamente alcançáveis, ou seja, existe um caminho entre quaisquer dois vértices no conjunto.

1.5 Gráficos Especiais

Um **grafo completo** é um grafo não direcionado onde todos os vértices são adjacentes a todos os outros vértices, ou seja, existem arestas entre cada par de vértices.

Um **grafo bipartido** é um grafo não direcionado que pode ser particionado em V_1 e V_2 tal que para cada aresta (u , v) ∈ E ou

$$u \in V_1 \text{ ev } \in V_2 \text{ OU } u \in V_2 \text{ ev } \in V_1 \_ \_$$

ou seja, o grafo pode ser separado de forma que as únicas arestas estejam entre vértices em diferentes subconjuntos.

Uma **floresta** é um grafo acíclico não direcionado. Se também estiver conectado, então é uma **árvore** .

Um grafo **acíclico direcionado é conhecido como DAG** .

2. Representação do gráfico

Dois métodos comuns para implementar um gráfico em software é usar uma **lista de adjacências** ou uma **matriz de adjacências** .

2.1 Lista de Adjacência

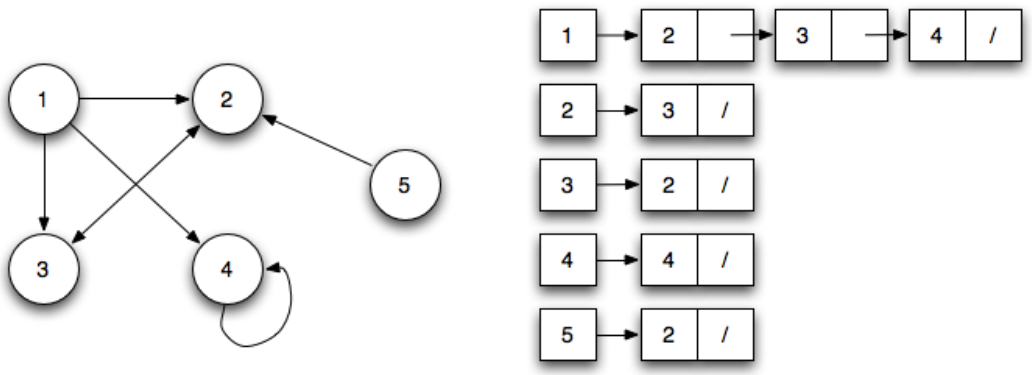
Em uma implementação de lista de adjacência, nós simplesmente armazenamos os vértices adjacentes (ou seja, arestas) para cada vértice em uma lista encadeada denotada por Adj [u]. Se somarmos os comprimentos de todas as listas de adjacência, obtemos

$$\sum |\text{Adj}[u]| = \begin{cases} |E| & \text{if directed} \\ 2|E| & \text{if undirected} \end{cases}$$

⇒ O armazenamento Θ(V + E) é necessário.

Esta representação é boa para gráficos esparsos onde | E | ≪ | V | ^{2} . Uma desvantagem é que determinar se uma aresta (u , v) ∈ E requer uma busca de lista ⇒ Θ(V).

Para o grafo direcionado original, a lista de adjacências seria



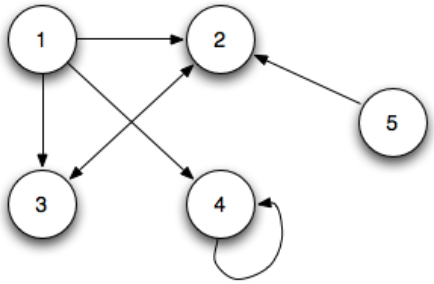
2.2 Matriz de adjacência

Em uma implementação de matriz de adjacência, armazenamos as arestas em uma matriz VxV A como valores binários ou números reais para arestas ponderadas.

⇒ O armazenamento Θ(V^2) é necessário (independente de E).

Esta representação é boa para gráficos densos onde $|E| \approx |V|^2$. A vantagem é que leva apenas $\Theta(1)$ tempo para determinar se uma aresta $(u, v) \in E$, pois é simplesmente um acesso a um elemento da matriz. Se o gráfico não é direcionado, então $\mathbf{A = A^T}$, portanto, apenas a metade triangular superior precisa ser armazenada.

Para o grafo direcionado original, a matriz de adjacência seria



	1	2	3	4	5
1	0	1	1	1	0
2	0	0	1	0	0
3	0	1	0	0	0
4	0	0	0	1	0
5	0	1	0	0	0

3.1 Breadth-First Search (BFS) – Busca em Largura

Agora que revisamos a terminologia básica associada aos grafos, o primeiro algoritmo que investigaremos é ***a busca em largura*** (BFS). Este algoritmo é usado para encontrar os caminhos mais curtos (por número de arestas) para cada vértice alcançável a partir de um determinado vértice dado.

3.1.1 Problema

Dado um vértice de origem **s**, encontre os *caminhos mais curtos* (em termos de número de arestas) para cada vértice alcançável por **s**.

3.1.2 Solução

O procedimento que usaremos encontrará todos os vértices alcançáveis a uma **distância k** antes de descobrir os vértices alcançáveis a uma **distância k+1**. Em última análise, o algoritmo produzirá uma árvore em largura com **s** como **raiz**.

Durante a execução do algoritmo, os vértices serão coloridos (denotados por u.color). As cores representam o estado atual do vértice da seguinte forma

- branco - o vértice não foi descoberto (ou seja, atualmente nenhum caminho foi encontrado para o vértice)
- cinza - o vértice foi descoberto e está na *fronteira*, ou seja, pode haver outros vértices que podem ser descobertos
- preto - o vértice foi descoberto e foi completamente pesquisado

O algoritmo também usa dois campos adicionais para cada vértice

u.π - vértice predecessor

u.d - distância quando o vértice é descoberto pela primeira vez (e subsequentemente é a distância mais curta da fonte)

Empregaremos uma **fila Q** que irá rastrear quais vértices estão atualmente sob descoberta. Assim, os vértices que ainda não foram colocados em Q serão brancos, os que estiverem em que serão cinzas e os que foram removidos de que serão pretos .

3.1.3 Algoritmo

O algoritmo para busca em largura é

```
BFS(G,s)
1. para cada vértice u ∈ GV - {s}
2. u.cor == BRANCO
3. ud = INF
4. u.pi = NIL
5. s.cor = CINZA
6. dp = 0
7. s.pi = NIL
8. Q = ∅
9. ENFILEIRA(Q,s)
10. enquanto Q ≠ ∅
11. u = DEQUEUE(Q)
12. para cada v ∈ G.Adj[u]
13. se v.cor == BRANCO
14. v.cor = CINZA
15. vd = ud + 1
16. v.pi = u
17. ENFILEIRA(Q,v)
18. u.cor = PRETO
```

Basicamente, o algoritmo realiza as seguintes operações:

1. Inicialize Q com o vértice de origem s
2. Retire o vértice principal u de Q e marque como **preto**
3. Enfileire todos os vértices brancos adjacentes a u marcando-os como **cinza** , defina sua distância para a distância de u + 1 e defina seu π para u
4. Repita 2-3 até Q = ∅

3.1.4 Análise

Como nenhum vértice é enfileirado/retirado da fila mais de uma vez ⇒ O(V)

Cada lista de adjacências é escaneada apenas uma vez (quando o vértice é desenfileirado) com tamanho máximo o número total de arestas ⇒ O(E)

Sobrecarga de inicialização ⇒ O(V)

Assim, o tempo total de execução para BFS é

$$\Rightarrow O(V)+O(E)+O(V)=O(V+E)$$

Pode-se provar que o algoritmo produz os **caminhos mais curtos** (em termos do número mínimo de arestas) para todos os vértices alcançáveis da fonte s. Esses caminhos podem ser representados por uma árvore em largura que é dada pelo subgrafo predecessor

$$G_\pi(V_\pi,E_\pi) \text{ where}$$
$$V_\pi=\{v\in V:\pi(v)\neq nil\}\cup\{s\}$$
$$E_\pi=\{(\pi(v),v):v\in V_\pi-\{s\}\}$$

Em outras palavras, o grafo predecessor contém todos os vértices com predecessores alcançáveis mais a fonte e todas as arestas predecessoras.

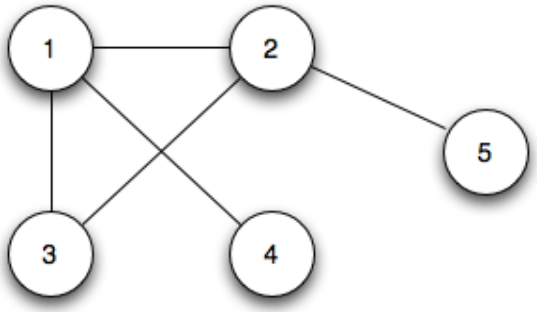
Além disso, pode-se mostrar que como o subgrafo predecessor é uma árvore , pelo teorema B.2 do CLRS

$$|E_\pi| = |V_\pi| - 1$$

A árvore predecessora pode ser percorrida (usando os π's) para fornecer o caminho mais curto de s para v .

3.1.5 Exemplo

Considere o gráfico de cinco nós (não direcionado)

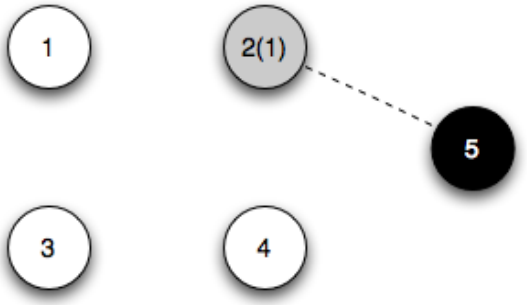


Se selecionarmos o vértice 5 como fonte, então d[5]=0, π[5]=/, Q={5}, então a inicialização nos dá

Q = {5}

	1	2	3	4	5
d	∞	∞	∞	∞	0
π	/	/	/	/	/

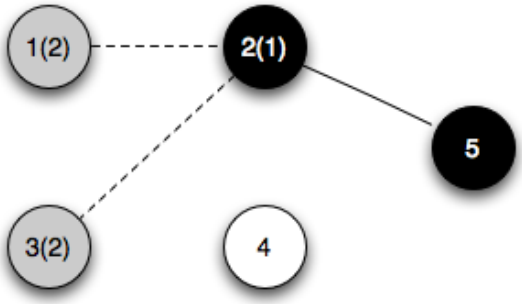
Iteração 1 : desenfileirar o vértice 5 e enfileirar o vértice 2



Q = {2}

	1	2	3	4	5
d	∞	1	∞	∞	0
π	/	5	/	/	/

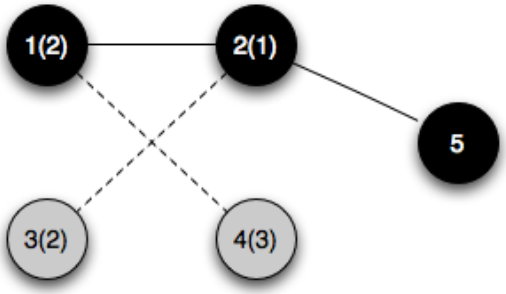
Iteração 2 : desenfileirar vértice 2 e enfileirar vértices 1,3



Q = {1,3}

	1	2	3	4	5
d	2	1	2	∞	0
π	2	5	2	/	/

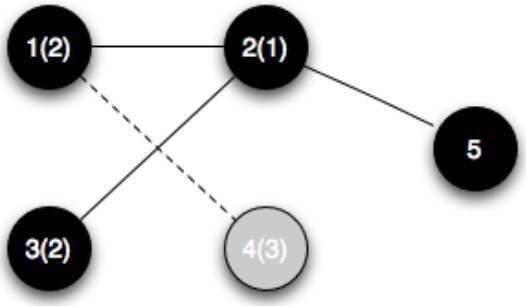
Iteração 3 : desenfileirar o vértice 1 e enfileirar o vértice 4



Q = {3,4}

	1	2	3	4	5
d	2	1	2	3	0
π	2	5	2	1	/

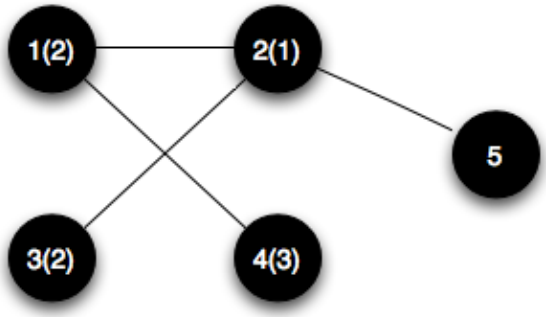
Iteração 4 : desenfileirar o vértice 3 e enfileirar nenhum vértice



Q = {4}

	1	2	3	4	5
d	2	1	2	3	0
π	2	5	2	1	/

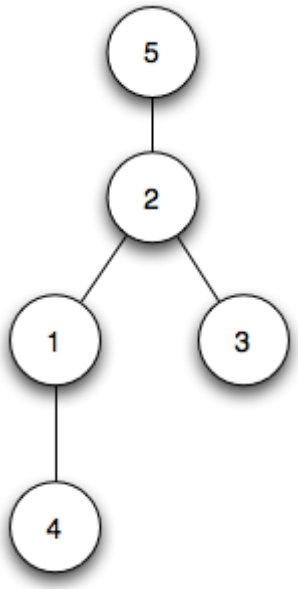
Iteração 5 : desenfileira o vértice 4 e não enfileira nenhum vértice (deixando assim a fila vazia)



Q = ∅

	1	2	3	4	5
d	2	1	2	3	0
π	2	5	2	1	/

O grafo predecessor final, ou seja, árvore em largura, para este BFS é



3.2 Depth-First Search (DFS) – Busca em Profundidade

A Busca em Profundidade é um tipo de busca onde todos ou quase todos os vértices atingíveis de um grafo são visitados por meio de suas arestas. Quando o último vértice é alcançado, acontece um backtracking. A cada backtracking ocorrido, é feita uma varredura dos vértices irmãos, possibilitando que todos os vértices sejam alcançados.
O objetivo da busca em profundidade é percorrer o grafo até que se encontre o objeto de busca, ou até que toda a estrutura atingível seja percorrida.

3.2.1 Problema

Tendo um grafo qualquer G=(V, E), fazer uma busca em profundidade.

3.2.2 Solução

Inicialmente, seleciona-se e visita-se um vértice s pertencente a V (vértice inicial). Então explora-se qualquer aresta (s, w) incidente em s, visita-se w.

Como passo geral, um vértice qualquer w a ser explorado, que seja adjacente ao vértice mais recente visitado v, poderá já ter sido visitado antes ou nao. Se w já foi visitado, então retorna-se a v e escolhe-se uma outra aresta incidente em v, e ainda não explorada. Se w não foi visitado, visita-se w e aplica-se o processo recursivamente sobre w. Após completar a busca através de todas as arestas incidentes em w, retorna-se a v, o vértice do qual w foi alcançado primeiro.

Este processo de selecionar arestas não exploradas incidentes em v, continua até que a lista dessas arestas acabe. Assim, a busca em profundidade irá terminar quando voltamos a s (vétice inicial) e não encontramos nenhuma aresta a ser explorada.

3.2.3 Algoritmo

O Algoritmo de Busca em Profundidade é:

```
DFS(G, u)

      u.visitou = true

      para cada v ∈ G.Adj[u]

            if v.visitou == false

                  DFS(G, v)

iniciar(){

      Para cada u ∈ G

            u.visitou = false

      Para cada u ∈ G

            DFS(G, u)

}
```

Basicamente, o Algoritmo realiza as seguintes operações:

1. Coloca qualquer um dos vértices do grafo no topo da lista;
2. Pega um item do topo da lista e adiciona-o à lista visitada do vértice;
3. Cria uma lista do nó adjacente do vértice;
4. Adiciona os nós não visitados ao topo da lista de nós visitados.

3.2.4 Análise

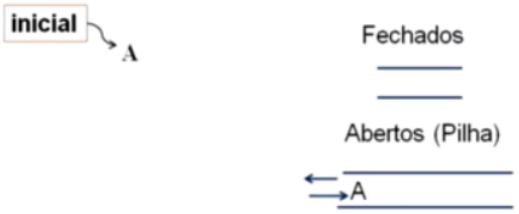
Para cada vértice, a rotina de busca é invocada apenas uma vez, com argumento v. A execução da rotina com argumento v examina, diretamente, todas as arestas que saem de v e apenas essas. Portanto, na união de todas as execuções de DFS, cada aresta do grafo é examinada apenas uma vez. Como o exame de cada aresta consome uma quantidade de tempo que não depende do tamanho do grafo, o tempo total gasto nas execuções de DFS é O(m), sendo m o número de arcos do grafo.

Concluímos assim que o algoritmo de Busca em Profundidade consome O(n+m) unidades de tempo. Como n+m é o tamanho do grafo, o algoritmo é linear.

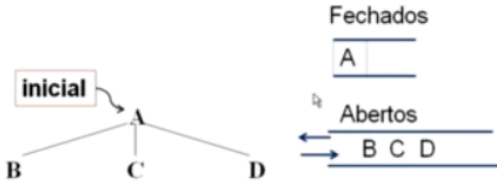
3.2.5 Exemplo

Fechados = Lista de nós visitados

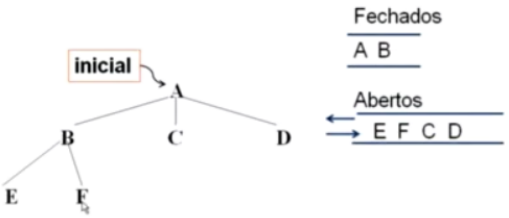
1. Tem-se um nó inicial A que é inserido na pilha, enquanto a lista de Fechados ainda está vazia;



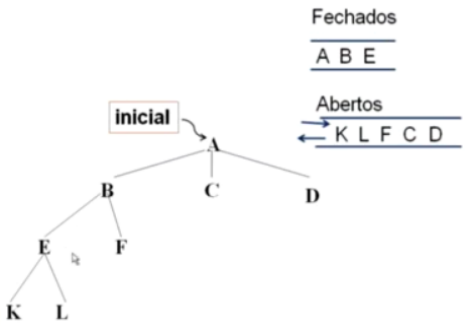
2. Quando o nó A é adicionado à lista de Fechados, são gerados três novos nós B, C, D que logo são inseridos na lista de na pilha dos nós Abertos;



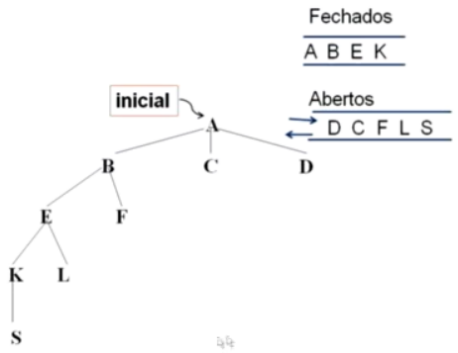
3. Em seguida, o primeiro nó da pilha é adicionado à lista de Fechados e é expandindo, recebendo E e F que são acrescentados ao topo da pilha;



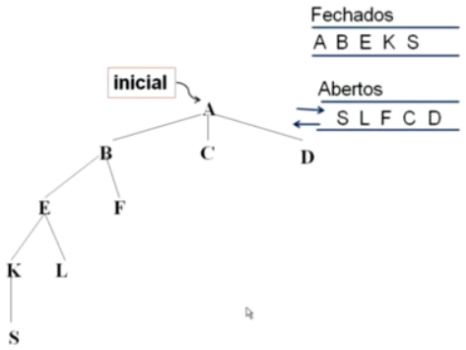
4. Como o nó é está no topo da pilha ele será adicionado à lista de Fechados e será expandido, recebendo os nós K e L, esses mesmos nós também são adicionados ao topo da pilha de nós Abertos ficando na frente dos nós que já estavam lá;



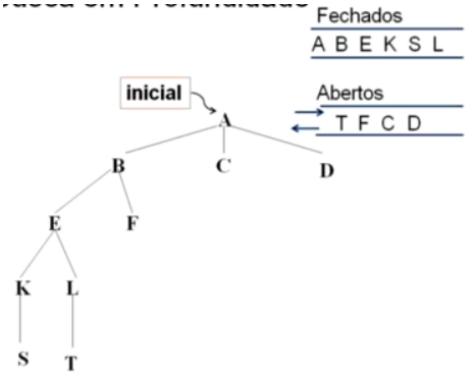
5. Adicionado o nó K à lista de Fechados, é feita a sua expansão e é encontrado um único estado novo que é o nó S que será adicionado à pilha;



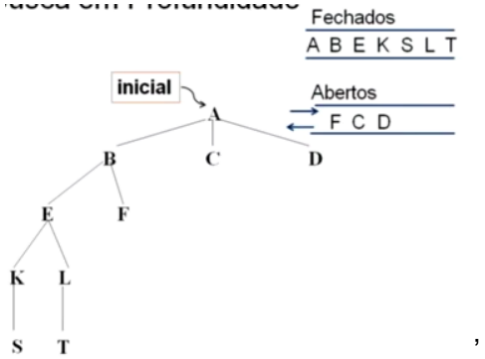
6. É concebido que o nó S tem a característica de não gerar novos nós, portanto, o nó S apenas passa para a lista de nós fechados porque ele foi expandido, no entanto, não apareceram novos nós;



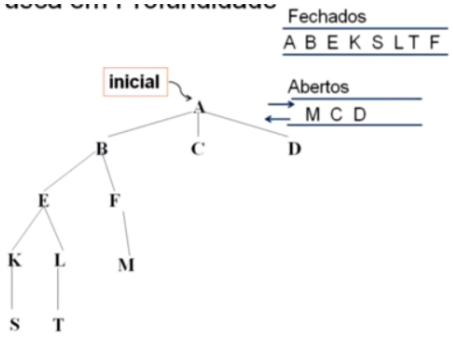
7. Em seguida, o próximo nó expandido é o nó L, que gera um novo estado apenas que é o nó T, logo, o nó T entra no topo da pilha;



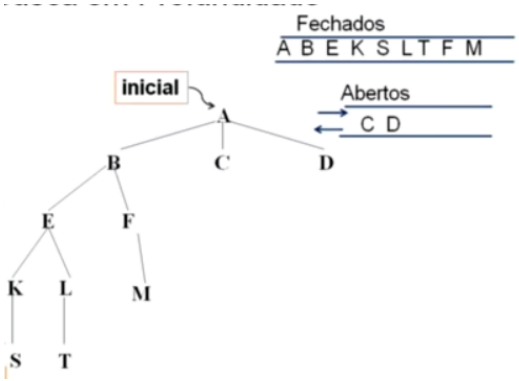
8. Com a expansão do nó T, é percebido que ele não gera nenhum novo nó, indo para a lista de nós fechados e não acrescentando novos nós à lista de nós abertos;



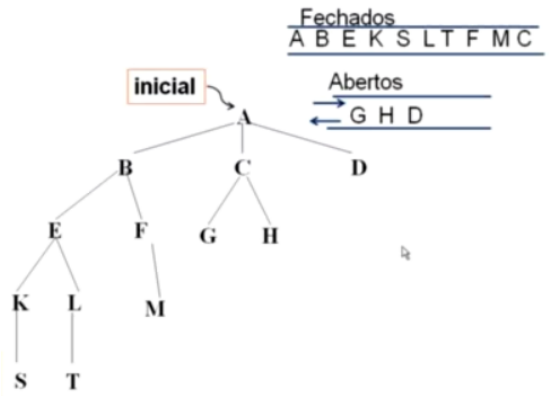
9. O próximo nó a ser expandido é o nó F, que gera o estado M, que é adicionado ao topo da pilha;



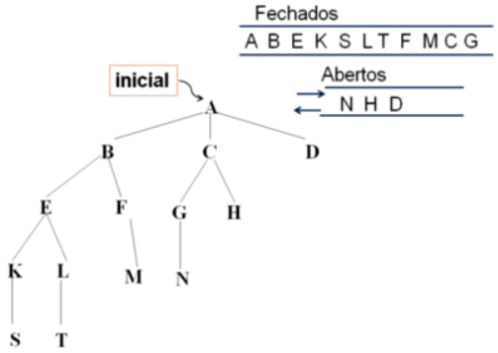
10. Em seguida, o nó expandido é o nó M que estava no topo da pilha, mas no entanto não produz novos estados, portanto é apena adicionado à lista sem produzir novos nós;



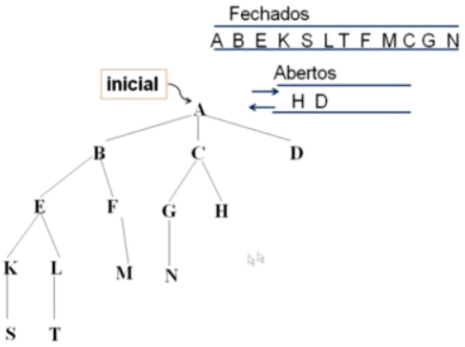
11. Como o item do topo da pilha é o nó C, ele será expandido. Nota-se que ele gera dois novos nós G e H que são adicionados ao topo da pilha;



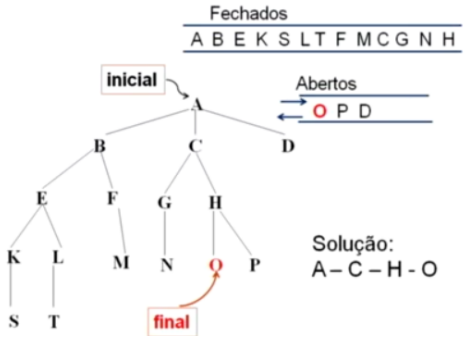
12. Temos a expansão do nó G que gera o nó N que é adicionado ao topo e o nó G vai pra lista de nós fechados;



13. O próximo nó a ser expandido é o nó N, que no caso não possui nenhum nó filho sendo apenas adicionado à lista de nós Fechados;



14. A expansão do nó H gera os estados O e P, e o procedimento de busca se encerra, pois é encontrado o nó objetivo, o nó O. Com isso, temos a solução sendo o caminho que vai do nó inicial até o nó objetivo: A - C - H - O.



4. Árvore geradora mínima

A Árvore Geradora Mínima é uma árvore de extensão com peso menor ou igual a cada uma das outras árvores de extensão possíveis. Generalizando mais, qualquer grafo não direcional (não necessariamente conectado) tem uma floresta de árvores mínimas, que é uma união de árvores de extensão mínimas de cada uma de suas componentes conexas.

Um exemplo de uso de uma árvore de extensão mínima seria a instalação de fibras óticas num campus de uma faculdade. Cada trecho de fibra óptica entre os prédios possui um custo associado (isto é, o custo da fibra, somado ao custo da instalação da fibra, mão de obra, etc). Com esses dados em mãos (os prédios e os custos de cada trecho de fibra ótica entre todos os prédios), podemos construir uma árvore de extensão que nos diria um jeito de conectarmos todos os prédios sem redundância. Uma árvore geradora mínima desse grafo nos daria uma árvore com o menor custo para fazer essa ligação.

4.1 Algoritmo de Prim

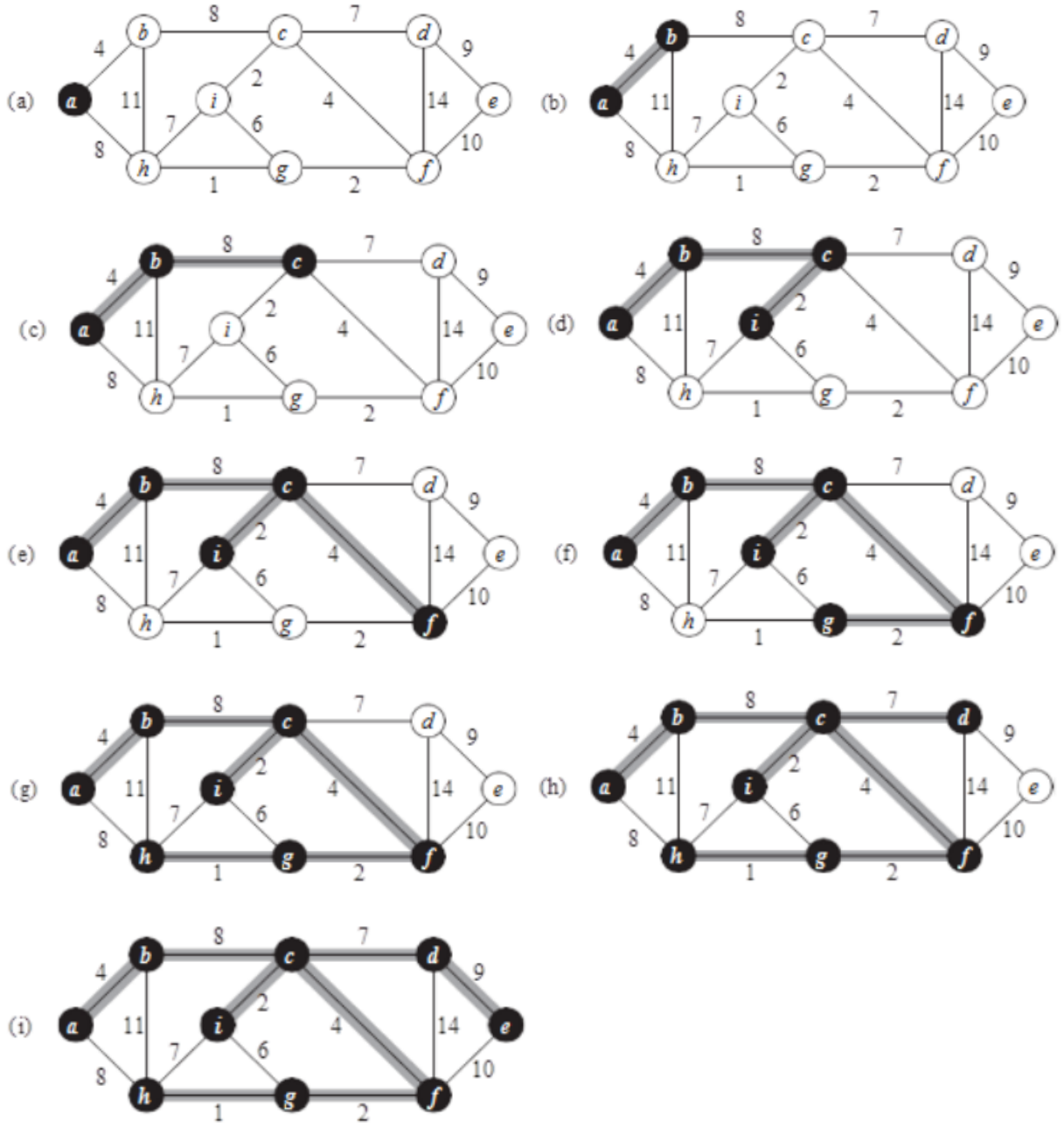
O Algoritmo de Prim é um algoritmo de caráter guloso: em cada iteração, abocanha a aresta mais barata conectada sem se preocupar com o efeito global, a longo prazo, dessa escolha. A prova de que essa estratégia está correta decorre do critério de minimalidade baseado em cortes.

4.1.1 Algoritmo

```
Prim(G, w, r)
 1   for cada vértice u ∈ V(G)
 2       chave[u] ← ∞
 3       π(u) ← null
 4   chave[r] ← 0
 5   Q ← V(G)
 6   while Q != 0
 7       u ← Extract-Min(Q) ## menor chave[u]
 8   for cada v ∈ Adj[u]
 9       if v ∈ Q and w(u, v) < chave[v]
 10           π(v) ← u
```

```
11            chave[v] ← w(u, v)
```

4.1.2 Forma exemplificada do Algoritmo de Prim.



O algoritmo de PRIM inicia-se conectando o nó a. A partir da conexão do nó a define-se o conjunto de nós candidatos a serem conectados, que são todos os ramos que possuem um nó conectado e outro aberto, nesta ilustração são os ramos (a-b) e (a-h). Dentro do conjunto de nós candidatos, escolhe-se aquele que possui ramo de menor custo (a-b). A cada iteração o algoritmo retorna ao ponto inicial para verificação dos pesos dos ramos candidatos, em caso de empate, escolhe-se o primeiro ramo verificado. Pode-se observar esta condição na etapa (b), onde os ramos candidatos são os ramos (a-h), (b-c) e (b-h), tem-se empate entre os ramos sendo o

primeiro deles escolhido. O algoritmo de PRIM encerra com a formação da árvore geradora mínima.

4.2 Algoritmo de Kruskal

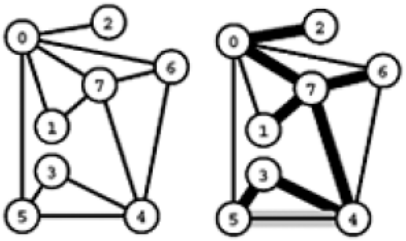
O algoritmo de Kruskal é um algoritmo em teoria dos grafos que busca uma árvore geradora mínima para um grafo conexo com pesos. Isto significa que ele encontra um subconjunto das arestas que forma uma árvore que inclui todos os vértices, onde o peso total, dado pela soma dos pesos das arestas da árvore, é minimizado.

4.2.1 Algoritmo

```
Kruskal(G, w)
1 A ← 0
2 for cada vértice v ∈ V(G)
3     MakeSet(v)
4 ordena E de maneira crescente pelo peso w
5 for cada aresta (u, v) ∈ E, em ordem crescente de w
6     if FindeSet(u) != FindSet(v)
7           A ← A uniao {(u, v)}
8           Union(u, v)
```

4.2.2 Exemplo

Considere o problema de encontrar uma MST no grafo não-dirigido com custos nas arestas definido a seguir. (Para simplificar o rastreamento, as arestas são apresentadas em ordem crescente de custo.) Em seguida, veja o rastreamento da execução do algoritmo de Kruskal. Cada linha da tabela registra, no início de cada iteração, as arestas da floresta e o custo da floresta.



3-5 1-7 6-7 0-2 0-7 0-1 3-4 4-5 4-7 0-6 4-6 0-5
0 1 2 3 4 5 6 7 8 9 9 11

floresta	custo
-	0
3-5	0+0
3-5 1-7	0+1
3-5 1-7 6-7	1+2
3-5 1-7 6-7 0-2	3+3
3-5 1-7 6-7 0-2 0-7	6+4
3-5 1-7 6-7 0-2 0-7 3-4	10+6
3-5 1-7 6-7 0-2 0-7 3-4 4-7	16+8

A última linha dá as arestas de uma MST. Resumindo, o algoritmo de Kruskal escolhe as seguintes arestas para formar uma MST:

3-5 1-7 6-7 0-2 0-7 3-4 4-7
0 1 2 3 4 6 8
(Os espaços em branco correspondem às arestas que foram rejeitadas porque formam um circuito com as arestas escolhidas em iterações anteriores.)

5. Problema do Menor Caminho

O problema do caminho mínimo consiste basicamente em: dado um grafo com pesos nas arestas, obter o caminho de menor custo entre dois vértices x e y. Como muitas vezes o peso representa a distância entre os vértices este problema passou a ser conhecido como problema do caminho mínimo.

5.1 Algoritmo de Dijkstra

O Algoritmo de Dijkstra é um algoritmo que calcula o caminho mais curto em termos de peso total de arestas, entre o vértice inicial e todos os demais vértices do grafo, onde o peso total das arestas é a soma dos pesos das arestas que compõem o caminho.

Para cada vértice v do grafo, mantemos um atributo d[v] que é um limite superior para o peso do caminho mais curto do nó inicial s a v.

Dizemos que d[v] é uma estimativa de caminho mais curto inicialmente feito “infinito”.

Também armazenamos o vértice que precede v(p[v] - precedente de v) no caminho mais curto de s a v.

5.1.1 Exemplo

Siga as inicializações apontadas:

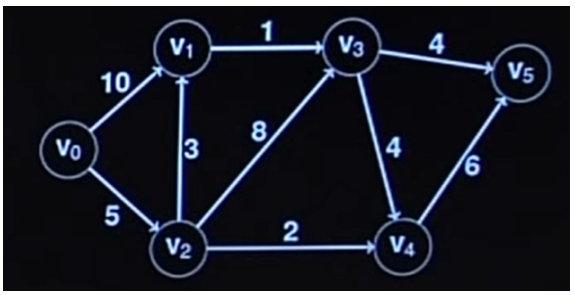
Enquanto houver vértice aberto:

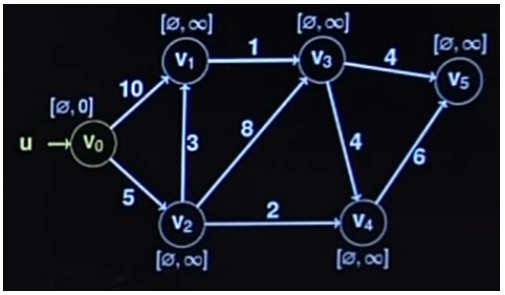
Escolha u cuja estimativa seja a menor dentre os abertos;

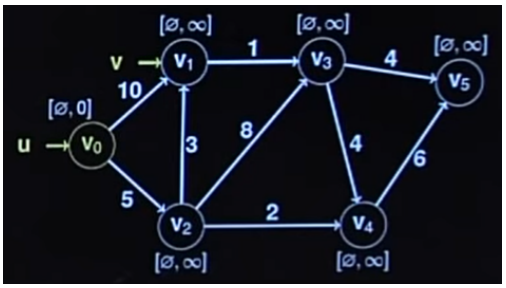
Feche u;

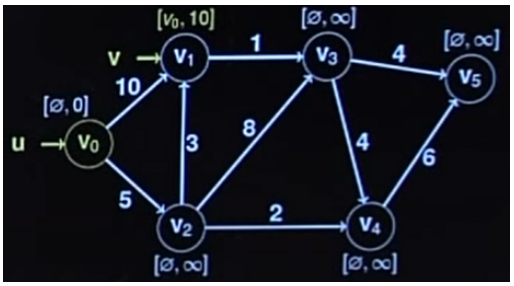
Para todo nó aberto v na adjacência de u:

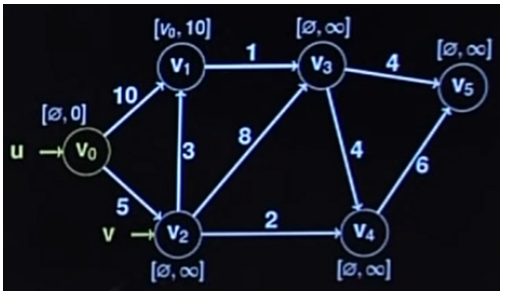
Relaxe a aresta (u,v)

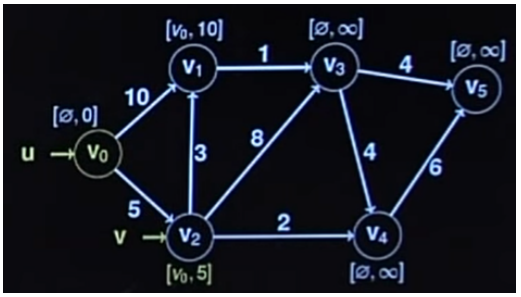












Acontecendo assim de forma sucessiva até que se ache a menor distância entre os pontos requisitados.

Observações:

Fechamos um nó u somente se já conhecemos o menor caminho deste ao nó inicial s;

Então fazemos o relaxamento de seus vizinhos:

Sabemos a menor distância de s a u;

Temos a estimativa de s a v, v vizinho de u;

Buscamos saber a distância de s a u mais a distância de (u, v) é a melhor que a estimativa atual de s a v. Se for, atualizamos essa estimativa.

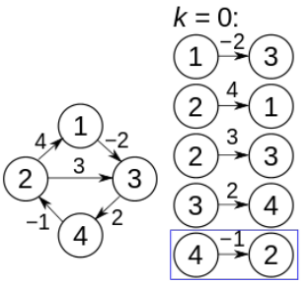
5.2 Algoritmo de Floyd-Warshall

O Algoritmo de Floyd-Warshall é um algoritmo que tem por objetivo encontrar o menor caminho entre todos os vértices. É aplicado quando é necessário calcular a menor distância entre todos os pares de vértices em um grafo direcionado e ponderado. Este algoritmo retorna uma matriz de caminhos mínimos e também trabalha com arestas de pesos negativos.

5.2.1 Exemplo

Na iteração k=0;

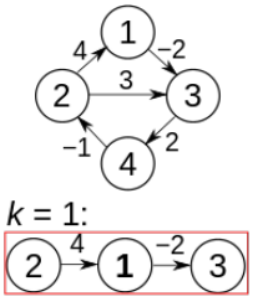
Somente os caminhos representados por uma única adjacência no grafo são conhecidos.



Na iteração k=1;

Todos os caminhos que passam pelo vértice 1 são descobertos;

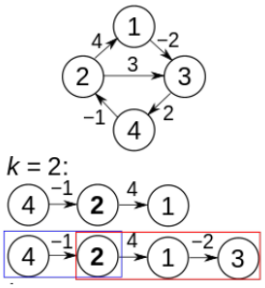
Em particular, o caminho [2, 1, 3] substitui o caminho [2, 3].



Na iteração k=2;

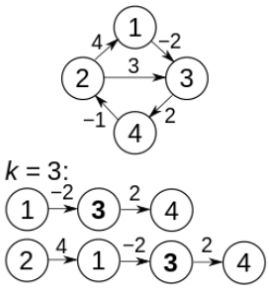
Todos os caminhos que passam pelo vértice 2 ou pelos vértices 2 e 1 são descobertos;

Em particular, o caminho [4, 2, 3] não é considerado, dado que [4, 2, 1, 3] não é um caminho mais curto até então.



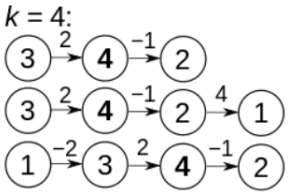
Na iteração k=3;

Todos os caminhos que passam pelo vértice 3 ou pelos vértices 3 e 2 ou 1 são descobertos.



Finalmente, em k=4;

Todos os caminhos mais curtos são determinados.



5.3 Algoritmo A*

O problema que um algoritmo A∗ procura resolver é encontrar uma sequência de passos (= caminho) que leva de uma dado nó inicial s a um destino t em um grafo.

O grafo é tipicamente muito grande e muitas vezes não pode ser dado explicitamente. Ele é dado através de uma função ou iterador viz() que para cada nó fornece os nós que são seus vizinhos.

Frequentemente os nós representam um conjunto de configurações possíveis. O algoritmo é iterativa e no início de cada iteração temos uma partição dos seus nós em dois conjuntos:

• vistos: nós que já foram encontrados pelo algoritmos

• não_vistos: nós que o algoritmo não tem a menor ideia que existem.

O conjunto de nós vistos é ainda particionado em:

• vistos_não_examinados: nós que foram descobertos, mas que ainda o algoritmo não terminou olhou com cuidado.

• vistos_examinados: nós que foram descobertos e o algoritmo não tem mais nada a fazer com eles.

O conjunto de nós vistos_não_examinados é o coração do algoritmo. Esse conjunto é mantido em uma fila priorizada que é o segredo para o desempenho do algoritmo.

Como tipicamente o número de nós é muito grande, infelizmente, não é possível manter o conjunto de nós examinados explicitamente. Desta forma, a divisão dos nós vistos em vistos_não_examinados e vistos_examinados pode não ser muito séria, pode não ser uma partição. Podemos ter nós vistos que estão em vistos_não_examinados e vistos_examinados o que atrapalha um pouco a metáfora e o consumo de tempo e espaço. . .

6. Referências

Algoritmo A∗ . Ime.usp, 2015. Disponível em: <hhttps://www.ime.usp.br/~coelho/mac0323-2018/aulas/aula09/A-star-rabiscos.pdf>. Acesso em: 30, julho e 2022.

ANTONIO, Marcos. BCC204 - Teoria dos Grafos. decom.ufop, 2019. Disponível em: <http://www.decom.ufop.br/marco/site_media/uploads/bcc204/07_aula_07.pdf>. Acesso em: 31, julho e 2022.

ARRUDA, Heloísa. RESOLUÇÃO DE PROBLEMAS POR BUSCA EM PROFUNDIDADE, **livre saber**. Disponível em: <http://livresaber.sead.ufscar.br:8080/jspui/handle/123456789/81>. Acesso em: 30, julho e 2022.

FEOFILOFF, Paulo .Algoritmo de Prim. **ime.usp**, 2020. Disponível em: <https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/prim.html#sec:algorithm>. Acesso em: 30, julho e 2022.

FEOFILOFF, Paulo .BUSCA EM PROFUNDIDADE. **ime.usp**, 2020. Disponível em: <https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/dfs.html>. Acesso em: 30, julho e 2022.

PICININI, Mirian. UM ESTUDO SOBRE TÉCNICAS DE BUSCA EM GRAFOS E SUAS APLICAÇÕES , **pesc.coppe.ufrj**. Disponível em: <https://www.pesc.coppe.ufrj.br/uploadfile/1368208720.pdf>. Acesso em: 29, julho e 2022.