ESTRUTURA DE DADOS

AULA 5

Prof. Vinicius Pozzobon Borin

CONVERSA INICIAL

O objetivo desta aula é apresentar os conceitos que envolvem a **estrutura de dados do tipo grafo**. Ao longo da aula, conceituaremos essa estrutura de dados, bem como as formas de representação de um grafo, matematicamente e em pseudocódigo. São elas:

- Matriz de adjacências;
- Matriz de incidências;
- Lista de adjacências.

Mostraremos ainda como descobrir um grafo por meio de dois algoritmos distintos:

1. Algoritmo de busca por largura;
2. Algoritmo de busca por profundidade.

Por fim, apresentaremos um algoritmo bastante tradicional para encontrar o caminho mínimo entre dois vértices em um grafo, o **algoritmo de Dijkstra.**

Todos os conceitos trabalhados anteriormente, como análise assintótica, recursividade, listas encadeadas, árvores, bem como conceitos básicos de programação estruturada e manipulação de ponteiros, aparecerão de forma recorrente ao longo de toda esta aula.

Todos os código apresentados aqui estão na forma de pseudocódigos. Para a manipulação de ponteiros e endereços em pseudocódigo, adotamos a seguinte nomenclatura:

- Para indicar o endereço da variável, será adotado o símbolo **(->] antes do nome da variável**. Por exemplo: $px = (->]x$. Isso significa que a variável *px* recebe o endereço da variável *x*;
- Para indicar um ponteiro, será adotado o símbolo **[->) após o nome da variável.** Por exemplo: $x(->]$: inteiro. Isso significa que a variável *x* é uma variável do tipo ponteiro de inteiros.

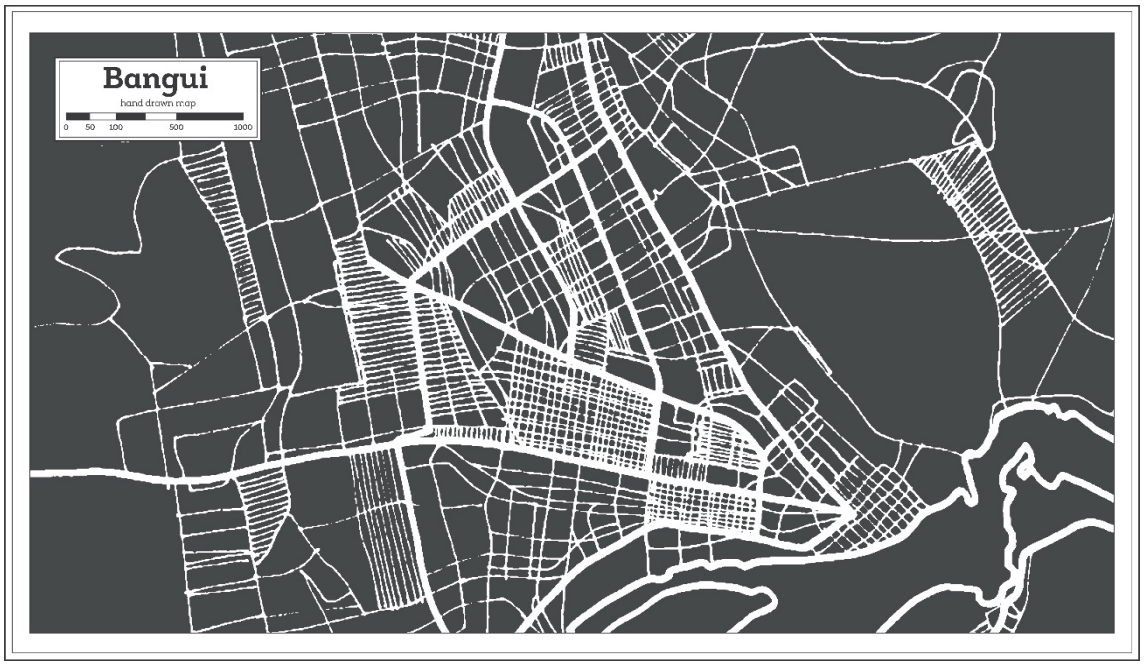
TEMA 1 – GRAFOS: DEFINIÇÕES

Ao longo de nossas aulas, já investigamos estruturas de dados que funcionam de modo linear, como a lista encadeada, e também aquelas que funcionam de modo não linear, na forma de árvores binárias. Porém, uma árvore,

binária ou não, ainda segue uma lógica na construção de sua estrutura por meio da geração de ramos e subdivisões. Nesta aula, investigaremos uma estrutura de dados que ter a sua construção sem um padrão definido e totalmente aleatória: os grafos.

Para entendermos o conceito e a aplicabilidade dos grafos, começaremos com um exemplo prático. Imagine que você foi contratado por uma empresa para mapear ruas, estradas e rodovias de uma cidade. Assim, você recebeu um mapa aéreo da cidade e, por meio de um processamento digital de imagens, extraiu dele todas as ruas e avenidas, chegando a um mapa destacado semelhante ao que vemos na Figura 1.

Figura 1 – Mapa rodoviário de uma cidade hipotética



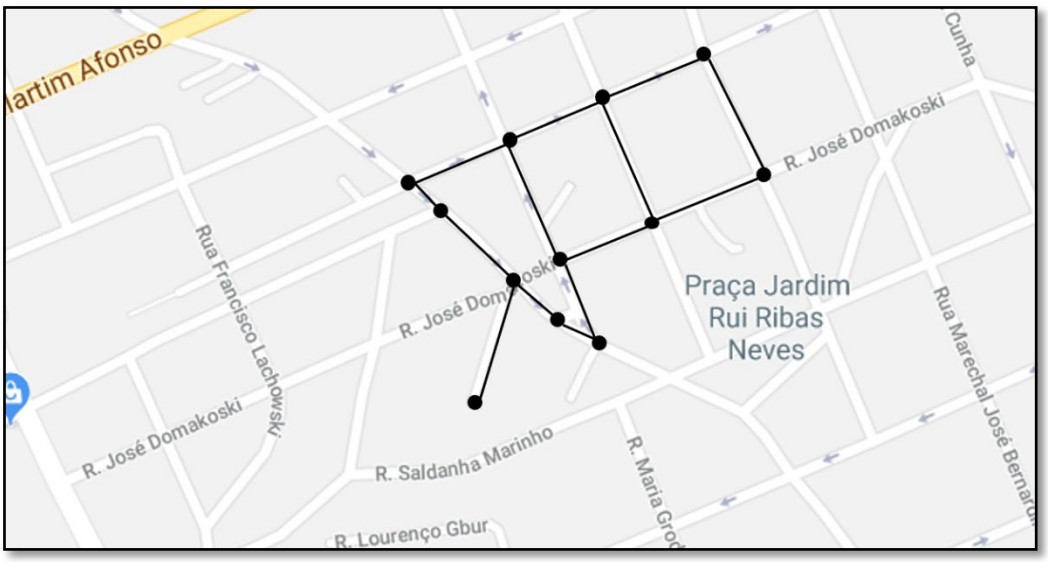
Crédito: Shustriks/Shutterstock.

Com esse mapa processado, você precisa realizar o mapeamento das estradas com o objetivo de desenvolver um *software* que calcule as melhores rotas a partir de um ponto de origem até um destino.

Para realizar esse cálculo de rotas, podemos representar o mapa rodoviário anterior em uma estrutura de dados do tipo grafo. Assim, podemos transformar cada ponto de intersecção entre ruas e avenidas em um **vértice** de um grafo. E cada conexão entre duas intersecções, em uma **aresta** de um grafo. A Figura 2 ilustra um exemplo de mapeamento de uma região da cidade de Curitiba, em que os círculos pretos são os vértices de intersecção e as linhas

pretas, as arestas. Embora na figura somente algumas ruas estejam mapeadas, poderíamos repetir o processo para a cidade inteira. Com o mapeamento pronto, bastaria aplicar um algoritmo para encontrar o menor caminho do grafo, como o de **Dijkstra**, que será apresentado no Tema 5 desta aula, e encontrar as rotas desejadas de um ponto ao outro.

Figura 2 – Mapa rodoviário de uma cidade hipotética com vértices e arestas dos grafos marcados



A aplicabilidade dos grafos se estende a inúmeras áreas da Engenharia e da Tecnologia: inteligência artificial, processamento de imagens, reconhecimento de padrões, telecomunicações, jogos digitais etc.

1.1 Definição de grafos

Um **grafo *G*** é um conjunto de vértices conectados por meio de arestas sem uma distribuição fixa ou padronizada.

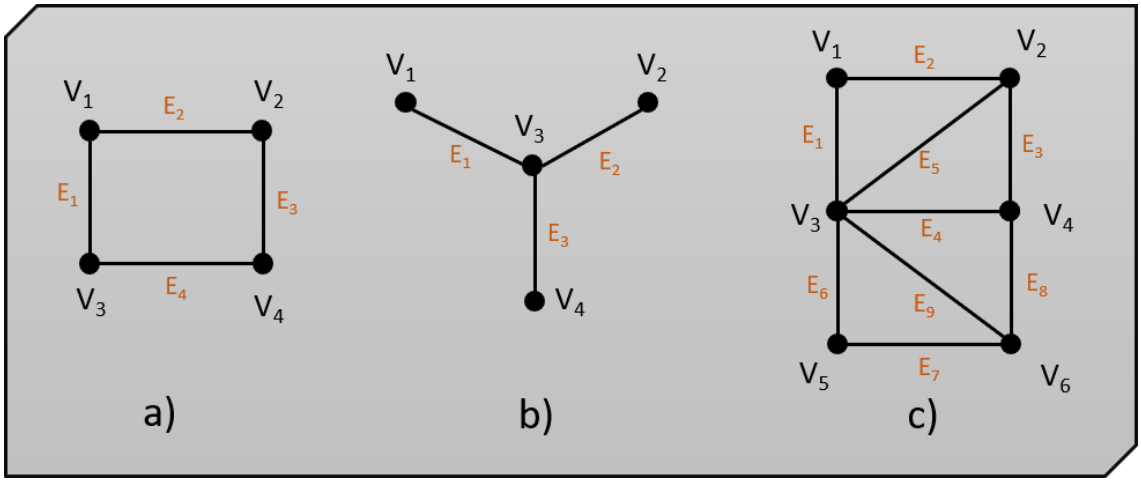
Os **vértices *V*** de um grafo são seus pontos. Cada ponto pode ser um ponto de encontro entre caminhos (rotas) de um grafo, ou, então, o vértice pode conter informações relevantes para o grafo, como dados de informações de cadastros. Tudo depende da aplicação.

As **arestas *E*** são as linhas de conexão entre os vértices. Cada aresta conecta dois vértices. Nem todo vértice precisa ter uma aresta conectada; ele pode permanecer isolado, caso o grafo assim seja construído.

A Figura 3 ilustra três exemplos de grafos. O exemplo da Figura 3(a) contém 4 vértices e 4 arestas. A Figura 3(b) também contém 4 vértices, mas

somente 3 arestas. Na Figura 3(c), perceba que um mesmo vértice chega a ter 5 arestas partindo dele. A quantidade de vértices e arestas em um grafo pode ser ilimitada, não havendo restrições.

Figura 3 – Exemplos de grafos



Para a construção do grafo, não existe uma regra fixa. Qualquer arranjo de vértices e arestas pode ser considerado um grafo. Chamamos de **grafo completo** quando existe uma, e somente uma aresta para cada par distinto de vértices; chamamos de **grafo trivial** aquele que apresenta um único vértice.

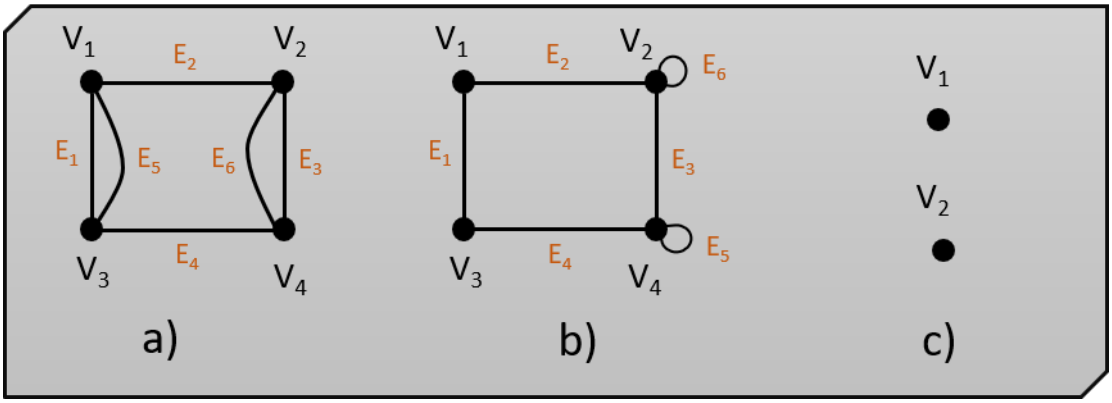
Assim como tínhamos o grau de cada nó de uma árvore (Aula 4), em grafos também podemos encontrar o **grau de cada nó de um vértice**. O grau de um vértice nada mais é do que a soma de todas as arestas que incidem sobre ele. Por exemplo, na Figura 3(a), todos os vértices têm grau 2, enquanto que na Figura 3(c), temos vértices de grau 2 e outros chegando a grau 5, como o vértice 3.

Dentre as particularidades da construção de grafos, podemos citar as **arestas múltiplas**. Arestas múltiplas são aquelas que estão conectadas aos mesmos vértices. Na Figura 4(a), temos essa situação. Os vértices 1 e 3 estão conectados por duas arestas (arestas 1 e 5).

Na Figura 4(b), temos um vértice com **laços**. Um laço acontece quando uma aresta contém somente um vértice ao qual se conectar, iniciando e terminando nele. Vemos isso nas arestas 5 e 6.

Por fim, vemos um grafo desconexo (Figura 4(c)). Nesse tipo de grafo, temos pelo menos um vértice sem nenhuma aresta. No exemplo da figura, ambos estão sem arestas (lembre-se: isso não é obrigatório!).

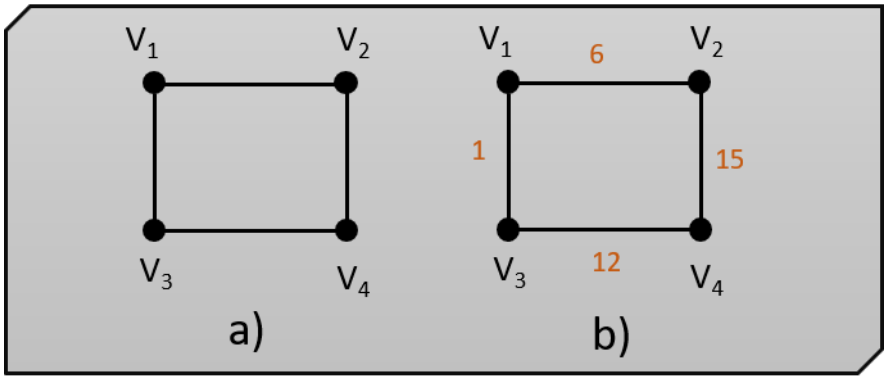
Figura 4 – Peculiaridades na construção de grafos



Para finalizar a abordagem da teoria, precisamos entender que podemos atribuir pesos a nossas arestas. O peso da aresta representa um custo atrelado àquele caminho. Voltando ao nosso exemplo de mapeamento de estradas: caso queiramos traçar uma rota de um ponto A até um ponto B da cidade mapeada, os pesos nas arestas podem ser calculados a partir do tráfego de veículos da rua, por exemplo. Quanto mais veículos, maior o peso da aresta e pior o desempenho dela na escolha da rota.

Quando não indicamos nenhum peso nas arestas, assumimos que elas têm todas o mesmo valor (Figura 5(a)). Quando damos um número para cada aresta, indicamos os valores no próprio desenho do grafo (Figura 5(b)). Chamamos um grafo com pesos em arestas de **grafo ponderado**.

Figura 5 – Peculiaridades na construção de grafos



Podemos representar os grafos matematicamente ou graficamente. As representações matemáticas acabam por ser ideais para a implementação em um algoritmo, por exemplo, já que podem ser manipuladas da forma que o desenvolvedor necessitar.

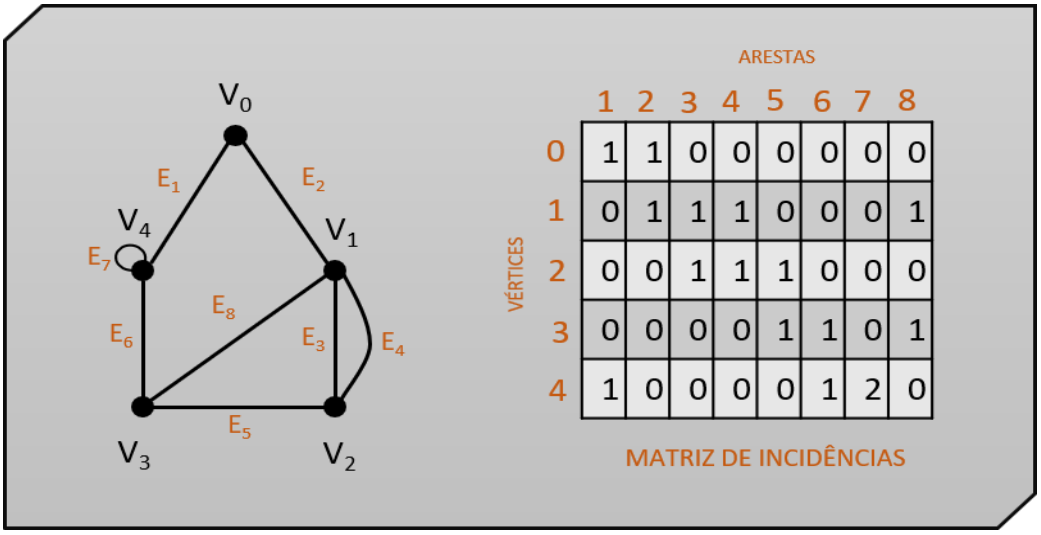
Apresentaremos três maneiras distintas e bastante comuns de representação. Todas elas podem ser implementadas em programação empregando estruturas de dados mais simples e já foram introduzidas anteriormente em nossas aulas, como vetores, matrizes ou, ainda, listas encadeadas (Aula 3).

2.1 Matriz de incidências

A representação de um grafo por uma matriz de incidências consiste em criar uma matriz de dimensão *VxE*, em que *V* é o número de vértices do grafo e *E*, o número de arestas. Por exemplo, se o grafo contém 10 vértices e 5 arestas, temos uma matriz 10x5.

Observe o exemplo da Figura 6. Nessa representação, temos um exemplo com 5 vértices. Utilizaremos o primeiro vértice V_0, pois em programação o primeiro índice de nosso vetor, ou matriz, é o zero.

Figura 6 – Exemplo de matriz de incidências



No exemplo, temos um grafo desenhado com 5 vértices e 8 arestas. Isso resulta em uma matriz de dimensão 5x8. Precisamos, portanto, preencher essa matriz para indicar o grafo construído.

Ao analisar o grafo desenhado, devemos observar cada uma de suas arestas (colunas da matriz). Elas devem ser preenchidas segundo as possibilidades a seguir:

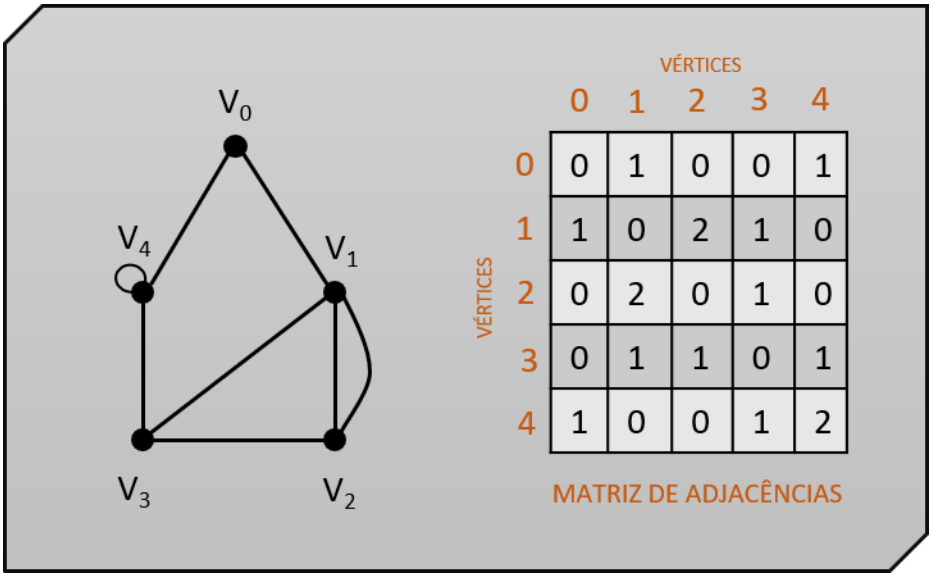
$$\begin{cases} 0,\ caso\ o\ vértice\ (linha\ da\ matriz)\ não\ faça\ parte\ da\ ligação. \\ 1,\ caso\ o\ vértice\ (linha\ da\ matriz)\ faça\ parte\ da\ ligação. \\ 2,\ caso\ aquela\ aresta\ seja\ do\ tipo\ laço. \end{cases}$$

Tome como exemplo a aresta 1, E_1, na representação gráfica. Observe que há uma ligação entre o vértice 0 e o vértice 4. Portanto, as linhas desses vértices na matriz recebem os valores 1, enquanto todas as outras recebem valor zero. O vértice 7, como é um laço, contém o valor 2 nele mesmo.

2.2 Matriz de adjacências

A representação de um grafo por uma matriz de adjacências consiste em criar uma matriz quadrada de dimensão *V*, em que *V* é o número de vértices do grafo. Por exemplo, se o grafo contém 10 vértices, teremos uma matriz 10x10, não importando o número de arestas. A Figura 7 ilustra um exemplo de matriz de adjacências.

Figura 7 – Exemplo de matriz de adjacências



Assim como na representação anterior, povoamos nossa matriz com valores 0, 1 ou 2. A análise que fazemos para preenchimento da matriz é efetuada de uma forma diferente agora. Observamos cada uma das linhas dos vértices, e preenchemos na matriz:

$$\begin{cases} 0, \textit{ caso o outro vértice não tenha conexão com o vértice analisado.} \\ 1, \textit{ caso o outro vértice tenha conexão com o vértice analisado.} \\ 2, \textit{ caso o outro vértice tenha conexão com o vértice analisado e seja um laço.} \end{cases}$$

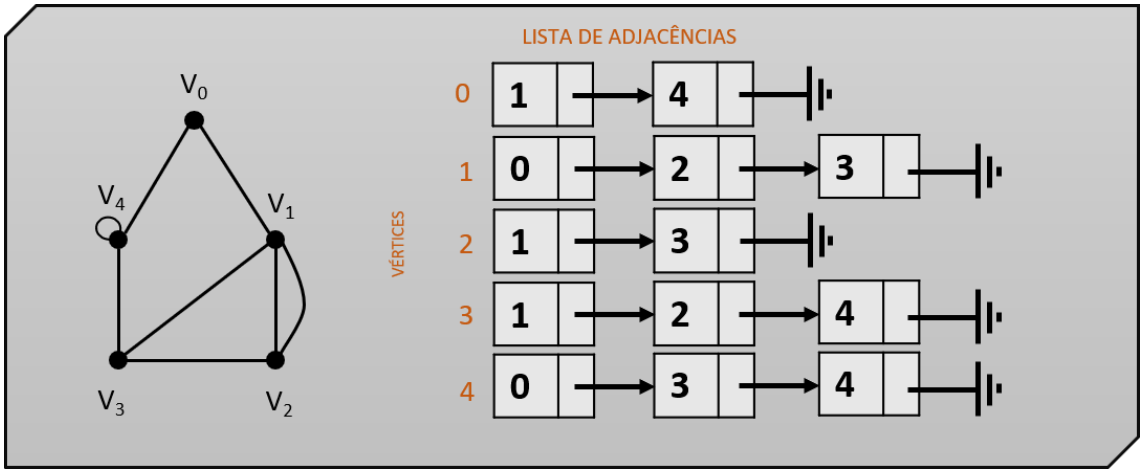
2.3 Lista de adjacências

A representação de um grafo por uma lista de adjacências é muito adotada na área de programação, pois trabalha com listas encadeadas e manipulação de ponteiros de dados.

A Figura 8 contém um exemplo de lista de adjacências. A proposta dessa representação é criar um vetor (ou uma lista encadeada) do tamanho da quantidade de vértices existentes no grafo. Em cada posição desse vetor, teremos uma lista encadeada contendo os endereços dos vizinhos de cada vértice. Conceitualmente, vizinhos de um vértice são todos os vértices que se conectam diretamente a ele por meio de uma aresta.

Assim, teremos uma lista encadeada de vizinhos para cada posição do vetor de vértices criado. Na lista, cada vizinho apontará para outro vizinho daquele mesmo nó.

Figura 8 – Exemplo de lista de adjacências



No exemplo da Figura 8, temos 5 vértices e, portanto, podemos ter um vetor de dimensão 5 (ou uma lista encadeada com 5 nós). Cada posição do vetor terá o endereço do *head* (também vizinho) para uma lista encadeada de vizinhos daquele vértice.

Observe o vértice zero V_0. O vértice zero está na primeira posição do vetor (posição zero). Ele contém, como vizinhos, o vértice 1 e o vértice 4. Assim, na posição zero do vetor, temos o endereço para um dos vizinhos de V_0. Esse

vizinho será o *head* de uma lista encadeada. O vizinho escolhido para ser o *head* foi o vértice 1. Assim, V_1 apontará para o outro vizinho, V_4. Teremos uma lista encadeada de vizinhos do vértice V_0. De forma análoga, criamos uma lista encadeada de vizinhos para cada vértice existente no grafo.

Precisamos investigar o algoritmo de criação dessa estrutura de dados do tipo lista de adjacências. Podemos criar essa estrutura de duas formas distintas. Na primeira, mostrada na Figura 9, declaramos um vetor de dimensão igual ao número de vértices de nosso grafo. Esse vetor será do tipo ponteiros de registros. Assim, cada posição do vetor poderá conter uma estrutura do tipo lista encadeada, armazenando o primeiro elemento dessa lista (*head*). A Figura 9 ilustra o pseudocódigo de lista de adjacências.

Figura 9 – Pseudocódigo de lista de adjacências

```
1  //Vetor do tamanho do número de vértices do grafo
2  //O vetor conterá o endereço o primeiro vizinho (variável ponteiro)
3  Vertices[NUMERO_DE_VERTICES]: ListaDeVizinhos[->)
4
5  //Lista encadeada de vizinhos de cada vértice
6  registro ListaDeVizinhos
7     prox: ListaDeVizinhos[->)
8  fimregistro
```

De forma alternativa, também podemos substituir o vetor por uma estrutura do tipo lista. Assim, teremos duas listas encadeadas, uma chamada de vertical, contendo os vértices e os endereços para os primeiros vizinhos, e uma lista horizontal, contendo as listas de vizinhos de cada vértice. A Figura 10 representa esse caso.

Figura 10 – Pseudocódigo de lista de adjacências

```
1  //Lista encadeada com os vértices e o primeiro vizinho
2  registro Vertices
3     numero: inteiro
4     prox: Vertices[->)
5  fimregistro
6
7  //Lista encadeada de vizinhos de cada vértice
8  registro ListaDeVizinhos
9     prox: Vertices[->)
10 fimregistro
```

Em ambas as situações, teremos um total de listas encadeadas igual ao número de vértices do grafo. Portanto, se houver um grafo com 10 vértices, teremos 10 listas encadeadas, e todas armazenam os vizinhos de cada vértice.

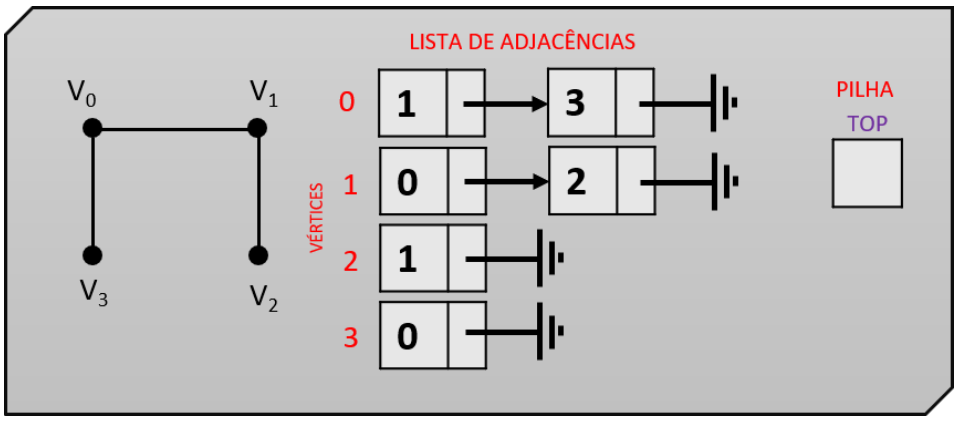
TEMA 3 – ALGORITMO DE BUSCA EM PROFUNDIDADE NO GRAFO

Após entender como construir um grafo, analisaremos como andamos pelos elementos dessa estrutura de dados. Então, imagine que temos um grafo já construído utilizando uma lista de adjacências.

Agora, queremos andar por cada vértice do grafo, sem repetir nenhum deles, partindo de um vértice de origem. Cada vértice visitado é marcado para que não seja revisitado. O algoritmo é encerrado quando o último vértice é visitado.

O algoritmo de **busca em profundidade**, também conhecido como Depth-First Search (DFS), apresenta uma lógica intuitiva e funcional para resolver esse problema de descoberta do grafo. Acompanhe o funcionamento desse algoritmo com um exemplo. O grafo que utilizaremos está na Figura 11.

Figura 11 – Busca em profundidade no grafo: estado inicial



A proposta desse algoritmo é partir de um vértice de origem e acessar a lista de adjacências desse vértice, ou seja, seus vizinhos. O primeiro vizinho detectado (mais adiante na lista encadeada do vértice de origem), será imediatamente acessado.

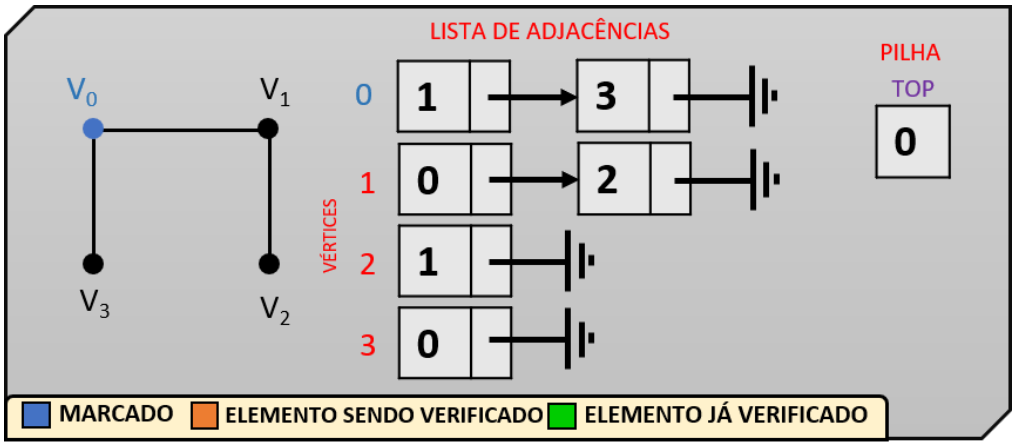
Assim, cada novo elemento ainda não descoberto é selecionado a partir das listas encadeadas de cada vértice, até que o último seja encontrado. Os elementos descobertos vão sendo empilhados e desempilhados segundo as regras de uma estrutura do tipo pilha. Ou seja, só podemos voltar ao vizinho de

um nó mais abaixo na pilha se resolvermos primeiro o elemento mais acima, no topo da pilha.

Analise o grafo da Figura 11. Com ele, mostraremos como se dá o funcionando lógico desse algoritmo e apresentaremos graficamente a descoberta do grafo. O grafo contém 4 vértices, iniciando em zero, e 3 arestas. Podemos construir a lista de adjacências desse grafo representada na Figura 11. Observe ainda que temos uma pilha que, neste momento, está vazia.

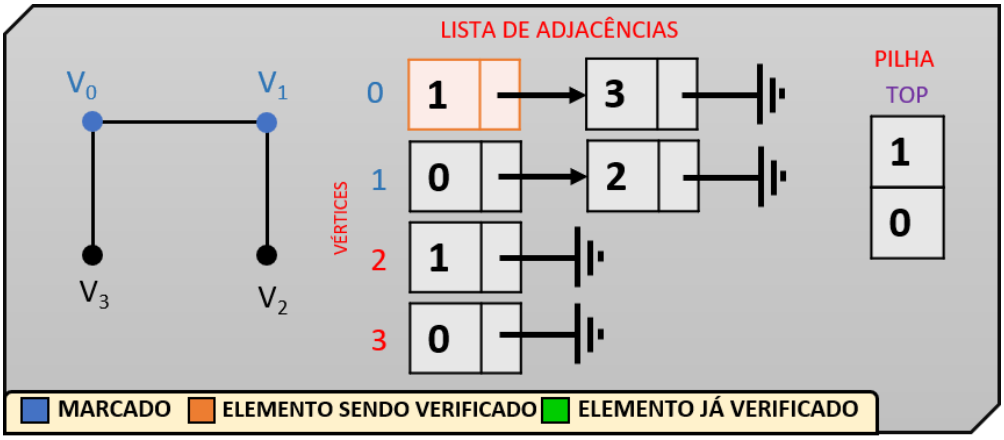
Desejamos iniciar a descoberta de nosso grafo pelo vértice zero. Na Figura 12, vemos que esse vértice é então marcado como já visitado (cor azul). Assim, não é possível revisitá-lo. Em nossa pilha, que estava vazia, colocamos o nosso vértice inicial zero.

Figura 12 – Busca em profundidade no grafo: partindo de V_0: etapa 1



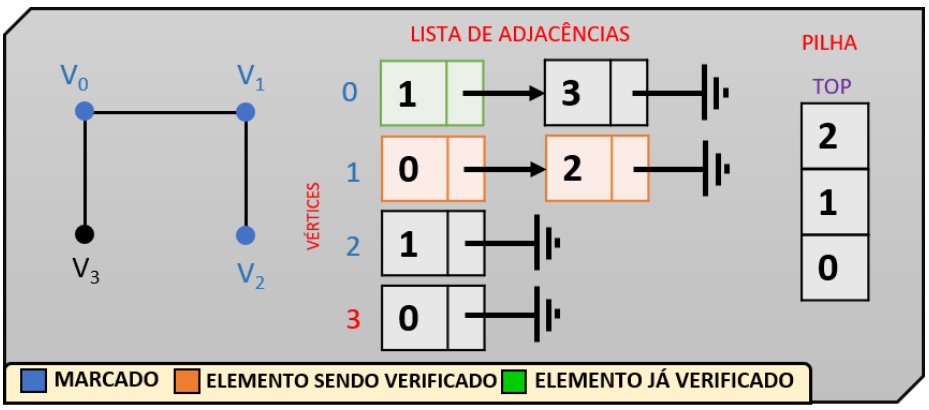
Como o vértice V_0 está no topo da pilha, acessamos sua respectiva lista de vizinhos (Figura 13). Acessamos, então, o primeiro elemento da lista encadeada de vizinhos de V_0, que é o vértice V_1 (destacado em laranja). Imediatamente, colocamos esse vértice como já visitado (cor azul) no grafo e o inserimos no topo da pilha, acima de V_0. Agora, temos dois dos quatro vértices já visitados.

Figura 13 – Busca em profundidade no grafo: partindo de V_0: etapa 2



O próximo vértice a ser visitado é o primeiro vizinho da lista encadeada do vértice V_1. Perceba que nem terminamos de varrer toda a lista encadeada de V_0 e já pulamos para a lista de outro vértice. O primeiro vizinho de V_1 é o próprio V_0, que, na verdade, já foi visitado. Nesse caso, passamos para o próximo elemento da lista encadeada, que é o vértice V_2. Ele ainda não foi visitado (em laranja), e então é marcado como visitado e colocado no topo da pilha, que agora contém três elementos.

Figura 14 – Busca em profundidade no grafo: partindo de V_0: etapa 3

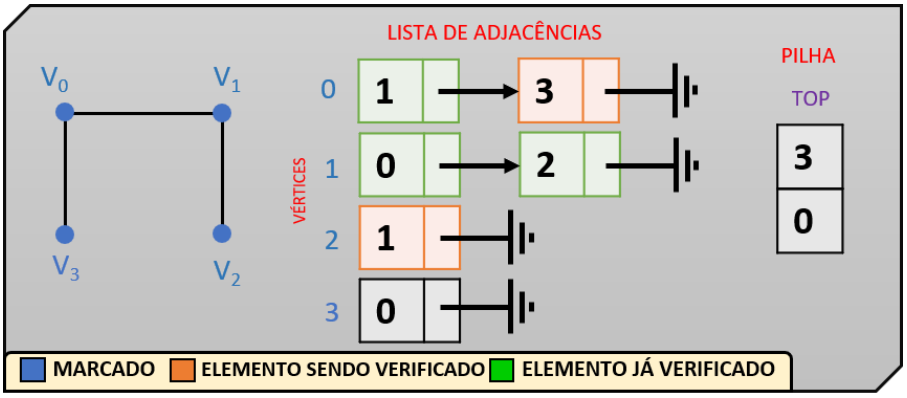


Nesse momento, estamos na lista encadeada do vértice V_2. Na Figura 15, vemos que precisamos acessar o primeiro vizinho do vértice V_2 (em laranja), que é o vértice V_1, que já foi visitado. Toda a lista de vizinhos do vértice V_2 já foi verificado, não restando nenhum.

Desempilhamos o vértice atual (V_2) e voltamos para o elemento abaixo dele na pilha (V_1), que já teve todos os seus vizinhos visitados. Desempilhamos V_1 e voltamos para V_0, acessando o segundo elemento da lista encadeada, pois

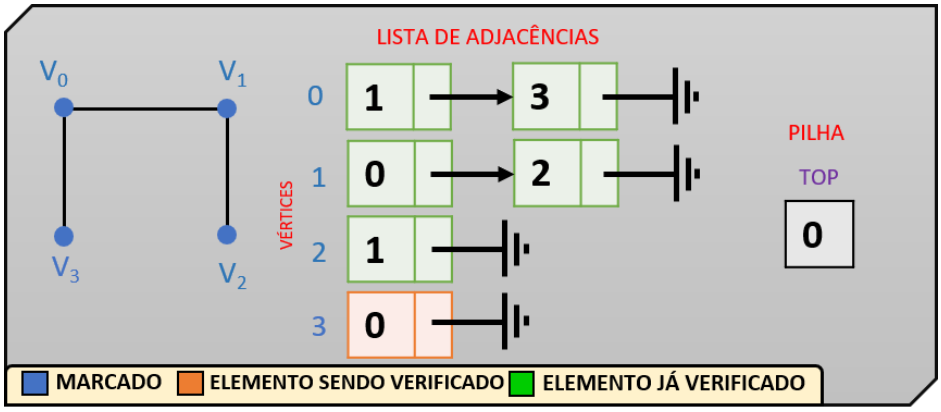
o primeiro já foi visitado anteriormente. Encontramos assim o último vértice não visitado: o V_3. Devemos marcá-lo e colocá-lo no topo da pilha.

Figura 15 – Busca em profundidade no grafo: partindo de V_0: etapa 4



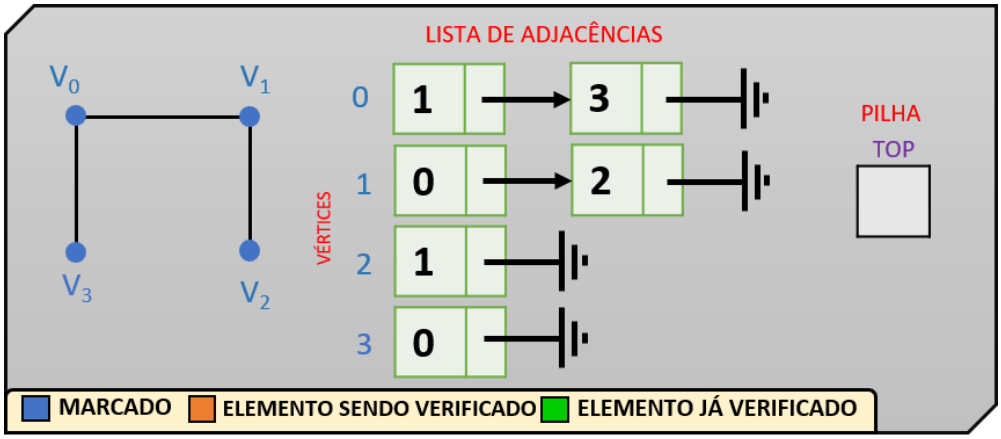
Todos os vértices já foram visitados na etapa anterior. Porém, ainda precisamos encerrar o nosso algoritmo. Na Figura 16, verificamos o primeiro vizinho do vértice V_3 na lista encadeada, o vértice V_0, anteriormente marcado. Chegamos ao final da lista, desempilhamos V_3, e resta somente V_0.

Figura 16 – Busca em profundidade no grafo: partindo de V_0: etapa 5



Embora V_0 já tenha sido visitado, faltava desempilhá-lo (Figura 18). Fazendo isso, temos nossa pilha vazia e os quatro vértices visitados, encerrando nosso algoritmo de busca por profundidade.

Figura 17 – Busca em profundidade no grafo: partindo de V_0: etapa 6



3.1 Pseudocódigo da Depth-First Search (DFS)

Apresentado o funcionamento do algoritmo de busca por profundidade, precisamos conhecer o seu pseudocódigo. Na Figura 18, temos a função da Depth-First Search (DFS). Observe que essa função faz parte de um algoritmo maior que realizará a chamada de si mesma.

Portanto, imagine uma estrutura de grafo criada por meio de uma lista de adjacências, como vemos na Figura 10. A Figura 18 utiliza esse grafo pronto para realizar a busca por profundidade.

Figura 18 – Pseudocódigo da função de busca em profundidade no grafo

```
1  função BuscaProfundidade (Vizinhos: ListaDeVizinhos, tam: inteiro,
2   v: inteiro, marcado[NUMERO_VERTICES]: inteiro, Top: Pilha[->])
3   var
4         vert: vertice[->) //Variável de varredura
5         w, i: inteiro
6   inicio
7      marcado[v] = 1 //Marca o vértice como visitado
8      Empilhar(Top, v) //Inserindo o vértice na pilha
9      para i de 1 até tam faça
10        //Localiza os vizinhos do vértice 'v', iniciando no 1º
11        vert = Vizinhos[v].Vertices
12        enquanto (vert <> NULO) faça
13           w = vert->numero
14           se (marcado[w] <> 1) então
15              //Acessa recursivamente o próximo vértice não visitado
16               BuscaProfundidade(Vizinhos, tam, w, marcado, Top)
17           fimse
18           vert = vert->prox  //Próximo vizinho de 'v'
19        fimenquanto
20     fimpara
21     Desempilhar(Top) //Remove o vértice na pilha
22  fimfunção
```

O pseudocódigo está comentado na figura. Porém, explicaremos melhor algumas de suas linhas a seguir:

- Linhas 1 e 2: declaração da função. A função recebe como parâmetro a lista de vizinhos de um nó, o tamanho dessa lista, o vértice em questão para ter sua lista acessada, um vetor contendo todos os vértices já visitados e o topo da pilha para sabermos qual vértice é o próximo a receber tratamento;
- Linhas 8 e 21: ambas as linhas servem para empilhar e desempilhar da pilha. O algoritmo que realiza esse procedimento foi descrito no Tema 4 da Aula 3;
- Linhas 9 a 19: procedimento que faz a busca por profundidade, ou seja, pega o primeiro elemento da lista encadeada e, quando necessário, chama recursivamente outra lista encadeada de um próximo vértice da pilha.

TEMA 4 – ALGORITMO DE BUSCA EM LARGURA NO GRAFO

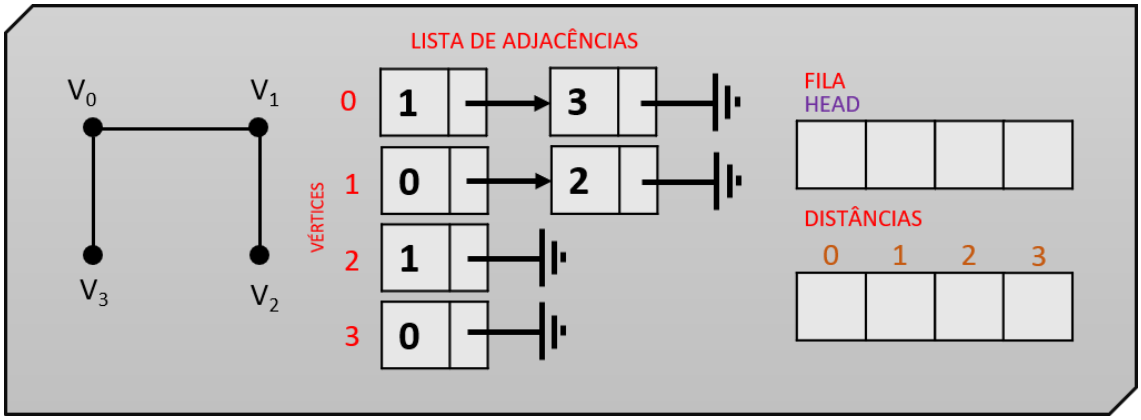
Neste tema, investigaremos outro algoritmo de descoberta de um grafo. A proposta continua a ser a mesma: visitar cada vértice do grafo, sem repetir nenhum deles, partindo de um vértice de origem. Cada vértice visitado é marcado para que não seja revisitado. O algoritmo é encerrado quando o último vértice é visitado.

O algoritmo de **busca em largura**, também conhecido como Breadth-First Search (BFS), trabalha com a ideia de visitar primeiro todos os vizinhos próximos do vértice selecionado antes de pular para o próximo vértice.

Uma diferença interessante em relação à busca por profundidade é que a BFS trabalha com uma estrutura do tipo fila para indicar qual é o próximo vértice a ser acessado, enquanto na Depth-first Search (DFS) tínhamos uma estrutura de pilha.

Acompanhe o funcionamento desse algoritmo com um exemplo. A Figura 19 ilustra o grafo que será trabalhado. Perceba que temos também um vetor de distâncias. Esse vetor manterá armazenada a distância de cada vértice em relação ao vértice de origem, ajudando na decisão de qual será o próximo vértice a ser visitado.

Figura 19 – Busca em largura no grafo: partindo de V_0: estado inicial



A proposta desse algoritmo é partir de um vértice de origem e acessar a lista de adjacências desse vértice, ou seja, seus vizinhos. A partir da lista de vizinhos, devemos calcular a distância deles até o vértice de origem e salvar as distâncias em um vetor de distâncias. Os elementos são enfileirados à medida que são acessados nas listas encadeadas.

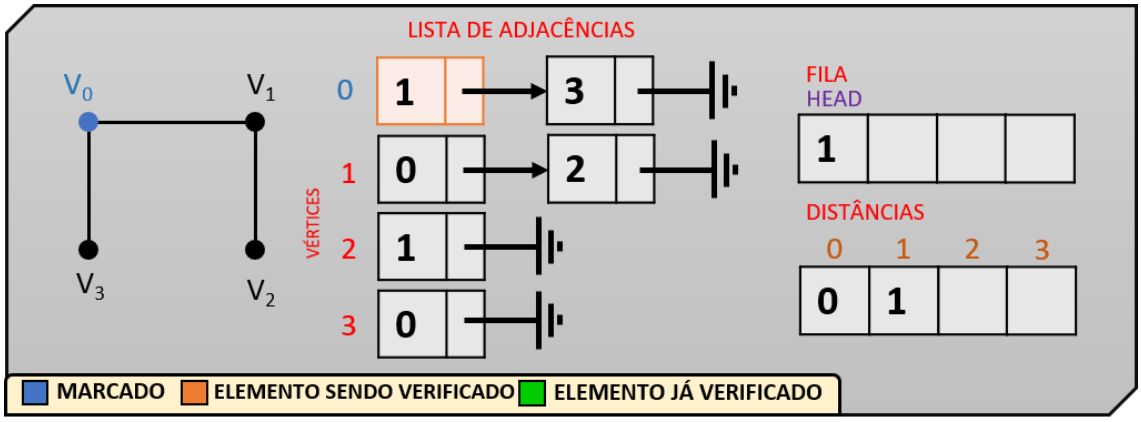
Reiniciamos nossa análise no vértice V_0. Desse modo, podemos imediatamente marcá-lo como já visitado (cor azul). Colocamos também esse vértice na fila. Lembre-se de que, em uma fila (Aula 3, Tema 5), a inserção acontece sempre no final dela (FIFO, do inglês *first in, first out*). Como a fila está vazia, inserimos o vértice na primeira posição. A distância do vértice de origem para ele mesmo será sempre zero (Figura 20).

Figura 20 – Busca em largura no grafo: partindo de V_0: etapa 1

Conhecendo a lista de vizinhos do vértice V_0, devemos passar elemento por elemento dessa lista encadeada, inserindo-os ao final da fila e calculando as distâncias. Na Figura 21, encontramos o primeiro vizinho de V_0, que é V_1. Colocamos V_1 na fila e calculamos a distância dele até a origem. O cálculo da distância é feito com base no valor da distância do vértice atual (valor zero)

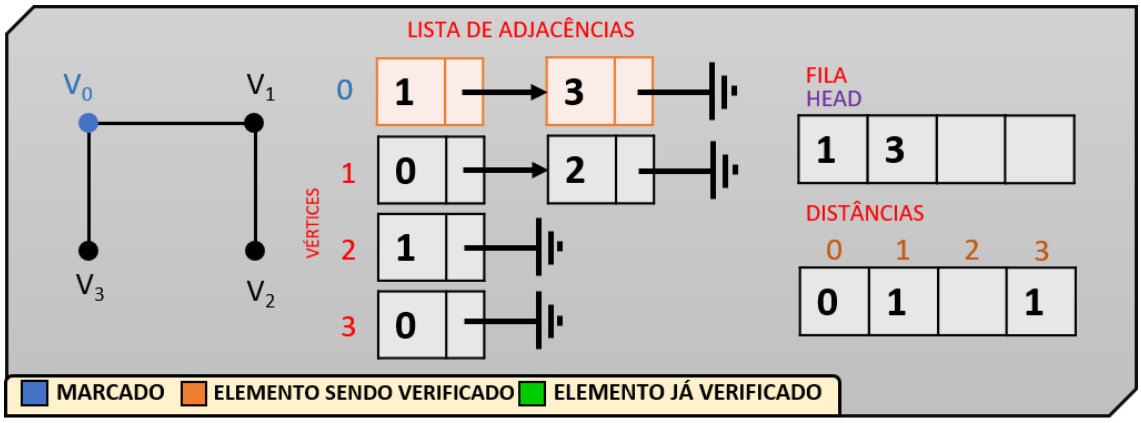
acrescido de uma unidade, resultando em distância 1 $(0 + 1 = 1)$, como vemos na Figura 21.

Figura 21 – Busca em largura no grafo: partindo de V_0: etapa 2



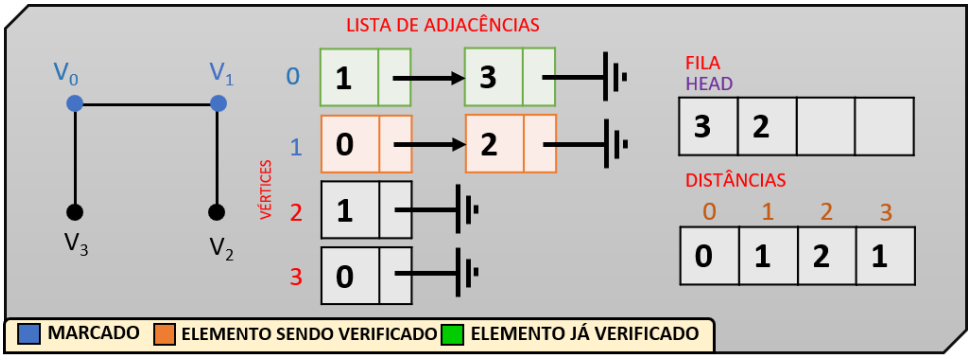
Na Figura 22, passamos para o próximo vizinho do vértice V_0. O vértice V_3 recebe o mesmo tratamento que V_1 e sua distância é de valor 1 $(0 + 1 = 1)$, pois está a um salto de distância da origem e é colocado no final da fila, após o vértice V_1. Assim, encerramos a varredura dos vizinhos do vértice V_0, calculamos as distâncias, mas ainda não os visitamos.

Figura 22 – Busca em largura no grafo: partindo de V_0: etapa 3



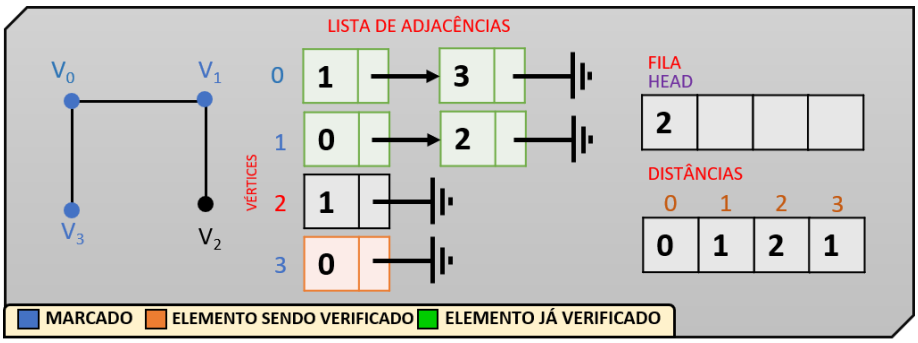
Com V_0 encerrado (cor verde), partimos para o próximo vértice da fila, o vértice V_1. Marcamos esse vértice, o removemos da fila e acessamos sua lista encadeada de vizinhos. O primeiro vizinho é o vértice V_0, que já foi visitado e, portanto, seguimos para o próximo vértice, que é V_2. Esse vértice ainda não foi visitado, então o colocamos ao final da fila e calculamos uma distância. Como ele está a dois vértices de distância da origem, sua distância é 2 $(1 + 1 = 2)$, como mostra a Figura 23.

Figura 23 – Busca em largura no grafo: partindo de V_0: etapa 4



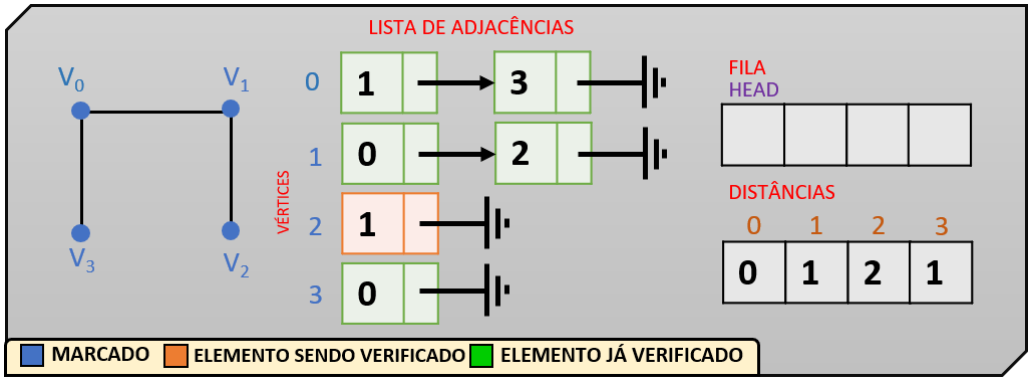
Na etapa 4, já encerramos os cálculos de todas as distâncias possíveis e enfileiramos todos os vértices restantes, V_3 e V_2. Assim, na etapa 5 (Figura 24), removemos da fila o próximo vértice V_3 e o marcamos usando a cor azul. Depois, acessamos sua lista de vizinhos. Nela, só existe o vértice V_0, que já foi acessado. Portanto, nada mais temos a fazer nessa etapa e podemos passar para o próximo elemento de nossa fila.

Figura 24 – Busca em largura no grafo: partindo de V_0: etapa 5



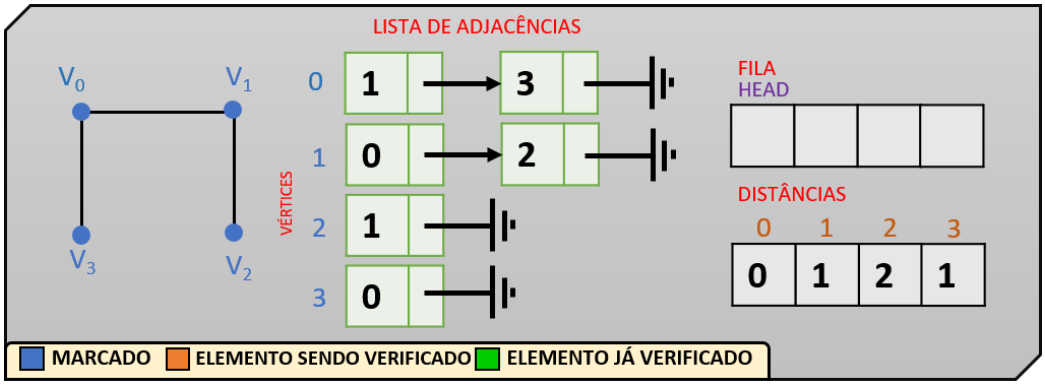
Na Figura 25, acessamos o vértice V_2, último ainda não descoberto, e vemos que ele tem como vizinho somente o vértice V_1. Como o vértice 1 já é conhecido, nada mais temos a fazer nessa etapa.

Figura 25 – Busca em largura no grafo: partindo de V_0: etapa 6



Por fim, na etapa 7 (Figura 26), todos os elementos já foram visitados e todas as listas encadeadas, varridas. Nosso algoritmo se encerra aqui.

Figura 26 – Busca em largura no grafo: partindo de V_0: etapa 7



4.1 Pseudocódigo da Breadth-First Search (BFS)

Apresentado o funcionamento do algoritmo de busca por largura, precisamos conhecer seu pseudocódigo. Na Figura 27, temos a função da Breadth-First Search (BFS). Observe que essa função faz parte de um algoritmo maior que realiza chamadas de si mesma.

Portanto, imagine uma estrutura de grafo criada por meio de uma lista de adjacências, como vemos na Figura 10. A Figura 27 utiliza esse grafo pronto para realizar a busca por largura.

Figura 27 – Pseudocódigo da função de busca em profundidade no grafo

```
1  função BuscaLargura (Vizinhos: ListaDeVizinhos, tam: inteiro, v: inteiro,
2  marcado[NUMERO_VERTICES]: inteiro, distancias[NUMERO_VERTICES]: inteiro,
3  Head: Fila[->))
4  var
5        vert: vertice[->) //Variável de varredura
6        w, i, v_uso: inteiro
7  inicio
8     marcado[v] = 1 //Marca o vértice como visitado
9     distancias[v] = 0 //Distância de 'v' para ele mesmo é zero
10    InserirNaFila(Head, v) //Insere o vértice na fila
11    enquanto (Head <> NULO) faça
12       v_uso = RemoverDaFila(Head) //Remove o vértice da fila
13       para i de 1 até tam faça
14          //Varre a lista de vizinhos do vértice
15          vert = Vizinhos[v_uso].Vertices
16          enquanto (vert <> NULO)
17             w = vert->numero
18             //Se vertice não estiver, marcado calcula a distância
19             //em relação a origem
20             se (marcado[w] == 0) então
21                marcado[w] = 1
22                distancias[w] = distancias[v_uso] + 1
23                //Insere ele na fila para visitar seus vizinhos depois
24                InserirNaFila (Head, w)
25             fimse
26             //Proximo vértice adjacente a 'v'
27             vert = vert->prox
28          fimenquanto
29       fimpara
30    fimenquanto
31 fimfunção
```

O pseudocódigo está comentado na figura. Porém, a seguir, explicamos melhor algumas linhas:

- Linhas 1 e 2: declaração da função. A função recebe como parâmetro a lista de vizinhos de um nó, o tamanho dessa lista, o vértice em questão para ter a sua lista acessada, um vetor contendo todos os vértices já visitados, e outro vetor contendo as distâncias calculadas e o início (*head*) da fila; assim, saberemos qual vértice é o próximo a receber tratamento;
- Linhas 8 e 21: ambas as linhas servem para empilhar e desempilhar. O algoritmo que realiza esse procedimento foi descrito no Tema 4 da Aula 3;
- Linhas 9 a 19: procedimento que faz a busca por profundidade, ou seja, pega o primeiro elemento da lista encadeada e, quando necessário, chama recursivamente outra lista encadeada do vértice seguinte da pilha.

TEMA 5 – ALGORITMO DO CAMINHO MÍNIMO EM GRAFO: DIJKSTRA

Aprendemos no Tema 2 a construir a representação em pseudocódigo de um grafo. Nos temas 3 e 4, aprendemos a descobrir um grafo, passando por cada vértice uma única vez e empregando dois algoritmos distintos. Agora, neste último tema, estudaremos um algoritmo que, a partir de um grafo conhecido, encontra a menor rota (caminho com menor peso) entre dois vértices.

O algoritmo investigado neste tema será o **algoritmo de Dijkstra**, que recebeu esse nome em homenagem ao cientista da computação holandês Edsger Dijkstra, que publicou o algoritmo pela primeira vez em 1959. Esse algoritmo é o mais tradicional para realizar a tarefa de encontrar um caminho. Para realizar tal procedimento, podemos utilizar um **grafo ponderado**, ou seja, aquele com pesos distintos nas arestas.

Aplicaremos o algoritmo de Dijkstra em um grafo ponderado e obteremos as menores rotas, partindo de uma origem, para todos os outros vértices do grafo. Todo esse algoritmo será apresentado utilizando uma **métrica aditiva**. Isso significa que essa métrica encontrará a menor rota considerando o menor peso somado entre os caminhos.

Um exemplo prático de métrica aditiva é o tráfego de veículos em rodovias. Quanto maior o tráfego, pior o desempenho da aresta do grafo.

Existem outros tipos de métricas. Na multiplicativa, por exemplo, o melhor caminho é encontrado calculando-se o produto dos pesos das arestas. O maior valor dentre os produtos é a melhor rota. Um exemplo dessa métrica é o limite de velocidade das estradas: quanto maior o limite, mais rápido o veículo anda, e, portanto, melhor é aquela rota/aresta.

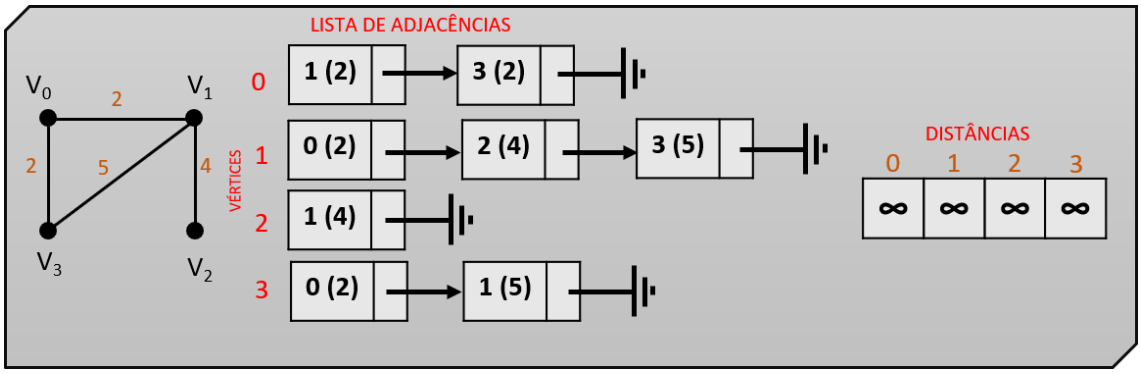
Como trabalharemos com um grafo ponderado, precisamos adaptar nossa lista de adjacências para que o registro também armazene os pesos das arestas. Podemos deixar nossa lista como vemos na Figura 28, alterando o registro de vértices para também conter um campo para pesos.

Figura 28 – Pseudocódigo de lista de adjacências para grafo ponderado

```
1   //Lista encadeada com os vértices e o primeiro vizinho
2   registro Vertices
3      numero: inteiro
4      peso: inteiro
5      prox: Vertices[->)
6   fimregistro
7
8   //Lista encadeada de vizinhos de cada vértice
9   registro ListaDeVizinhos
10     prox: Vertices[->)
11  fimregistro
```

Na Figura 29, vemos o estado inicial da lista de adjacências ponderadas. Os valores que estão entre parênteses são os pesos das arestas. Por exemplo, na lista encadeada do vértice V_1, temos como primeiro elemento V_0, e o peso da aresta entre eles está indicado pelo valor 2 entre parênteses.

Figura 29 – Algoritmo de Dijkstra: partindo de V_1: estado inicial.



O algoritmo do caminho mínimo precisa calcular as menores rotas de um vértice para todos os outros. Portanto, um vetor com distâncias fica armazenado. Nesse vetor, todas as rotas iniciam com um valor infinito, ou seja, como se o caminho de um vértice até o outro não fosse conhecido. À medida que os trajetos vão sendo calculados, os pesos das rotas são alterados.

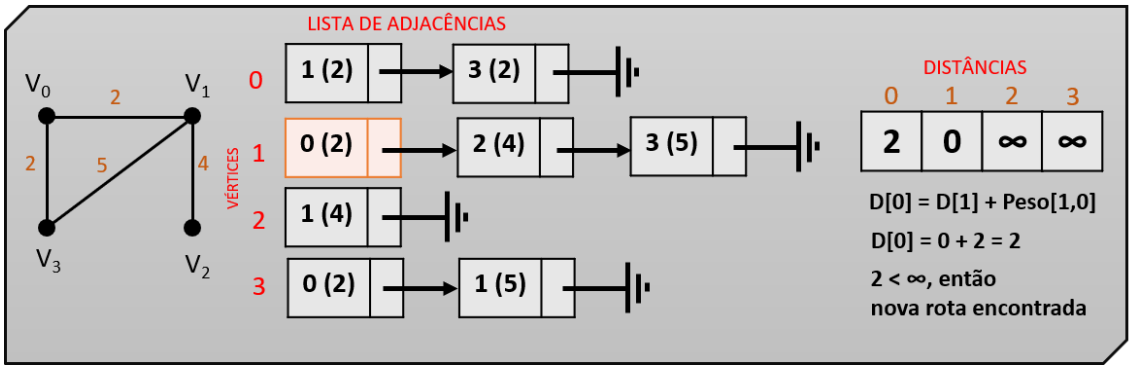
Explicaremos o funcionamento do algoritmo iniciando pelo vértice V_1. Assim, encontraremos a rota de V_1 para todos os outros vértices existentes no grafo ponderado.

Na Figura 30, iniciamos a primeira etapa da operação do método. Como partiremos de V_1, já iniciamos acessando a lista de vizinhos desse vértice e calculando as rotas para eles. Encontramos a rota de V_1 para ele mesmo. Nesse

caso, o vértice de origem para ele mesmo terá sempre peso zero, conforme apresentado no vetor de distâncias.

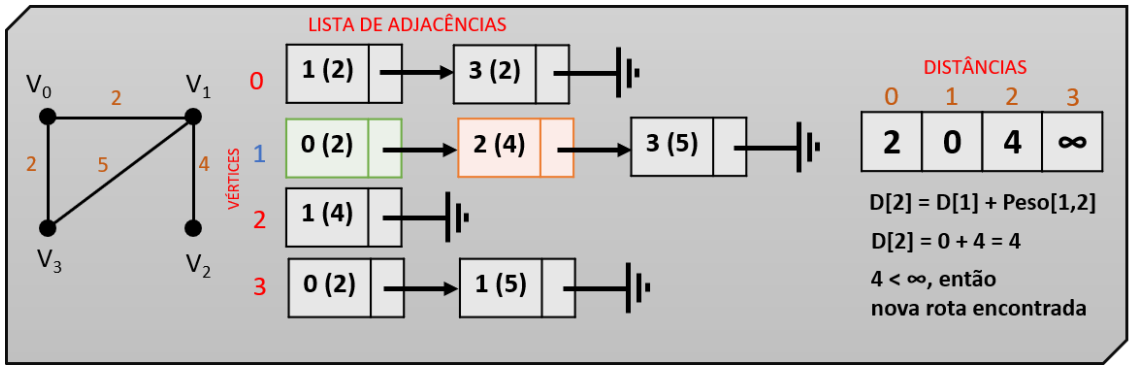
Em seguida, acessamos o *head* da lista encadeada de V_1, que é V_0, com um peso de aresta de 2. Assim, o cálculo da distância entre eles será o peso em $V_1 = 0$ acrescido do peso dessa aresta, resultando no valor 2. Esse valor, por ser menor do que infinito, é colocado no vetor, na posição de V_0. O cálculo é apresentado no canto inferior direito da Figura 30.

Figura 30 – Algoritmo de Dijkstra: partindo de V_1: etapa 1



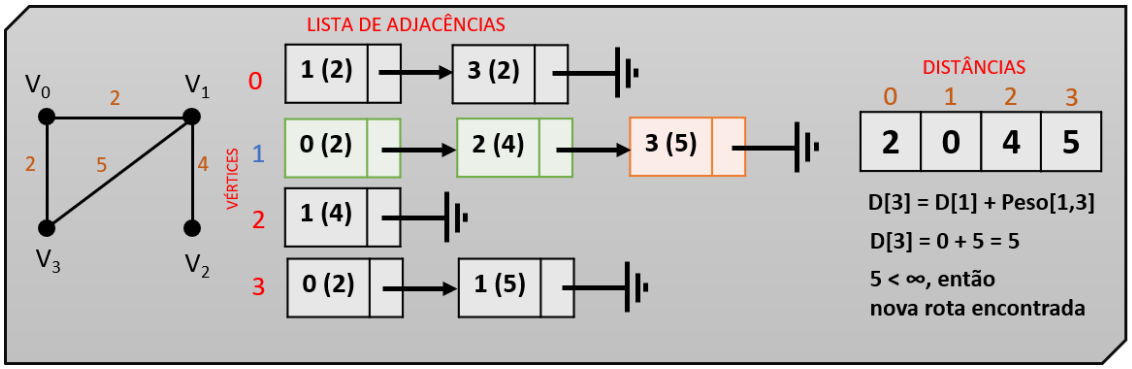
Seguimos na Figura 31 acessando o segundo elemento da lista de vizinhos de V_0. Em V_2, fazemos o peso de $V_0 = 0$, acrescido da aresta para V_2, resultando no valor 4, que por ser menor do que infinito é colocado como rota válida entre V_1 e V_2.

Figura 31 – Algoritmo de Dijkstra: partindo de V_1: etapa 2



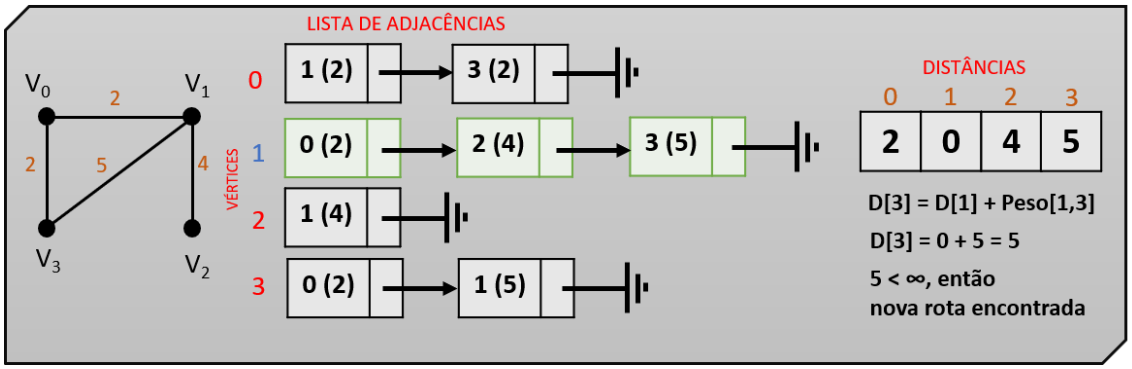
A etapa 3 é mostrada na Figura 31. De forma semelhante às duas etapas anteriores, calculamos a distância entre V_1 e V_3 e obtemos o resultado 5; assim, colocamos o vetor de distâncias na posição de V_3.

Figura 32 – Algoritmo de Dijkstra: partindo de V_1: etapa 3



Todos os vizinhos do vértice V_1 têm suas rotas calculadas. Um panorama geral é apresentado na Figura 33, indicando que todos os vizinhos foram calculados (cor verde).

Figura 33 – Algoritmo de Dijkstra: partindo de V_1: etapa 4

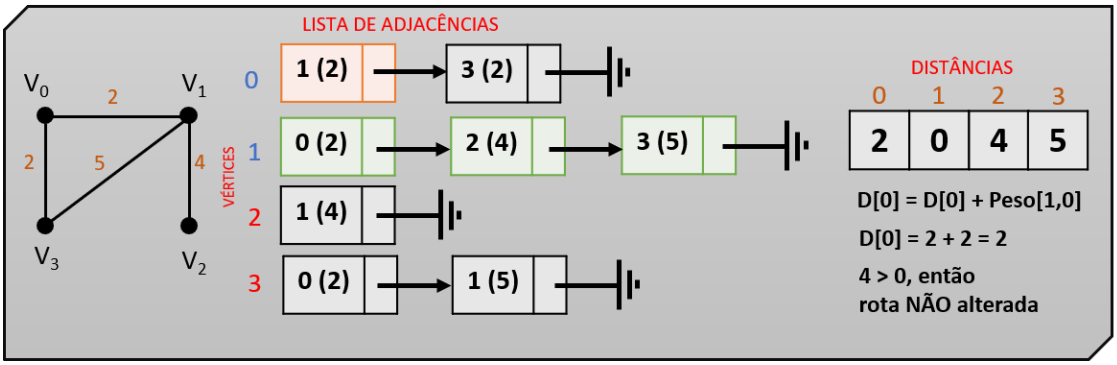


Precisamos passar para o próximo vértice e calcular as rotas partindo de V_1, passando por esse vértice intermediário. O vértice que calcularemos agora é o de menor caminho no vetor distâncias e que ainda não tenha sido marcado. Será, portanto, o vértice V_0, com distância 2 para V_1.

Na Figura 34, temos V_0 marcado e seu primeiro vizinho, acessado. Seu primeiro vizinho é o próprio vértice de origem V_0. Isso significa que precisamos calcular o peso da rota que inicia em V_0, passa para V_1 e retorna para V_0. O peso dessa rota será o peso já calculado até $V_0 = 2$, acrescido do peso da aresta de V_0 para V_1 $(2 + 2 = 4)$.

Fazer um trajeto partindo da origem, passando por outro vértice e depois retornando para a origem sempre resultará em um peso na rota superior, se comparado com o peso da origem individualmente $(V_1 = 0)$. Portanto, essa nova rota não substitui aquela já existente, como vemos na Figura 34.

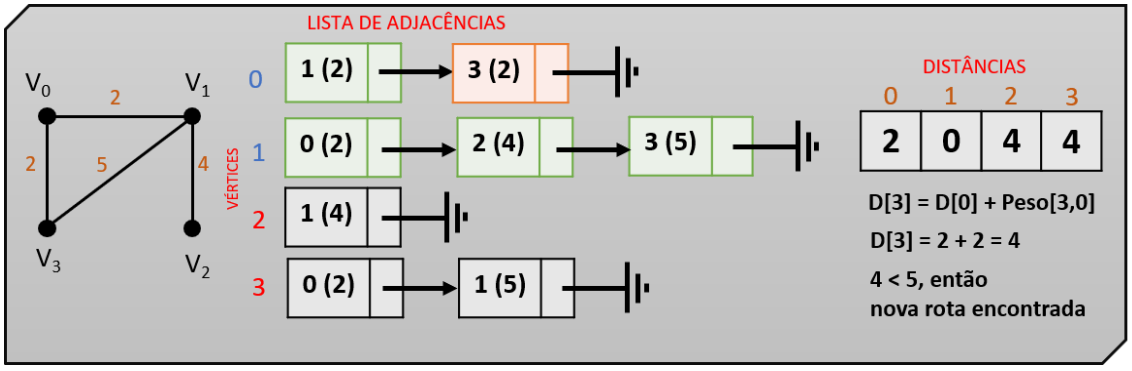
Figura 34 – Algoritmo de Dijkstra: partindo de V_1: etapa 5



Na Figura 35, seguimos para o próximo vizinho de V_0, o V_3. Isso significa que calcularemos uma rota $V_1 \rightarrow V_0 \rightarrow V_3$. O peso dessa rota será o peso até V_0 já calculado e de valor 2, somado ao peso da aresta V_0 e V_3, que também é 2. Assim, $2 + 2 = 4$.

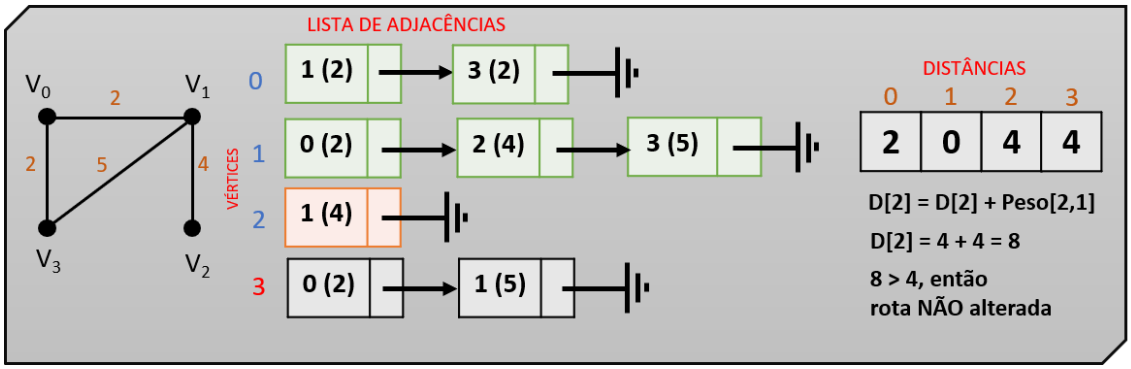
Observe agora algo interessante: o peso da rota de V_1 até V_3, passando por V_0, resultou em peso 4. Anteriormente, calculamos o peso da rota direta entre V_1 e V_3, cujo valor resultou em 5. Portanto, a rota que passa por V_0 tem um peso menor ($4 < 5$), resultando em uma nova rota até V_3. Note que uma rota com peso menor, mesmo passando por um vértice a mais, acaba resultando em um caminho menor que a outra.

Figura 35 – Algoritmo de Dijkstra: partindo de V_1: etapa 6



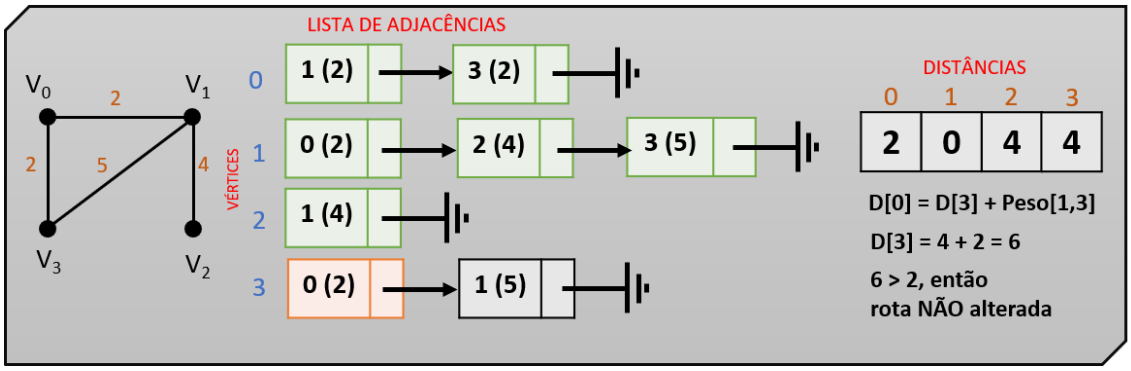
Na Figura 36, já encerramos nossos cálculos passando pelo vértice V_0. Seguimos para o próximo vértice não visitado e de menor distância no vetor, o vértice V_2. O único vizinho de V_2 é o vértice origem V_1. Fazer o trajeto $V_1 \rightarrow V_2 \rightarrow V_1$ tem um custo atrelado inferior ao custo zero de permanecer em V_1. Portanto, essa rota não é válida.

Figura 36 – Algoritmo de Dijkstra: partindo de V_1: etapa 7



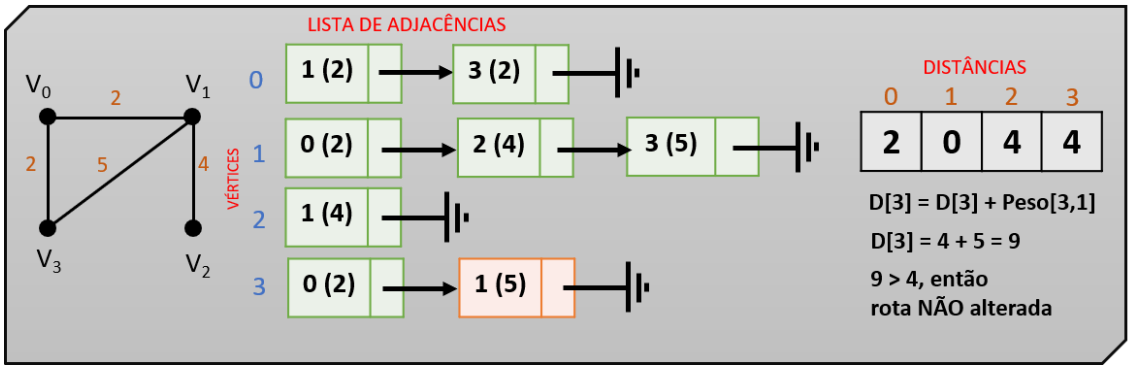
Na Figura 37, seguimos para o próximo, e último, vértice não visitado e de menor distância no vetor: o vértice V_3. O primeiro vizinho de V_3 é V_0. Assim, faremos o trajeto $V_1 \rightarrow V_3 \rightarrow V_0$. O custo até V_3 é 4 e o peso de V_3 até V_0 é 2, resultando em 6 (4 + 2), custo superior ao da rota atual, que é 4.

Figura 37 – Algoritmo de Dijkstra: partindo de V_1: etapa 8



Na Figura 38, temos o segundo vizinho de V_3, o V_1. Assim, faremos o trajeto $V_1 \rightarrow V_3 \rightarrow V_1$. O custo até V_3 é 4 e o peso de V_3 até V_1 é 5, resultando em 9 (4 + 5), custo superior ao da rota atual que é 0.

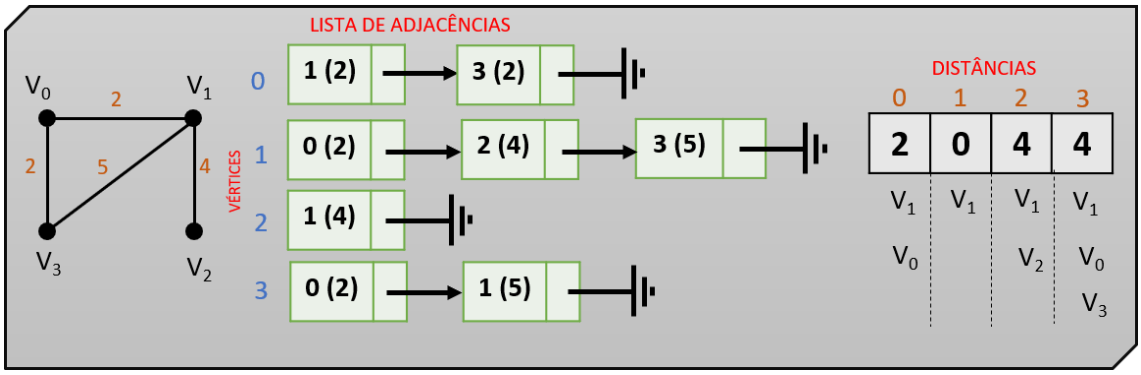
Figura 38 – Algoritmo de Dijkstra: partindo de V_1: etapa 9



Na Figura 39, já percorremos todos os vizinhos de todos os vértices e calculamos todas as rotas possíveis. Desse modo, as distâncias resultantes

estão no vetor de distâncias, e abaixo de cada valor está a sequência de vértices para aquela rota. Por exemplo, para atingirmos o vértice V_3, a melhor rota encontrada foi $V_1 \rightarrow V_0 \rightarrow V_3$, e não $V_1 \rightarrow V_3$ diretamente.

Figura 39 – Algoritmo de Dijkstra: partindo de V_1: etapa 10



5.1 Pseudocódigo de Dijkstra

Apresentado o funcionamento do algoritmo de caminho mínimo Dijkstra, precisamos conhecer seu pseudocódigo. Na Figura 40, temos a função de Dijkstra. Observe que essa função faz parte de um algoritmo maior que realiza chamadas de si mesma.

Portanto, imagine uma estrutura de grafo criada por meio de uma lista de adjacências ponderada, como vemos na Figura 28. A Figura 40 usufrui desse grafo pronto para realizar os cálculos de rotas.

O pseudocódigo está comentado na figura. Porém, explicaremos melhor algumas linhas a seguir:

- Linha 1: declaração da função. O algoritmo recebe como parâmetro o vértice de origem que terá todas as rotas calculadas a partir dele. E ele terá peso de rota igual a zero;
- Linhas 10 a 14: inicializa o vetor de distâncias com infinito, ou um valor suficientemente alto e que seja substituído por qualquer rota válida no grafo;
- Linha 18: laço de repetição que executará para o total de vértices do grafo;
- Linhas 21 a 25: encontra o próximo vértice com menor distância para ter suas rotas calculadas;
- Linhas 29 a 37: calcula todas as rotas/distâncias a partir do vértice atualmente marcado;

Figura 40 – Pseudocódigo do algoritmo de caminho mínimo

```
1   função Dijkstra (Origem: inteiro)
2   var
3      distancias[NUMERO_VERTICES]: inteiro
4      predecessor[NUMERO_VERTICES]: inteiro
5      visitado[NUMERO_VERTICES]: inteiro
6      cont: inteiro //Numero de nós já visitados
7      D_min, prox, i, j: inteiro
8      vert: vertice[->)
9   inicio
10     para i de 0 até (NUMERO_VERTICES - 1) faça
11        distancias[i] = INFINITO
12        predecessor[i] = Origem
13        visitado[i] = 0
14     fimpara
15     distancias[Origem] = 0
16     cont = 0
17
18     enquanto (cont < NUMERO_VERTICES - 1) faça
19        D_min = INFINITO
20        //Próximo vértice é aquele com a menor distância
21        para i de 0 até (NUMERO_VERTICES - 1) faça
22           se ((distancias[i] < D_min) E (visitado[i]) == 0) então
23              D_min = distancias[i]
24              prox = i
25           fimse
26        fimpara
27        visitado[prox] = 1
28        enquanto (vert <> NULO) faça
29           se (visitado[vert->numero] == 0) então
30              se ((D_min + vert->peso) < distancias[vert->numero]) então
31                 distancias[vert->numero] = D_min + vert->peso
32                 predecessor[vert->numero] = prox
33              fimse
34           fimse
35           cont = cont + 1
36           vert = vert->prox
37        fimenquanto
38     fimenquanto
39  fimfunção
```

FINALIZANDO

Nesta aula, aprendemos sobre estruturas de dados do tipo grafo. Aprendemos que grafos não apresentam uma estrutura fixa em sua topologia de construção e que são constituídos de vértices e arestas – que conectam os vértices.

Cada vértice do grafo contém um conjunto de vértices vizinhos, os quais são vértices conectados por meio de uma única aresta. Aprendemos que podemos representar um grafo de três maneira distintas: usando matriz de incidências, matriz de adjacências e lista de adjacências. Esta última constrói um grafo utilizando diversas estruturas de listas encadeadas, em que cada vértice terá uma lista contendo todos os seus vizinhos.

Vimos também dois algoritmos distintos de descoberta do grafo, ou seja, como passear pelos vértices uma única vez sem repeti-los. Para isso, conhecemos os algoritmos de busca por largura e de busca por profundidade.

Por fim, vimos um algoritmo de cálculo de rotas dentro de um grafo, que calcula o menor trajeto dentro de um grafo ponderado, além do algoritmo de Dijkstra.

REFERÊNCIAS

ASCENCIO, A. F. G. **Estrutura de dados**: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson, 2011.

_____. **Fundamentos da programação de computadores**: algoritmos, Pascal, C/C++ (padrão ANSI) JAVA. 3. ed. São Paulo: Pearson, 2012.

CORMEN, T. H. **Algoritmos**: teoria e prática. 3. ed. Rio de Janeiro: Elsevier, 2012.

LAUREANO, M. **Estrutura de dados com algoritmos e C**. Rio de Janeiro: Brasport, 2008.

MIZRAHI, V. V. **Treinamento em linguagem C**. 2. ed. São Paulo: Pearson, 2008.

PUGA, S.; RISSETI, G. **Lógica de programação e estrutura de dados**. 3. ed. São Paulo: Pearson, 2016.