PROGRAMAÇÃO III

AULA 6

Prof. Vinicius Pozzobon Borin

CONVERSA INICIAL

O objetivo desta etapa é apresentar os conceitos que envolvem a estrutura de dados do tipo grafo. Ao longo deste documento será conceituado essa estrutura de dados, bem como as formas de representação de um grafo. São elas: matriz de adjacências e lista de adjacências.

Ainda, será mostrado como realizar a descoberta de um grafo por intermédio de dois algoritmos distintos: algoritmo de busca por largura; e algoritmo de busca por profundidade.

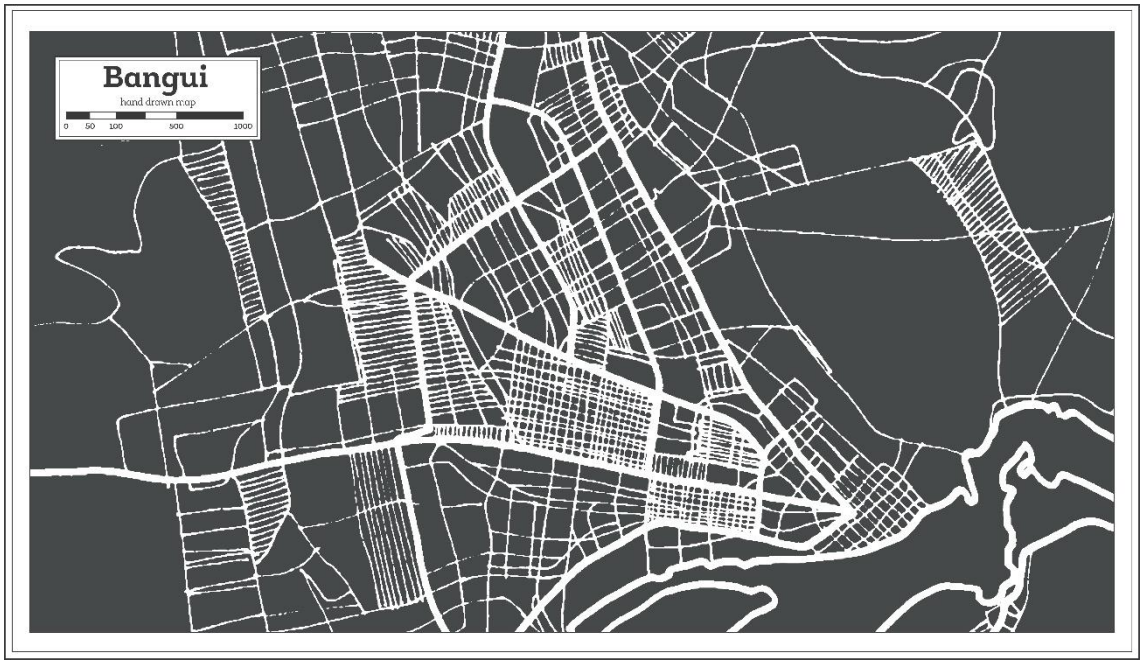
Por fim, será apresentado um algoritmo bastante tradicional para encontrar o caminho mínimo entre dois vértices em um grafo: algoritmo de Dijkstra.

TEMA 1 – CONCEITOS DE GRAFOS

Anteriormente, já investigamos estruturas de dados que funcionam de uma forma linear, como a lista encadeada, e também vimos as não lineares, na forma de árvores binárias. Porém, uma árvore, seja ela binária ou não, ainda segue um aspecto construtivo fixo da sua estrutura por meio da geração de ramos e subdivisões. Nesta etapa, vamos investigar uma estrutura de dados que pode vir a ter sua construção sem padrão de construção nenhum, os grafos.

Para entendermos o conceito e a aplicabilidade de grafos, vamos começar com um exemplo prático. Imaginemos que fomos contratados por uma empresa para mapear ruas, estradas e rodovias de uma cidade. Desse modo, recebemos um mapa aéreo da cidade e, por intermédio de um processamento digital de imagens, extraímos todas as ruas e avenidas dessa cidade, resultando em um mapa destacado semelhante ao da Figura 1, a seguir.

Figura 1 – Mapa rodoviário de uma cidade hipotética



Crédito: ShustrikS/Shutterstock.

Com esse mapa processado, precisaremos realizar o mapeamento das estradas com o objetivo de desenvolver um *software* que calcula as melhores rotas de um ponto de origem até um destino no mapa.

Para realizar esse cálculo de rotas, podemos mapear a rede rodoviária da Figura 1 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.

Na Figura 2, a seguir, há uma ilustração de 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 referida figura somente algumas ruas estejam mapeadas, poderíamos fazer esse processo para a cidade inteira. Com o mapeamento pronto, bastaria que aplicássemos um algoritmo para encontrar o menor caminho do grafo e encontrar as rotas desejadas de um ponto ao outro.

Figura 2 – Mapa rodoviário de uma cidade hipotética com vértices e arestas do grafo marcadas

Crédito: Vinicius Pozzobon Borin.

1.1 Definições de grafos

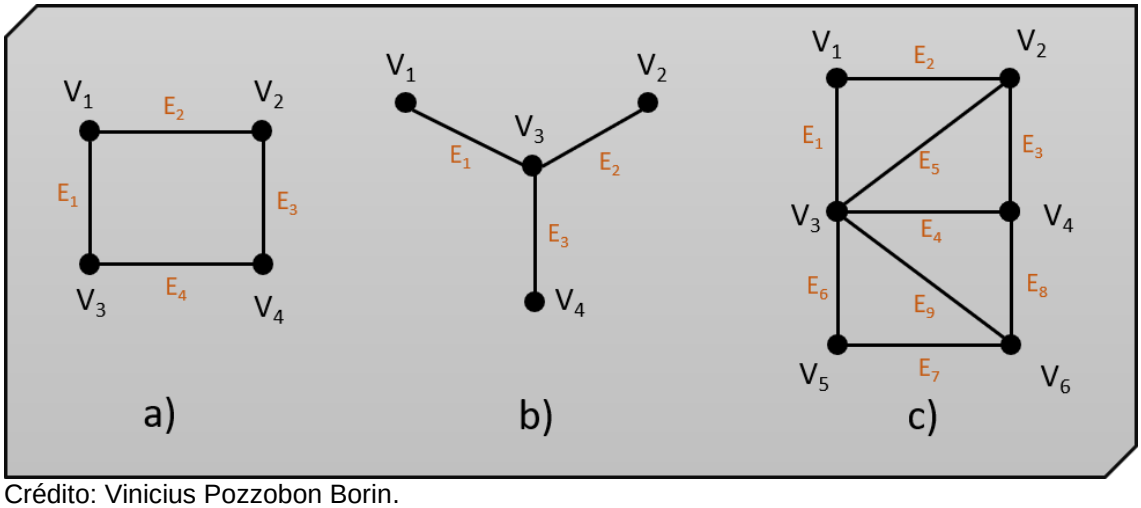
Um **Grafo** G é um conjunto de vértices conectados por meio de arestas sem uma distribuição fixa ou padronizada.

Vértices V de um grafo são seus pontos. Cada ponto poderá ser um ponto de encontro entre caminhos (rotas) de um grafo, ou então esse vértice poderá conter informações relevantes para o grafo, como dados de informações de cadastros. Tudo dependerá da aplicação.

Arestas E são as linhas de conexão entre vértices. Cada aresta conecta dois vértices. Nem todo vértice precisa ter uma aresta conectada, pois pode permanecer isolado caso o grafo assim seja construído.

A Figura 3, a seguir ilustra três exemplos de grafos. No exemplo da Figura 3ª, há 4 vértices e 4 arestas. Na Figura 3b, também há 4 vértices, mas somente 3 arestas. Na Figura 3c, é interessante perceber que um mesmo vértice chega a conter 5 arestas partindo dele. A quantidade de vértices e arestas em um grafo pode ser ilimitada, não havendo restrições.

Figura 3 – Exemplos de grafos



Crédito: Vinicius Pozzobon Borin.

Para a construção do grafo não existe uma regra fixa. Qualquer arranjo de vértices e arestas pode ser considerado um grafo. Vejamos alguns conceitos atrelados a essa construção:

- grafo completo: quando existe uma, e somente uma aresta para cada par distinto de vértices;
- grafo trivial: é um grafo com unicamente um vértice.

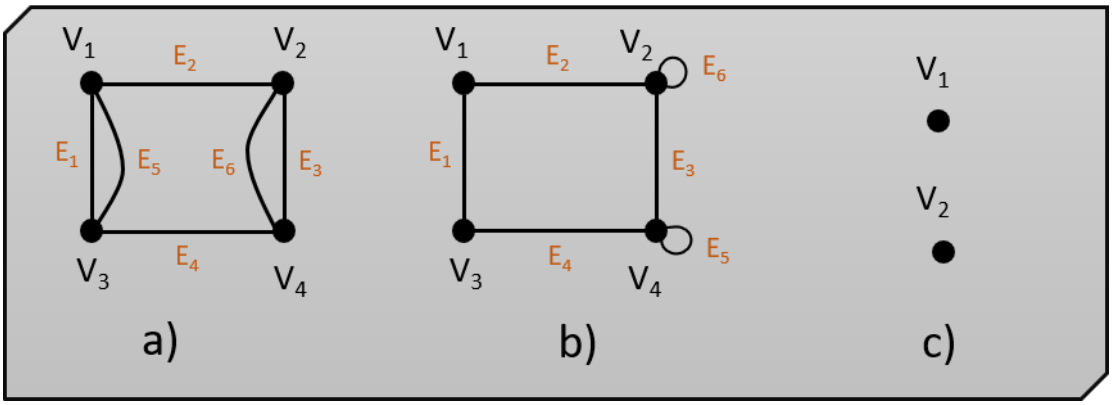
Assim como tínhamos o grau de cada nó de uma árvore, em grafos também podemos encontrar o **grau de cada nó (vértice)**. O grau de um vértice nada mais é do que a soma de todas as arestas incidentes nele. Por exemplo, na Figura 3 (a) anterior, todos os vértices têm grau 2, já na Figura 3 (c) temos vértices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

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

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

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 4, ambos estão sem arestas, mas isso não é obrigatório.

Figura 4 – Peculiaridades na construção de grafos

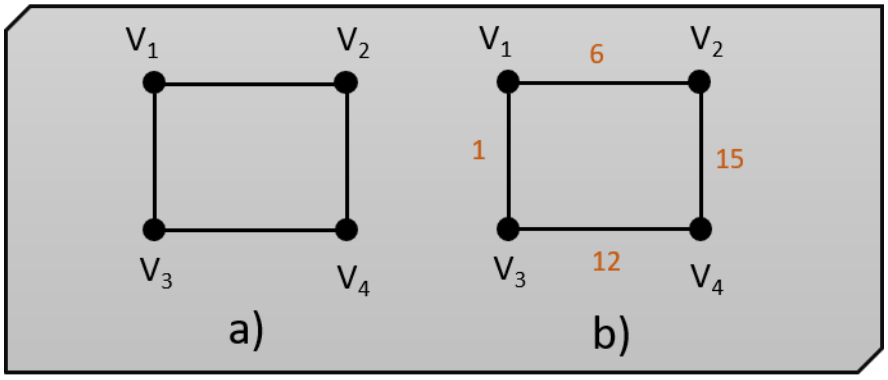


Crédito: Vinicius Pozzobon Borin.

Finalizando nossos conceitos teóricos, precisamos entender que podemos atribuir pesos para nossas arestas. O peso da aresta representa um custo atrelado àquele caminho. Por exemplo, vamos voltar ao nosso exemplo de mapeamento de estradas. Caso desejarmos traçar uma rota de um ponto A até outro ponto B da cidade mapeada, os pesos nas arestas poderiam ser calculados por meio do tráfego de veículos da rua, por exemplo. Enquanto 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 todas elas têm 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



Crédito: Vinicius Pozzobon Borin.

1.2 Aplicações de grafos

Acerca da aplicabilidade da estrutura de dados do tipo grafo, a gama de aplicações é bastante grande. Citamos:

- encontrar rotas e melhores trajetos em mapas;
- escrever algoritmos de inteligência artificial que calculam o menor número de movimentos necessários para a vitória em uma partida de damas, ou xadrez;
- mapear um jogo de tabuleiro para criar jogadas e movimentos;
- encontrar algo bastante próximo de nós, como o médico mais próximo conveniado ao nosso plano de saúde;
- conexões e tabelas de roteamento em redes de computadores;
- mapeamento de interações em redes sociais.

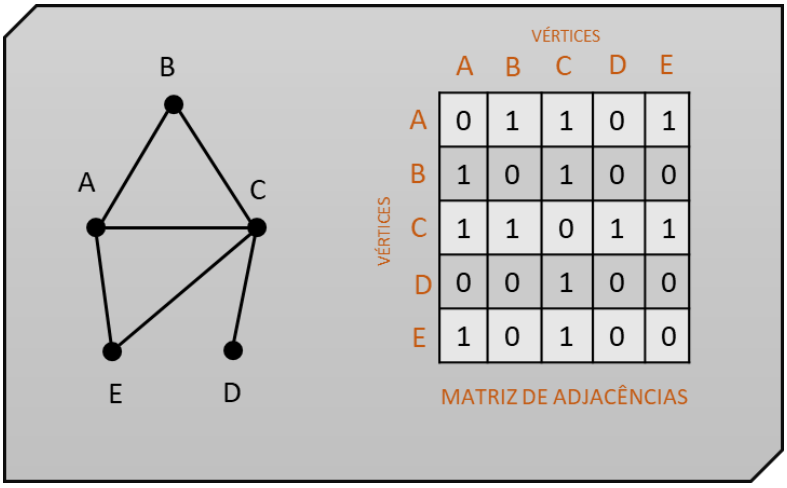
TEMA 2 – REPRESENTAÇÃO DE GRAFOS

Vamos apresentar duas maneiras distintas e bastante comuns de representação. Todas aqui apresentadas poderão ser implementadas em programação.

2.1 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 será o número de vértices do grafo. Por exemplo, se o grafo contém 10 vértices, teremos uma matriz 10x10, não importando o número de arestas. Na Figura 6 a seguir, há a ilustração de um exemplo de matriz de adjacências.

Figura 6 – Exemplo de matriz de adjacências



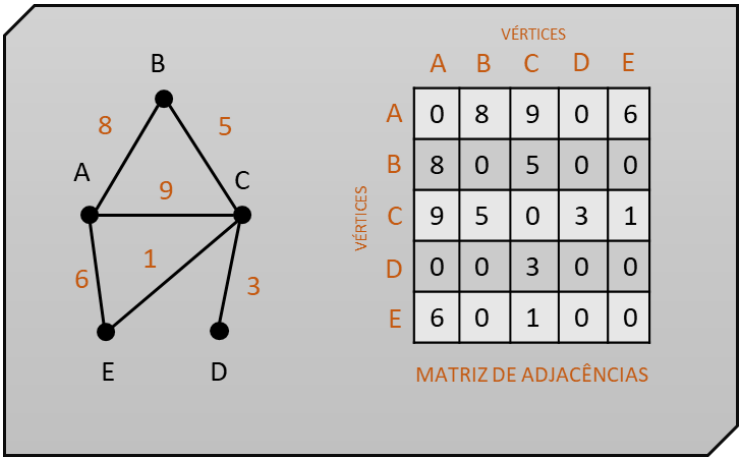
Crédito: Vinicius Pozzobon Borin.

Assim como na representação anterior, povoamos nossa matriz com valores 0 ou 1. 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:

$$\begin{cases} 0, \text{ caso o outro vértice não tenha conexão com o vértice analisado} \\ 1, \text{ caso o outro vértice tenha conexão com o vértice analisado} \end{cases}$$

Grafos podem, eventualmente, apresentar pesos em suas arestas, os quais são chamados de **grafos ponderados**. O emprego de uma matriz de adjacências facilita o uso para grafos ponderados, pois o peso de cada aresta é explicitado na estrutura. Vejamos a Figura 7 a seguir.

Figura 7 – Exemplo de matriz de adjacências para grafo ponderado



Crédito: Vinicius Pozzobon Borin.

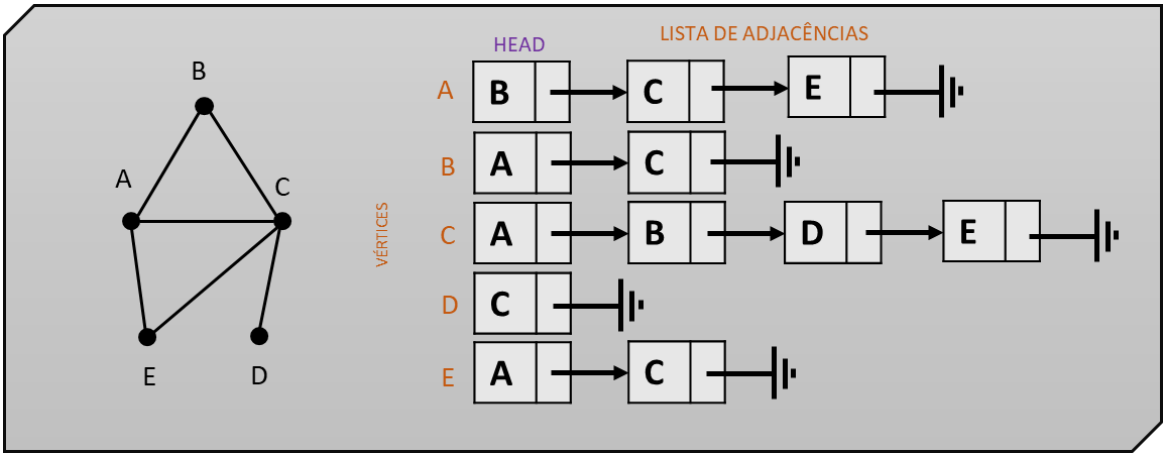
2.2 Lista de adjacências

A representação de um grafo por uma lista de adjacências é muito adotada no meio de programação, pois trabalha com listas encadeadas e manipulação de ponteiros de dados.

A Figura 8 a seguir contém um exemplo de lista de adjacências. A proposta dessa representação é a de criar um *array* do tamanho da quantidade de vértices existentes no grafo. Em cada posição desse *array* teremos uma lista encadeada contendo os endereços dos "vizinhos" daquele vértice. Conceitualmente, **vizinhos de um vértice são todos os outros vértices que estão conectados a ele**.

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

Figura 8 – Exemplo de lista de adjacências



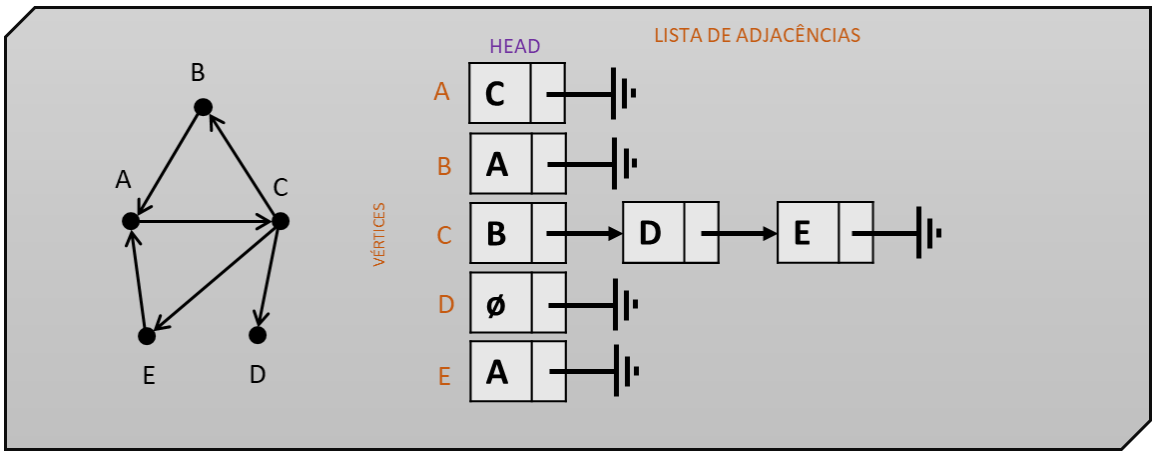
Crédito: Vinicius Pozzobon Borin.

No exemplo da Figura 8 anterior, teremos cinco vértices, portanto, podemos ter um *array* de dimensão 5, em que cada posição do array terá o endereço do *head* (também vizinho) para uma lista encadeadas de vizinhos daquele vértice.

Observemos o vértice A: ele está na primeira posição do *array* (posição zero), assim como contém como vizinho o vértice B, vértice C e o vértice E. Assim, na posição zero do *array* temos o endereço para um dos vizinhos de A. Esse vizinho será o *head* de uma lista encadeada. O vizinho escolhido para ser o *head* foi o vértice B. Assim, B apontará para o outro vizinho, C, que por sua vez aponta para E. Teremos uma lista encadeada de vizinho do vértice A. De forma análoga, criamos uma lista encadeada de vizinhos para cada vértice existente no grafo.

Grafos também podem ser do tipo dirigido. Todos os exemplos que vimos até então são não dirigidos. Um **gafo dirigido** é aquele que contém um único sentido de conexão entre os vértices e representamos esse sentido por uma seta. Vejamos a Figura 9 a seguir.

Figura 9 – Exemplo de lista de adjacências em grafos dirigidos



Crédito: Vinicius Pozzobon Borin.

É interessante notar que representamos como vizinho de cada vértice somente aqueles os quais respeitam os sentidos das setas. O vértice D não tem vizinhos, pois sobre ele há somente índice uma aresta vinda de C.

Embora seja possível implementar um grado dirigido com matriz de adjacências, o mais usual é o emprego da lista de adjacências.

2.3 Comparativo das representações

Podemos comparar o desempenho das representações de acordo com a quantidade de vértices e arestas contidas no grafo. Temos um **grafo denso** quando a quantidade de aresta E é equivalente ao quadrado da quantidade de vértices V, e um **grafo esparso** quando o total de arestas for igual ao de vértices:

- Grafo denso: $|E| = |V|^2$
- Grafo esparso: $|E| = |V|$

Em uma matriz de adjacência, sempre utilizaremos $|V|^2$ de espaço na memória. Ou seja, se tivermos um grafo com 10 mil vértices, teríamos 100.000.000 *bytes* de uso de memória. Portanto, uma desvantagem dessa representação é o excesso de uso de memória, especialmente para grafos grandes.

Em termos de complexidade de tempo de execução, a matriz é mais rápida que a lista de adjacência para grafos densos, pois acessa qualquer dado com o mesmo tempo. Por fim, essa representação é mais simples de ser aplicada para grafos ponderados.

Já uma lista de adjacência é mais rápida se comparada à matriz, e emprega menos espaço de memória para grafos esparsos. Em contrapartida,

para grafos bastante densos são mais lentas do que a versão matricial, isso porque grafos densos tendem a formar grandes listas de vizinhos, que quando representadas por listas encadeadas geram um tempo de acesso ao dado bastante elevado.

TEMA 3 – BUSCA EM PROFUNDIDADE

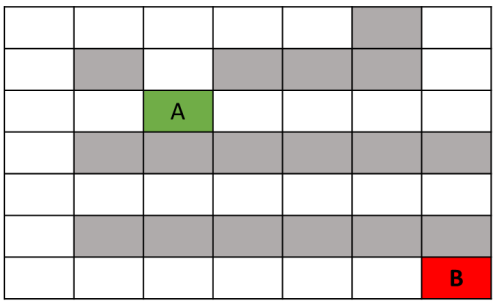
Após entendermos como construir um grafo, vamos analisar como caminhamos pelos elementos dessa estrutura de dados. Então, vamos assumir que temos um grafo já construído utilizando uma lista de adjacências.

Agora, queremos andar por cada vértice do grafo, sem repetir nenhum, 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, mais conhecido pelo seu nome inglês ***Depth-First Search*** **(DFS)**, apresenta uma lógica bastante simples e funcional para resolver esse problema de descoberta do grafo.

Vamos acompanhar o funcionamento desse algoritmo com um exemplo. Para um melhor entendimento inicial, adotaremos uma analogia de um labirinto. Na Figura 10 a seguir, temos um labirinto representado por um tabuleiro 7x7. Queremos partir de um ponto A e chegar em um ponto B.

Figura 10 – Labirinto 7x7 para exemplo



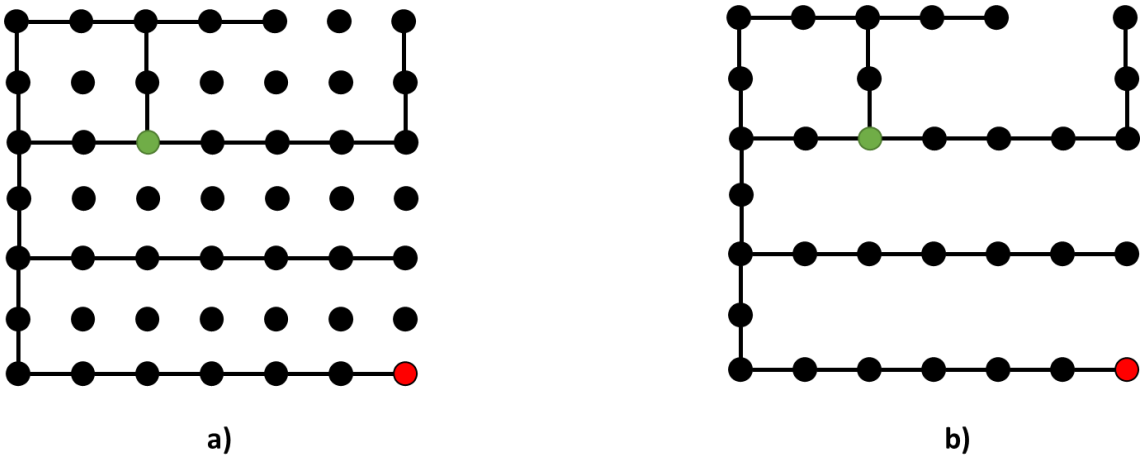
Crédito: Vinicius Pozzobon Borin.

Podemos entender que cada quadradinho do labirinto pode ser considerado um vértice. Se houver conexão entre dois quadradinhos, ou seja, se eles se encostam, existe, portanto, uma aresta de conexão entre eles.

Não existe aresta nos quadradinhos marcados em cinza. Esses quadrados são considerados paredes em nosso labirinto. Podemos representar o labirinto como um grafo quadrado de 7x7 vértices, conforme Figura 11 (a) a

seguir. Note que alguns vértices ficam sem conexão, pois são as paredes. Para fins de simplificação, podemos eliminar esses vértices do grafo (Figura 11, b).

Figura 11 – Grafo representativo do labirinto 7x7



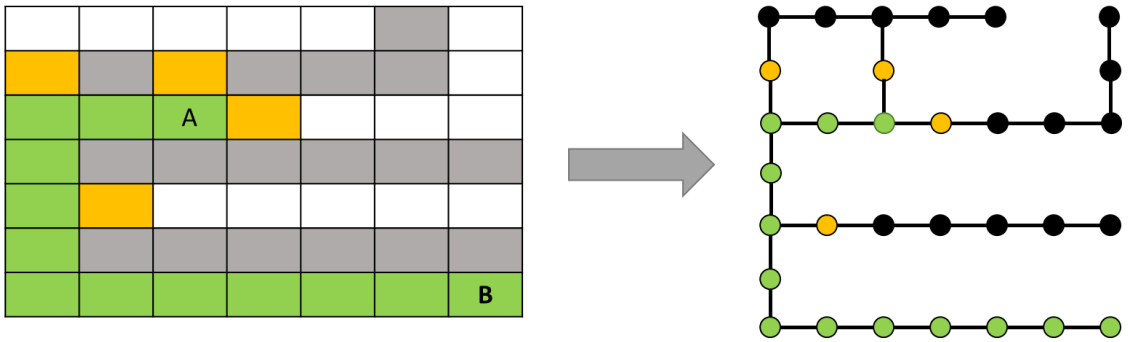
Crédito: Vinicius Pozzobon Borin.

Precisamos agora partir do ponto A, em verde, até chegar no ponto B, em vermelho utilizando o algoritmo de busca em profundidade. Lembre-se de que cada vértice só tem o conhecimento de quem são seus vizinhos. Portanto, ele não tem a menor ideia de que caminho deve seguir para chegar em B.

A DFS faz o seguinte procedimento: como cada vértice tem uma lista de vizinhos, a DFS simplesmente acessa o primeiro vizinho da lista daquele vértice e avança para ele. No novo vértice, acessa-se o primeiro vizinho novamente, e assim por diante, ou seja, vamos acessando o primeiro vizinho de cada lista até atingirmos o nosso objetivo.

Esta lógica pode nos gerar situações com desempenhos bastantes distintos. Imaginemos que, por coincidência, os primeiros vizinhos de cada vértices sigam exatamente em direção ao ponto B. Faremos um caminho direto em direção ao ponto B. Vejamos a Figura 12 a seguir.

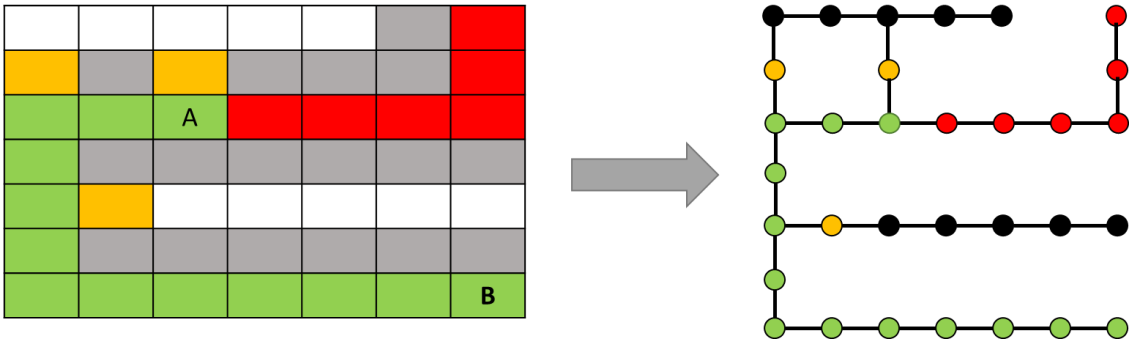
Figura 12 – Labirinto indicando o trajeto em verde de A até B: em amarelo, os outros vizinhos não acessados



Crédito: Vinicius Pozzobon Borin.

Todavia, podemos ter uma situação oposta. Ou seja, imaginemos que, partindo do ponto A, o primeiro vizinho seja para um caminho oposto. Então, percorreríamos o trajeto até um ponto sem saída, conforme pode ser visto na Figura 13 a seguir. Somente ao atingirmos o ponto final do trajeto é que retornamos para o último vértice com trajetos disponíveis e tomamos outro caminho, este em direção ao ponto B, por exemplo.

Figura 13 – Labirinto indicando o trajeto em verde de A até B

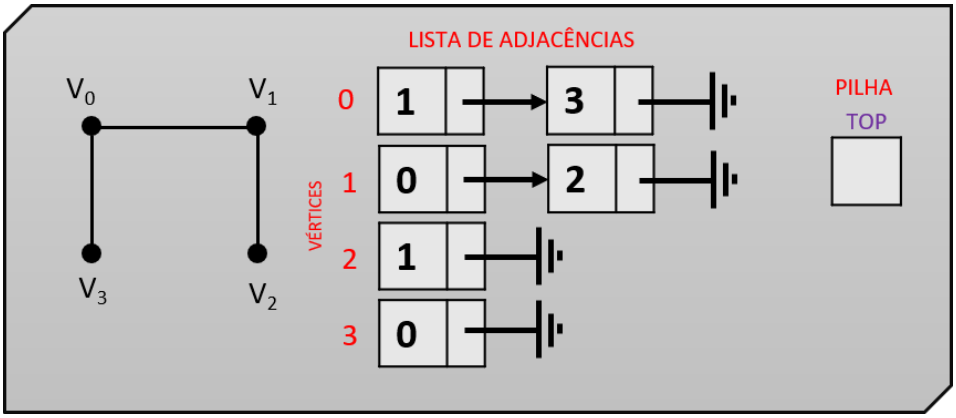


Crédito: Vinicius Pozzobon Borin.

Por meio do exemplo do labirinto foi possível notar que o algoritmo de busca em profundidade escolhe um caminho e vai profundamente percorrendo ele até tentar outro trajeto.

Vamos agora investigar um grafo mais simples com o objetivo de compreendermos o algoritmo da DFS. Na Figura 14 a seguir, vemos o grafo que vamos utilizar.

Figura 14 – Busca em profundidade no grafo: estado inicial



Crédito: Vinicius Pozzobon Borin.

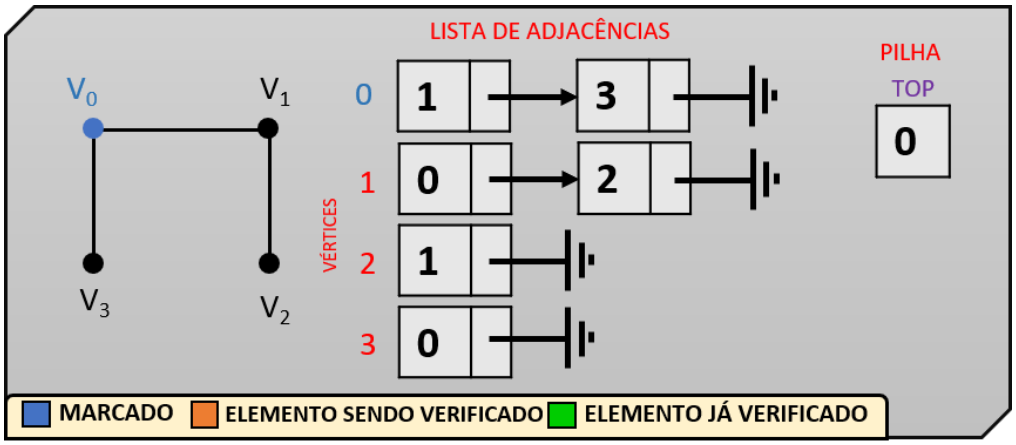
Queremos andar por todos os vértices, partindo de um vértice de origem. O primeiro vizinho detectado (mais na frente na lista encadeada daquele vértice) será imediatamente acessado.

Assim, cada novo elemento ainda não descoberto é selecionado com base nas listas encadeadas de cada vértice até que o último seja encontrado. Os elementos descobertos vão sendo empilhados e desempilhados seguindo 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.

Vamos analisar o grafo da Figura 14 anterior. Mostraremos como se dá o funcionamento lógico desse algoritmo e mostraremos graficamente a descoberta do grafo. O grafo contém quatro vértices iniciando em zero e três arestas. Podemos construir a lista de adjacências desse grafo. Ainda, é importante observar que temos uma pilha que, neste momento, está vazia.

Desejamos iniciar a descoberta de nosso grafo pelo vértice zero. Na Figura 15 a seguir, vemos que esse vértice é então marcado como já visitado (cor azul). Assim, não é possível mais revisitá-lo. Em nossa pilha, que estava vazia, colocamos nosso vértice inicial zero.

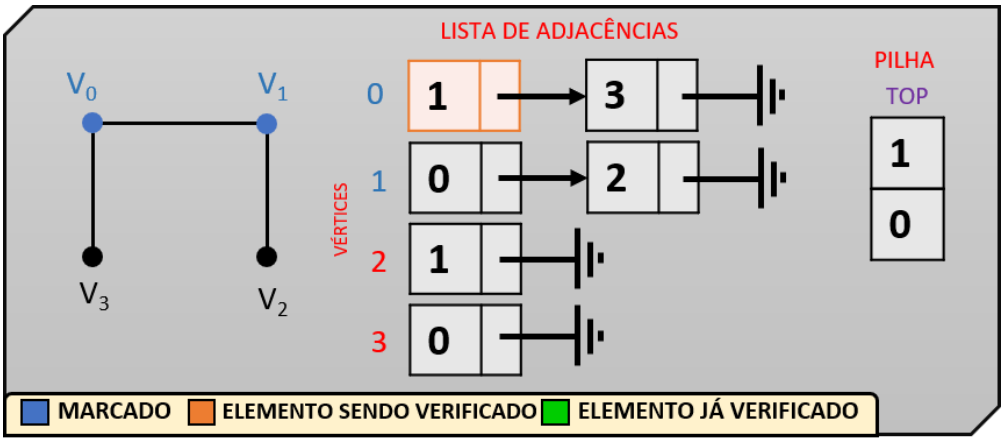
Figura 15 – Busca em profundidade no grafo, partindo de V_0: etapa 1



Crédito: Vinicius Pozzobon Borin.

Como esse vértice V_0 está no topo da pilha, vamos acessar sua respectiva lista de vizinhos. Acessamos, então, o primeiro elemento da lista encadeada de vizinhos de V_0, que é o vértice V_1 (destacado em laranja na Figura 16 a seguir). Imediatamente, colocamos esse vértice como já visitado (cor azul) no grafo e inserimos ele no topo da pilha, acima de V_0. Agora, temos dois dos quatro vértices já visitados.

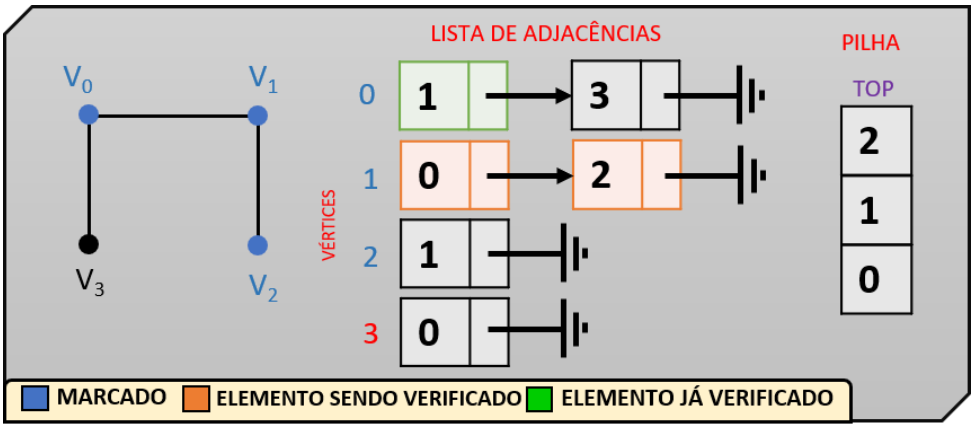
Figura 16 – Busca em profundidade no grafo, partindo de V_0: etapa 2



Crédito: Vinicius Pozzobon Borin.

O próximo vértice a ser visitado será o primeiro vizinho da lista encadeada do vértice V_1. É importante perceber que nem terminamos de varrer toda a lista encadeada de V_0, mas 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. Este, ainda não visitado (em laranja), é então marcado como visitado e colocado no topo da pilha, que agora contém três elementos. Vejamos a Figura 17 a seguir.

Figura 17 – Busca em profundidade no grafo, partindo de V_0: etapa 3

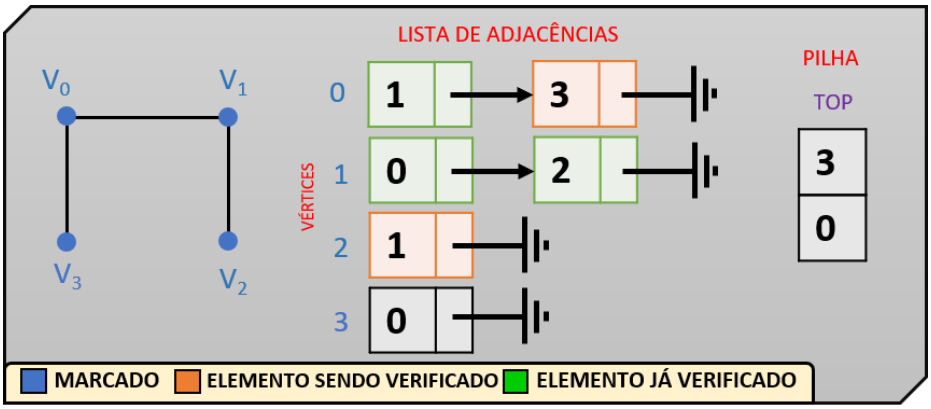


Crédito: Vinicius Pozzobon Borin.

Nesse momento, estamos na lista encadeada do vértice V_2. Na Figura 18 a seguir, vemos que precisamos acessar o primeiro vizinho do vértice V_2 (em laranja); esse vizinho é o vértice V_1, que já foi visitado. Ainda, toda a lista de vizinhos do vértice V_2 já foi verificada, não havendo mais nenhum vizinho de V_2.

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 ele, voltamos para V_0 e acessamos o segundo elemento da lista encadeada, pois o primeiro havia sido visitado anteriormente. Encontramos, assim, o último vértice não visitado, o V_3. Marcamos ele e colocamos no topo da pilha.

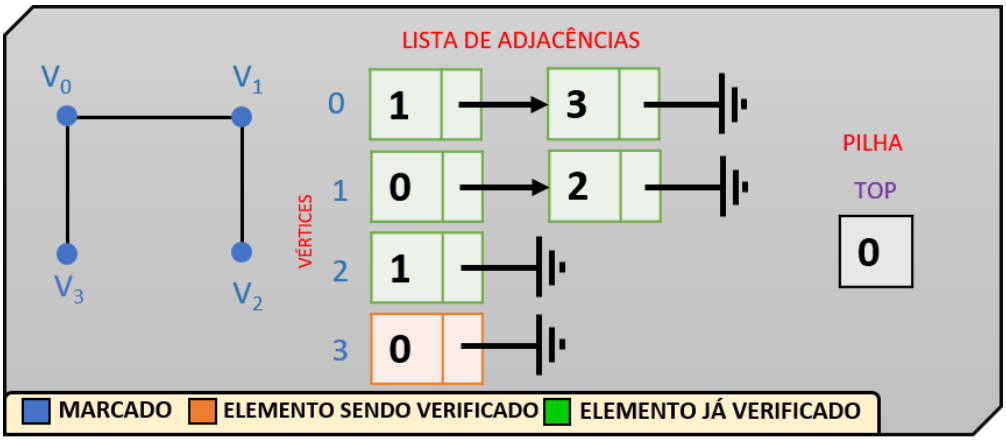
Figura 18 – Busca em profundidade no grafo, partindo de V_0: etapa 4



Crédito: Vinicius Pozzobon Borin.

Todos os vértices já haviam sidos visitados na etapa anterior. Porém, ainda precisamos encerrar nosso algoritmo. Na Figura 19 a seguir, no vértice V_3, verificamos o primeiro vizinho dele na lista encadeada, o vértice V_0, anteriormente marcado. Chegamos ao final da lista; desempilhamos V_3, restando somente V_0.

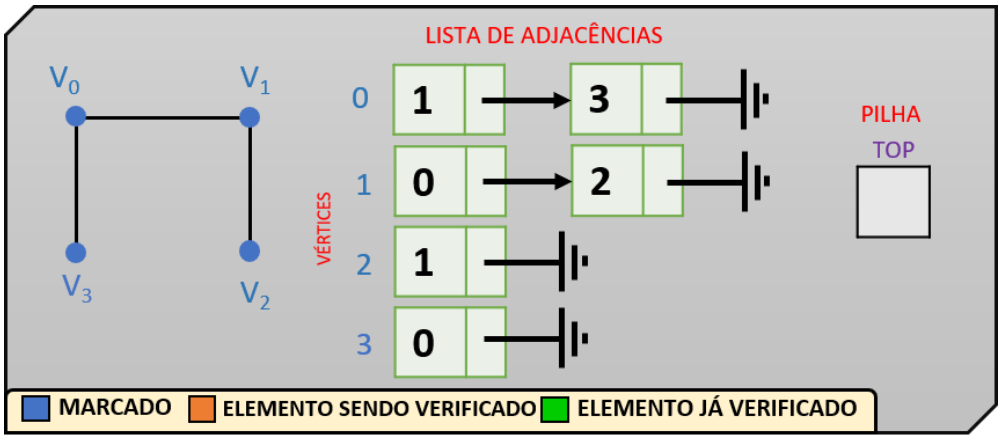
Figura 19 – Busca em profundidade no grafo, partindo de V_0: etapa 5



Crédito: Vinicius Pozzobon Borin.

Embora V_0 já tenha sido visitado, faltava desempilhá-lo (Figura 20). Fazendo isso, temos nossa pilha vazia e todos os quatro vértices visitados, encerrando nosso algoritmo de busca por profundidade.

Figura 20 – Busca em profundidade no grafo, partindo de V_0: etapa 6



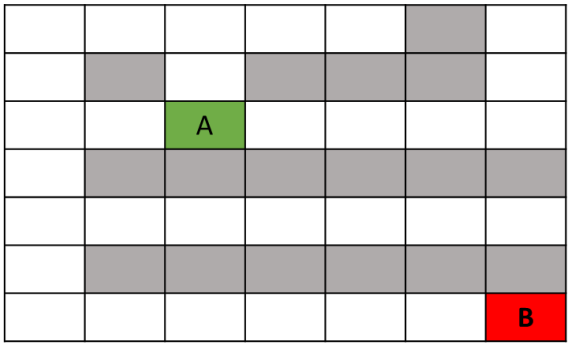
Crédito: Vinicius Pozzobon Borin.

TEMA 4 – BUSCA EM LARGURA

O algoritmo de busca em largura, mais conhecido pelo seu nome inglês ***Breath-First Search*** **(BFS)**, trabalha com a ideia de visitar primeiramente todos os vizinhos de um mesmo vértice atualmente marcado antes de pular para o próximo vértice.

Vamos acompanhar o funcionamento desse algoritmo com um exemplo. Para um melhor entendimento inicial, adotaremos uma analogia de um labirinto. Na Figura 21 a seguir, temos um labirinto representado por um tabuleiro 7x7. Queremos partir de um ponto A e chegar em um ponto B.

Figura 21 – Labirinto 7x7 para exemplo

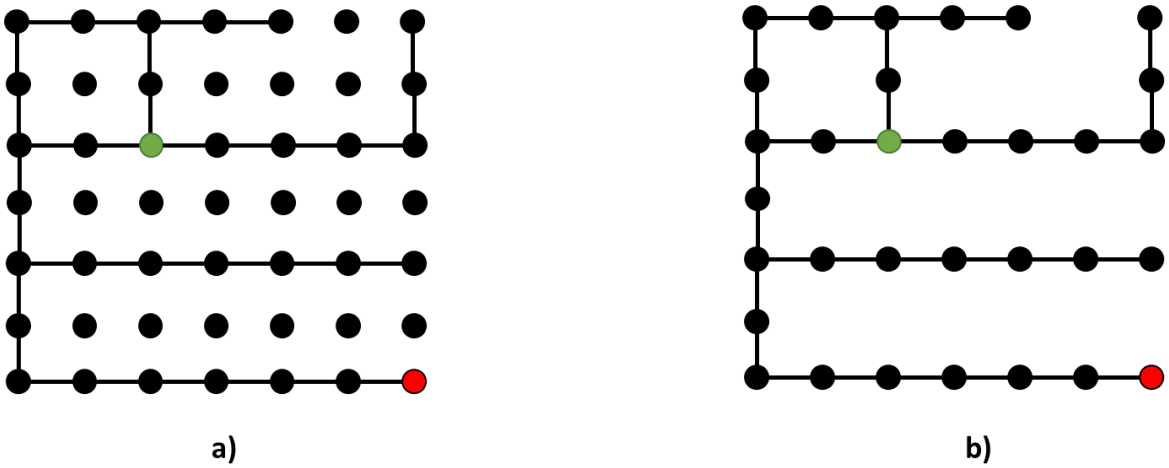


Crédito: Vinicius Pozzobon Borin.

Podemos entender que cada quadradinho do labirinto pode ser considerado um vértice. Se houver conexão entre dois quadradinhos, ou seja, se eles se encostam, existe, portanto, uma aresta de conexão entre eles.

Não existe aresta nos quadradinhos marcados em cinza. Esses quadrados são considerados paredes em nosso labirinto. Podemos representar o labirinto como um grafo quadrado de 7x7 vértices, conforme Figura 22 (a) a seguir. Notemos que alguns vértices ficam sem conexão, pois são as paredes. Para fins de simplificação, podemos eliminar esses vértices do grafo (Figura 22, b).

Figura 22 – Grafo representativo do labirinto 7x7

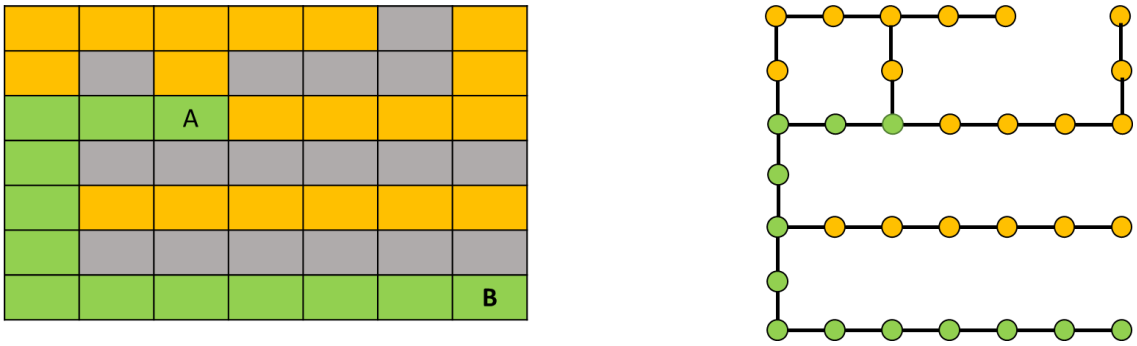


Crédito: Vinicius Pozzobon Borin.

Precisamos agora partir do ponto A, em verde, até chegar ao ponto B, em vermelho, utilizando um algoritmo de busca em largura. A BFS faz o seguinte procedimento: ao invés de tomar um caminho como único e seguido até o final, a BFS anda para todos os lados simultaneamente. Ela anda um pouco em cada um dos caminhos possíveis.

Vejamos a Figura 23, a seguir. Partindo do ponto A, temos três caminhos possíveis: esquerda, direta e cima. A BFS vai andar para todas essas direções simultaneamente. Dessa maneira, deixamos marcado em amarelo todos os caminhos traçados.

Figura 23 – Grafo representativo do labirinto 7x7

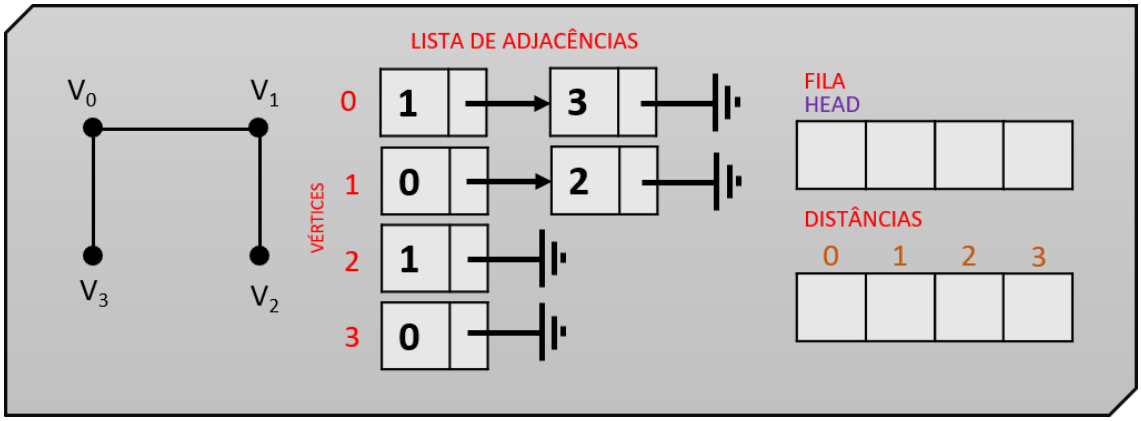


Crédito: Vinicius Pozzobon Borin.

Uma diferença interessante em relação à busca por profundidade é que ele trabalha com uma estrutura do tipo fila para indicar qual o próximo vértice a ser acessado, já na DFS tínhamos uma estrutura de pilha.

Vamos acompanhar o funcionamento desse algoritmo com um exemplo. Na Figura 24, a seguir, está ilustrado o grafo que será trabalhado. É interessante perceber que temos agora um *array* de distâncias, que 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 24 – Busca em largura no grafo, partindo de V_0: estado inicial



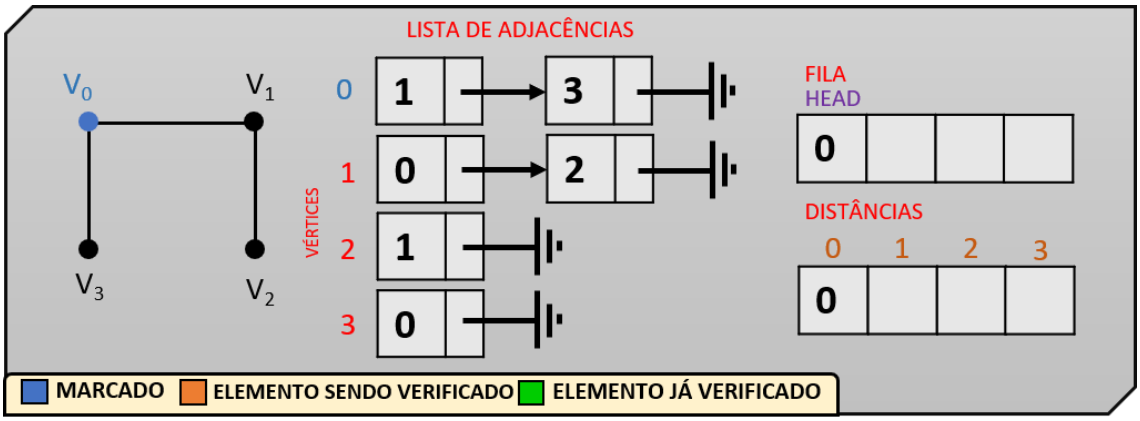
Crédito: Vinicius Pozzobon Borin.

A proposta desse algoritmo é a de partir de um vértice de origem e acessar a lista de adjacências daquele vértice, ou seja, seus vizinhos. Partindo da lista de vizinhos, calcula-se a distância deles para o vértice de origem e salva-se em

um vetor de distâncias. Os elementos vão sendo enfileirados à medida que vão sendo acessados nas listas encadeadas.

Iniciamos nossa análise no vértice V_0 novamente. Nesse modo, podemos imediatamente marcá-lo como já visitado (cor azul). Colocamos também esse vértice na fila. Lembre-se de que, em uma fila, a inserção acontece sempre no final dela (Fifo). Como a fila estava vazia, o inserimos na primeira posição. A distância do vértice de origem para ele mesmo será sempre zero (Figura 25).

Figura 25 – Busca em largura no grafo, partindo de V_0: etapa 1

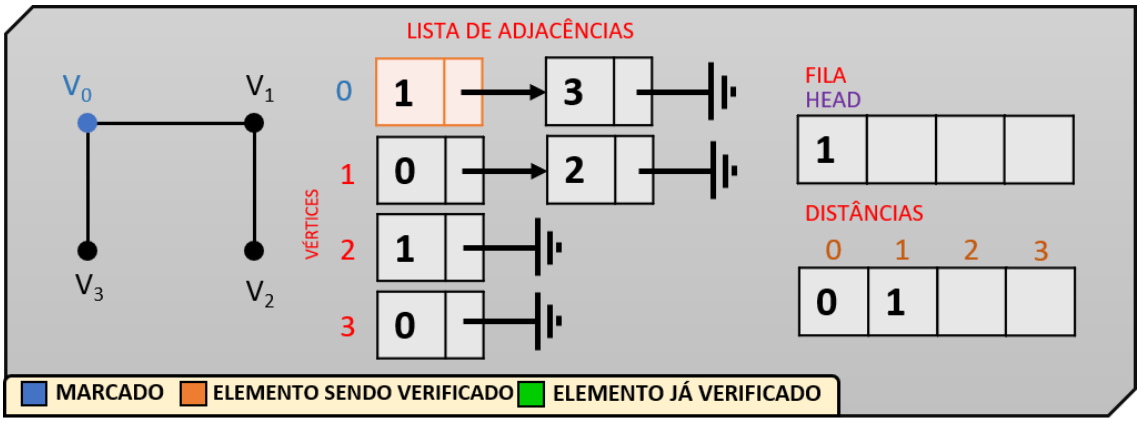


Crédito: Vinicius Pozzobon Borin.

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 26, a seguir, encontramos o primeiro vizinho de V_0, que é V_1. Colocamos V_1 na fila e calculamos a distância dele para a origem. O cálculo da distância é feito usando o valor da distância do vértice atual (valor zero) e acrescentando uma unidade, resultando em distância um $(0 + 1 = 1)$, conforme Figura 26, a seguir.

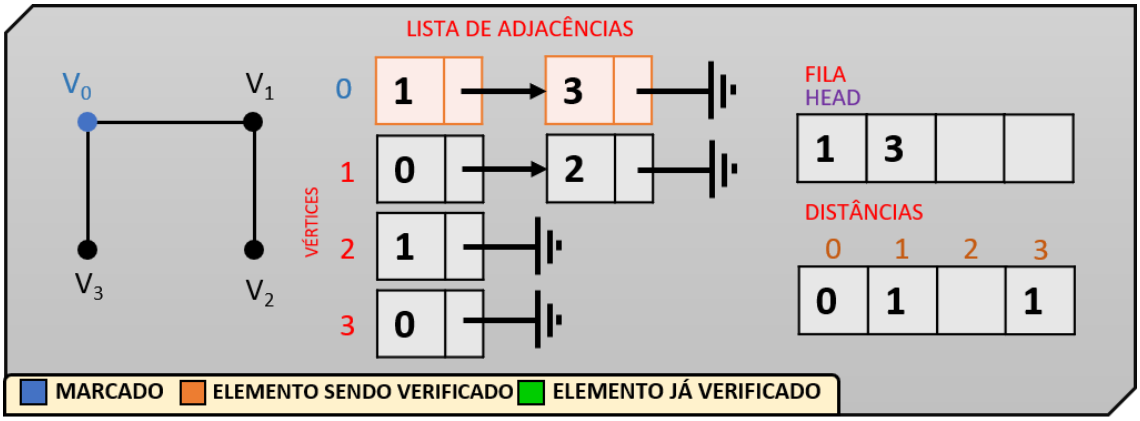
Figura 26 – Busca em largura no grafo, partindo de V_0: etapa 2



Crédito: Vinicius Pozzobon Borin.

Na Figura 27, a seguir, passamos para o próximo vizinho do vértice V_0. O vértice V_3 recebe o mesmo tratamento que V_1: sua distância é de valor um $0 + 1 = 1$, pois está a um salto de distância da origem, e ele é colocado no final da fila, após o vértice V_1. Assim, encerramos a varredura dos vizinhos do vértice V_0 e calculamos as distâncias, mas ainda não visitamos eles.

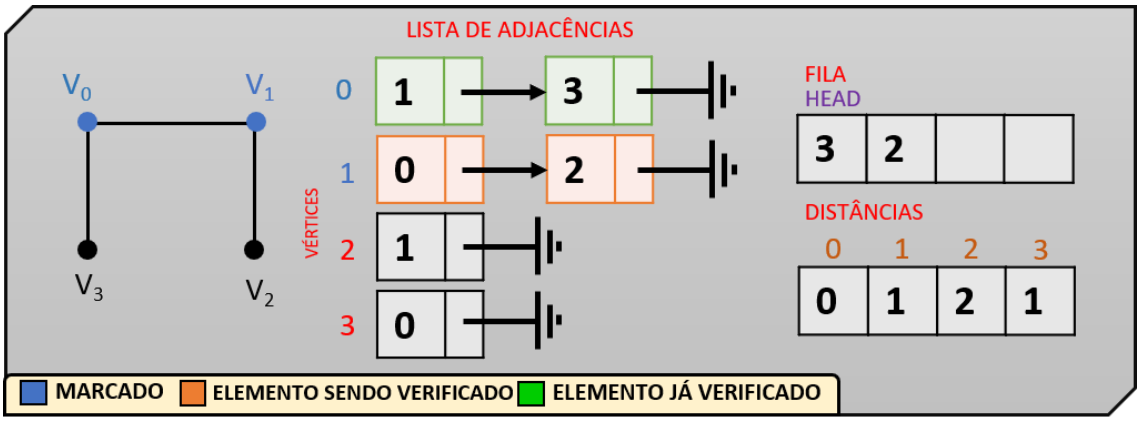
Figura 27 – Busca em largura no grafo, partindo de V_0: etapa 3



Crédito: Vinicius Pozzobon Borin.

Na Figura 28, a seguir, com V_0 encerrado (cor verde), vamos para o próximo vértice da fila, o vértice V_1. Marcamos ele, o removemos da fila e acessamos sua lista encadeada de vizinhos. O primeiro vizinho é o vértice V_0, que já foi visitado, portanto, seguimos para o próximo vértice, que é V_2. Esse vértice ainda não foi visitado, então colocamos ele ao final da fila e calculamos uma distância. Como ele está a dois vértices de distância da origem, sua distância é dois $(1 + 1 = 2)$.

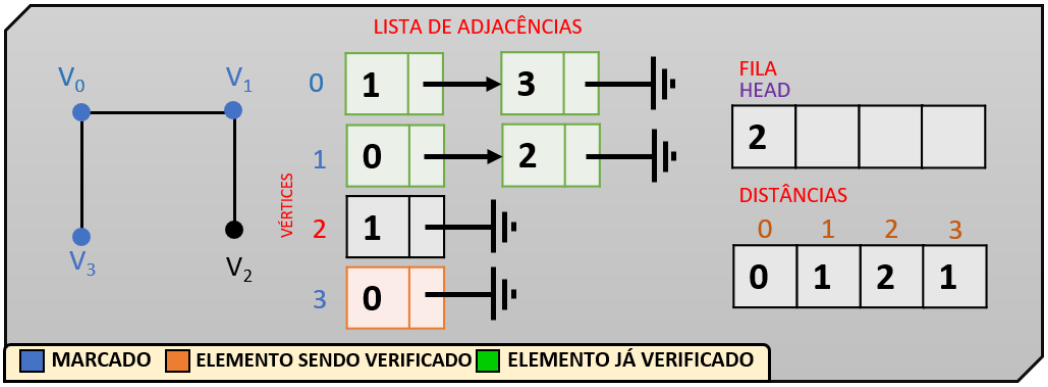
Figura 28 – Busca em largura no grafo, partindo de V_0: etapa 4



Crédito: Vinicius Pozzobon Borin.

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 29), removemos da fila o próximo vértice V_3 e marcamos ele (cor azul). Acessamos, então, a sua lista de vizinhos. Nela, só existe o vértice V_0, já acessado. Portanto, nada mais fazemos nessa etapa e podemos passar para o próximo elemento da nossa fila.

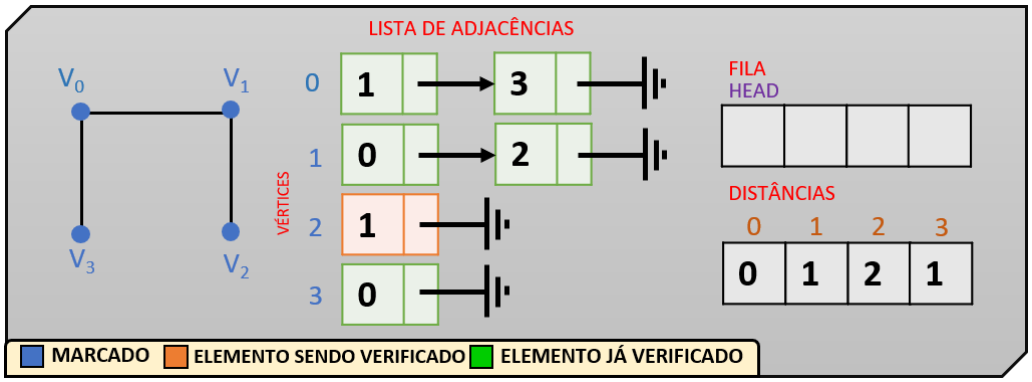
Figura 29 – Busca em largura no grafo, partindo de V_0: etapa 5



Crédito: Vinicius Pozzobon Borin.

Na Figura 30, a seguir, 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 fazemos nessa etapa.

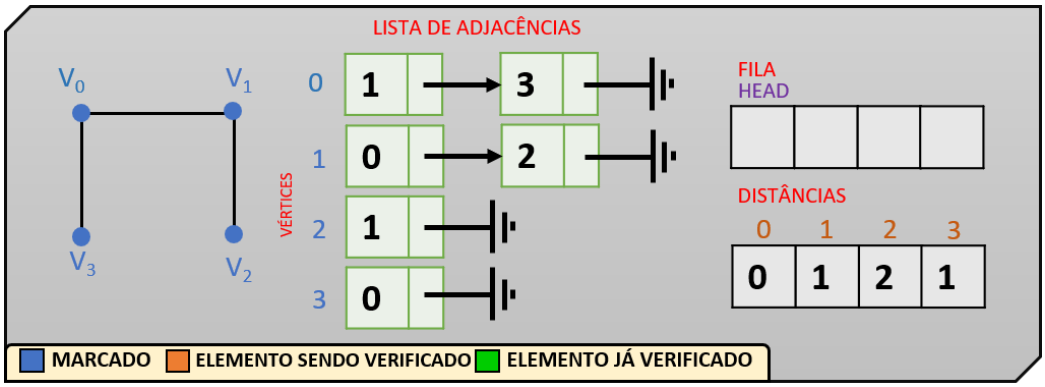
Figura 30 – Busca em largura no grafo, partindo de V_0: etapa 6



Crédito: Vinicius Pozzobon Borin.

Por fim, na etapa 7 (Figura 31), estamos com todos nossos elementos já visitados e todas as listas encadeadas varridas. Nosso algoritmo se encerra aqui.

Figura 31 – Busca em largura no grafo, partindo de V_0: etapa 7



Crédito: Vinicius Pozzobon Borin.

4.1 Comparativo de desempenho: DFS x BFS

A única e real diferença no funcionamento de ambas as buscas no grafo é que a BFS opera com uma estrutura de fila, já a DFS com uma pilha. Então qual o impacto disso na complexidade *Big-O*?

A análise não é trivial, mas vamos assumir que |V| é nosso total de vértices, e que |E| é o total de arestas. Inicialmente, independentemente do algoritmo escolhido, precisamos inicializar todos nossos vértices. Se considerarmos esse tempo de inicialização, temos para tal um O(|V|).

No BFS, com a fila, fazemos inserções e remoções sem dependência do tamanho do conjunto de dados, ou seja, o acesso ao dado, como inserção e remoção, é sempre O(1). Todavia, nesse algoritmo precisamos verificar as distâncias de um vértice com todos os seus vizinhos, portanto, em um pior cenário, temos esse procedimento sendo realizado para todas as arestas: O(|E|).

Como fazemos esse cálculo uma só vez para todo o algoritmo, ficamos com O(|E|). Juntando a inicialização com a complexidade do BFS, temos: O(|V| + |E|).

De maneira bastante similar, o DFS opera com uma pilha e também tem tempo constante O(1), mas realizado o procedimento para cada uma das arestas. Sendo assim, a complexidade da DFS também será O(|V| + |E|).

TEMA 5 – CAMINHO MÍNIMO (ALGORITMO DE DIJKSTRA)

Vamos agora estudar um algoritmo que, por meio de um grafo conhecido, encontra a menor rota (caminho com menor peso) entre dois vértices.

O algoritmo investigado neste tópico será o algoritmo de Dijkstra, que tem esse nome em razão do 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.

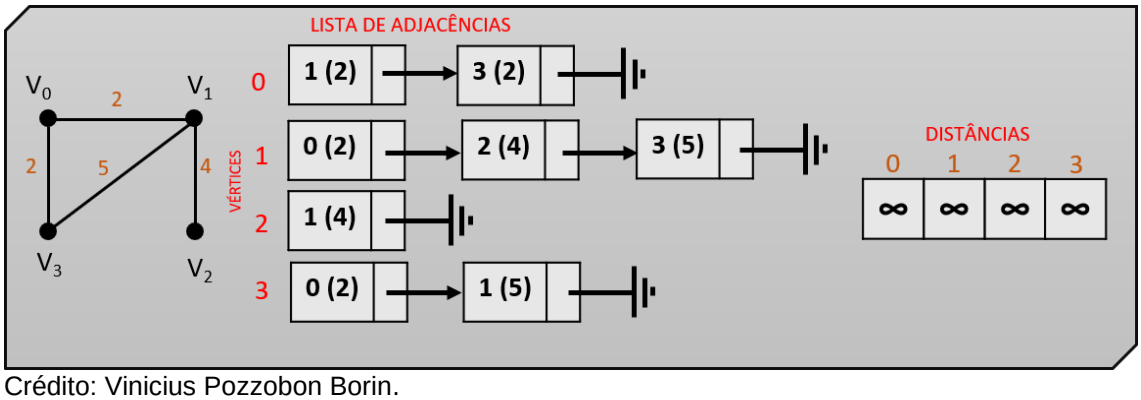
Vamos aplicar o algoritmo de Dijkstra em um grafo ponderado e obter as menores rotas, partindo de uma origem para todos os outros vértices desse grafo. Todo o algoritmo será apresentado utilizando uma **métrica aditiva**. Isso significa que essa métrica vai encontrar a menor rota considerando o **menor peso somado entre os caminhos**.

Um exemplo prático de métrica aditiva seria o tráfego de veículos em rodovias. Enquanto maior o tráfego, pior o desempenho daquela aresta, e, portanto, maior o seu peso no grafo.

Para conhecimento, existem outros tipos de métricas, como a multiplicativa, em que o melhor caminho encontrado é aquele cujo produto entre os caminhos é o maior valor, e os pesos são multiplicativos nessa situação. Um exemplo dessa métrica poderia ser o limite de velocidade das estradas: enquanto maior o limite, mais rápido o veículo anda e, portanto, melhor aquela rota/aresta.

Na Figura 32, a seguir, temos 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 32 – Algoritmo de *Dijkstra*, partindo de V_1: estado inicial



Crédito: Vinicius Pozzobon Borin.

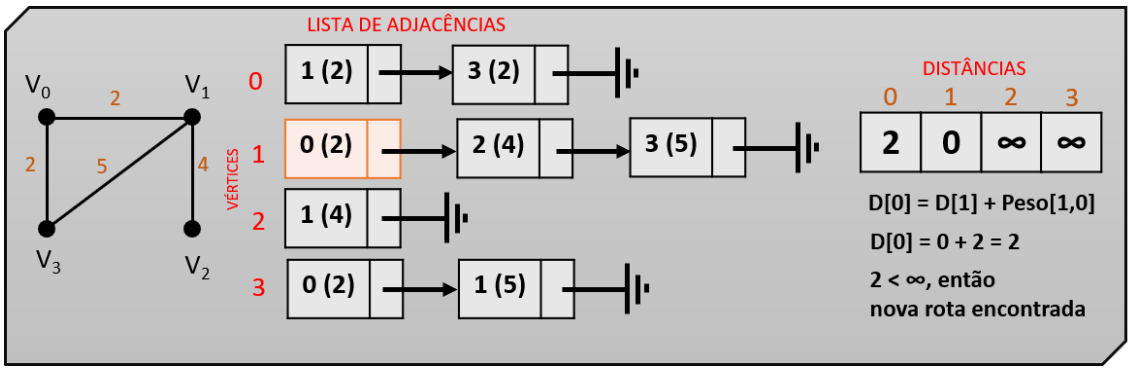
O algoritmo do caminho mínimo precisa calcular as menores rotas de um vértice em relação a todos os outros. Portanto, um vetor com distâncias fica armazenado. O estado inicial desse vetor é que todas as rotas iniciam com um valor infinito, ou seja, como se o caminho de um vértice até o outro não fosse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

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

Na Figura 33, a seguir, iniciamos a primeira etapa de operação do método. Como partiremos de V_1, já iniciaremos acessando a lista de vizinhos desse vértice e calculando as rotas para eles. Iniciamos encontrando 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 está apresentado no canto inferior direito da Figura 33.

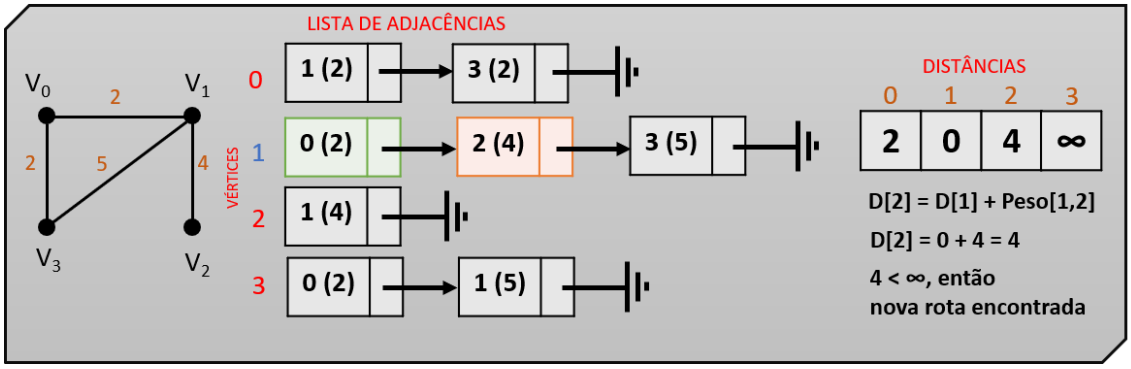
Figura 33 – Algoritmo de *Dijkstra*, partindo de V_1: etapa 1



Crédito: Vinicius Pozzobon Borin.

Seguimos na Figura 34, a seguir, 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.

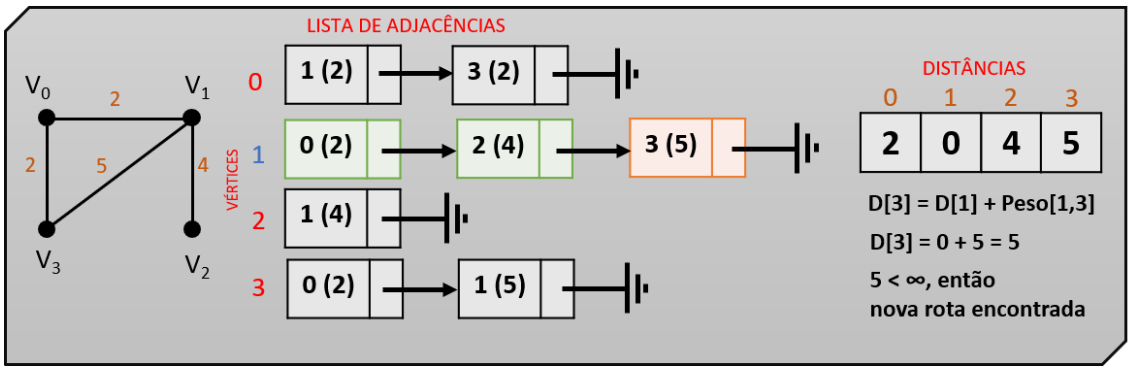
Figura 34 – Algoritmo de *Dijkstra*, partindo de V_1: etapa 2



Crédito: o autor.

A etapa 3 está na Figura 35, a seguir. De forma semelhante às duas etapas anteriores, calculamos a distância entre V_1 e V_3, resultando em 5, e colocamos no vetor de distâncias na posição de V_3.

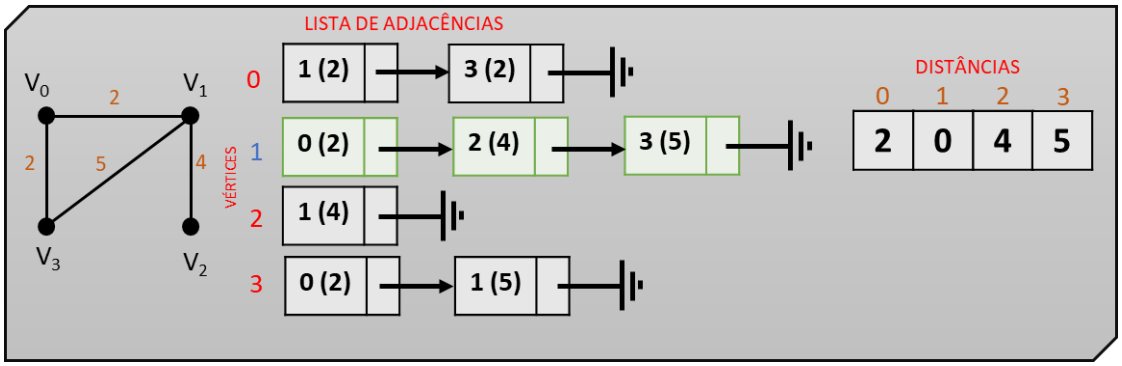
Figura 35 – Algoritmo de *Dijkstra*, partindo de V_1: etapa 3



Crédito: Vinicius Pozzobon Borin.

Todos os vizinhos do vértice V_1 têm suas rotas calculadas. Um panorama geral é apresentado na Figura 36, a seguir, indicando que todos os vizinhos foram calculados (cor verde).

Figura 36 – Algoritmo de *Dijkstra*, partindo de V_1: etapa 4

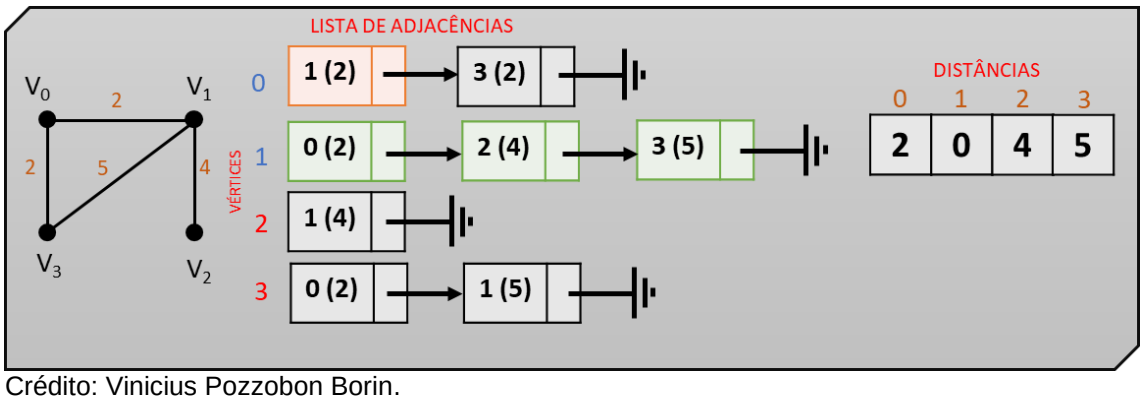


Crédito: Vinicius Pozzobon Borin.

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

Seu primeiro vizinho é o próprio vértice de origem V_0, na Figura 37, a seguir. Como esse vértice já foi visitado, podemos pulá-lo nessa análise.

Figura 37 – Algoritmo de *Dijkstra*, partindo de V_1: etapa 5

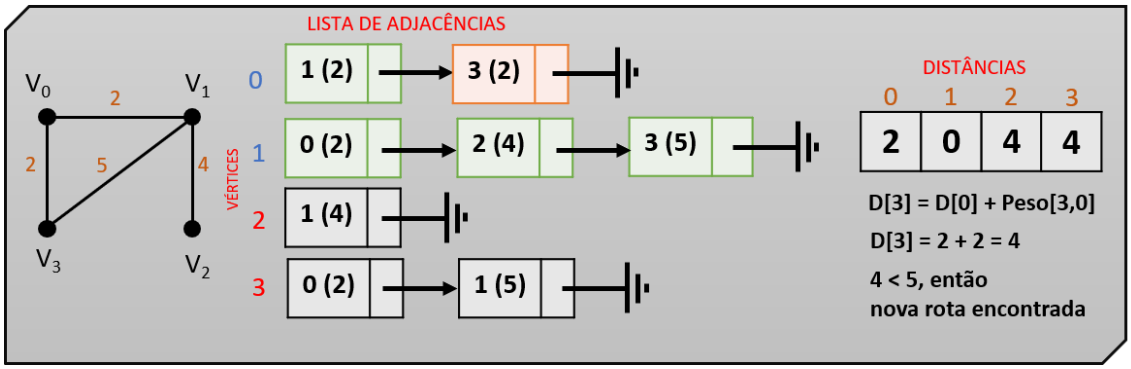


Crédito: Vinicius Pozzobon Borin.

Na Figura 38 adiante, 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 com o peso da aresta V_0 e V_3, que é 2 também. Assim $2 + 2 = 4$.

Observemos agora algo interessante. O peso da rota de V_1 até V_3, passando por V_0, resultou em peso 4. Anteriormente, tínhamos calculado o peso da rota direta entre V_1 e V_3, cujo valor resultou em 5. Portanto, a rota passando por V_0 tem um peso menor ($4 < 5$), resultando em uma nova rota até V_3. Notemos que uma rota com peso menor, mesmo passando por um vértice a mais, acaba resultando em um caminho menor que o outro.

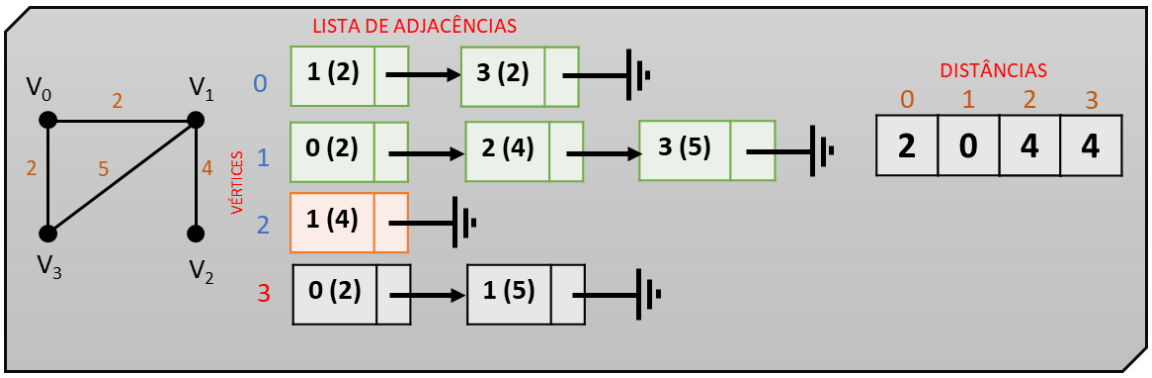
Figura 38 – Algoritmo de *Dijkstra*, partindo de V_1: etapa 6



Crédito: Vinicius Pozzobon Borin.

Na Figura 39 adiante, já encerramos nossos cálculos passando pelo vértice V_0. Seguimos então para o próximo vértice não visitado e de menor distância no vetor, o vértice V_2. O único vizinho de V_2 é vértice origem V_1, já visitado. Portanto, não existirá uma nova rota aqui.

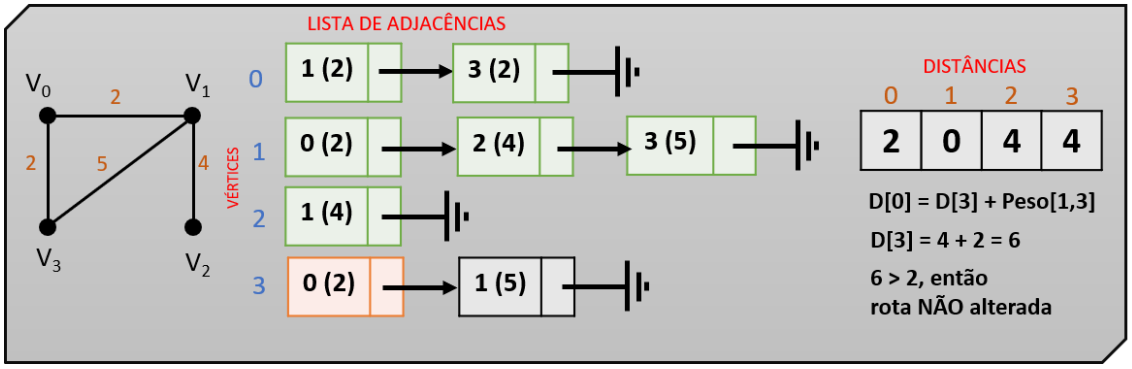
Figura 39 – Algoritmo de *Dijkstra*, partindo de V_1: etapa 7



Crédito: Vinicius Pozzobon Borin.

Na Figura 40 adiante, 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, custo superior a atualmente a rota de 4.

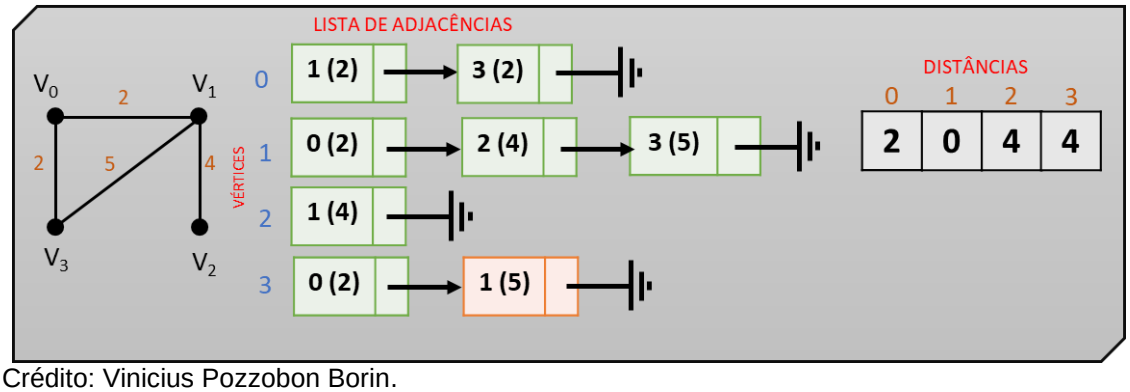
Figura 40 – Algoritmo de *Dijkstra*, partindo de V_1: etapa 8



Crédito: Vinicius Pozzobon Borin.

Na Figura 41, a seguir, temos o segundo vizinho de V_3, o V_1, que é a própria origem. Portanto, não precisamos analisar.

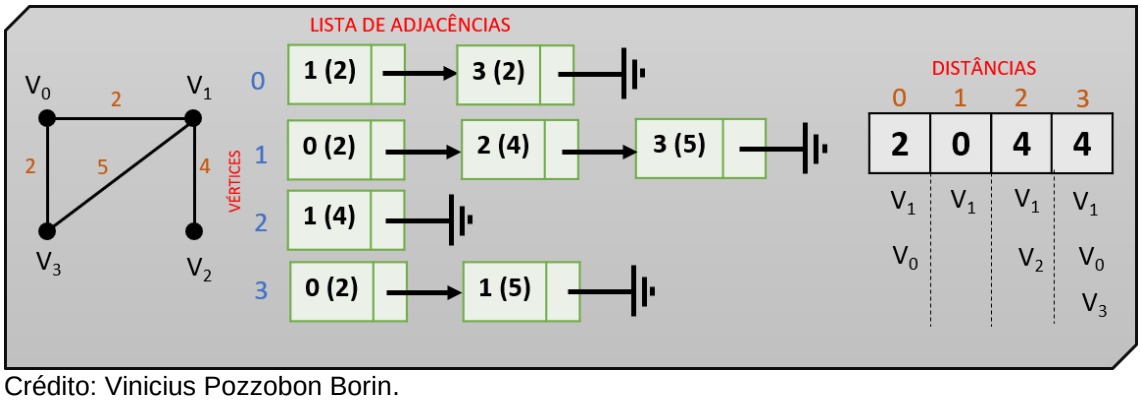
Figura 41 – Algoritmo de *Dijkstra*, partindo de V_1: etapa 9



Crédito: Vinicius Pozzobon Borin.

Na Figura 42, a seguir, já percorremos todos os vizinhos de todos os vértices e calculamos todas as rotas possíveis. Desse modo, as distâncias resultantes estão colocadas 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 42 – Algoritmo de *Dijkstra*, partindo de V_1: etapa 10



Crédito: Vinicius Pozzobon Borin.

FINALIZANDO

Nesta etapa, aprendemos sobre estrutura de dados do tipo grafo. Aprendemos que grafos não apresentam uma estrutura fixa na sua topologia de construção e são constituídos de vértices e arestas que conectam esses vértices.

Cada vértice do grafo conterá um conjunto de vértices vizinhos, os quais são vértices conectados por meio de uma única aresta. Aprendemos que podemos representar um grafo de duas maneiras distintas: 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, vimos o algoritmo de busca por largura (BFS) e o de busca por profundidade (DFS).

Por fim, vimos um algoritmo clássico de cálculo de rotas dentro de um grafo. Ele calcula o menor trajeto dentro de um grafo pondero. O algoritmo estudado foi o de *Dijkstra*.

REFERÊNCIAS

ASCENCIO, A. F. G.; ARAÚJO, G. S. de. **Estruturas de dados**: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson Prentice Halt 3, 2010.

BHARGAVA, A. Y. **Entendendo algoritmos**. São Paulo: Novatec, 2017.

CORMEN, T. H. **Algoritmos**: teoria e prática. 3. ed. São Paulo: Elsevier, 2012.

DROZDEK, A. **Estrutura de dados e algoritmos em C++**. Tradução da 4ª edição norte-americana. São Paulo: Cengage Learning Brasil, 2018.

FERRARI, R. et al. **Estruturas de dados com jogos**. São Paulo: Elsevier, 2014.

KOFFMAN, E. B.; WOLFGANG, P. A. T. **Objetos, abstração, estrutura de dados e projeto usando C++**. Porto Alegre: Grupo GEN, 2008.