
ESTRUTURAS AVANÇADAS DE DADOS
Uma introdução
Sandro José Rigo
Escola Politécnica
COLEÇÃO EAD
EDITORA UNISINOS

ESTRUTURAS AVANÇADAS DE DADOS

[UMA INTRODUÇÃO]

SANDRO JOSÉ RIGO

EDITORA UNISINOS
2011

APRESENTAÇÃO

Caro leitor, neste livro você encontrará um guia para aspectos introdutórios da área de Estruturas de Dados, organizado de modo a lhe servir como apoio para o estudo e compreensão de recursos de avaliação de algoritmos e utilização de estruturas avançadas de dados que lhe possibilitarão atender de forma adequada aos desafios de problemas complexos na área de desenvolvimento de programas de computador.

Este livro é parte do material da disciplina Estruturas Avançadas de Dados, do curso de Sistemas de Informação, sendo que devem ser utilizados os livros das disciplinas de Programação I e II como consulta para o embasamento prévio necessário.

Na parte inicial do livro serão apresentadas noções gerais sobre análise de algoritmos, relacionando a sua necessidade, formas adequadas para seu encaminhamento e exemplos de utilização. Com base nessas técnicas, serão propostas diversas análises ao longo do livro, para as diferentes estruturas de dados tratadas, de forma a proporcionar uma ampla aplicação destes recursos e facilitar a sua compreensão e fixação.

Tendo como base esse arcabouço relacionado com a análise de algoritmos, serão apresentadas diversas estruturas de dados, sempre a partir de contextos adequados para sua utilização e exemplos práticos que sejam capazes de ilustrar suas possibilidades, vantagens e limitações.

Inicialmente, serão estudadas as tabelas HASH e as estruturas HEAP, seguidas de noções gerais de estruturas TRIES e B-TREES. A seguir, serão descritas noções gerais de GRAFOS e uma série de aplicações dessas estruturas, tais como ordenação topológica, cálculo de amplitude mínima e identificação de caminhos mais curtos.

Para cada uma das estruturas são apresentados os seus conceitos gerais, formas de utilização e principais operações. Em especial, são demonstradas as suas possibilidades de aplicação em situações reais e avaliadas as suas características em situações de resolução de problemas, fomentando a aplicação das técnicas para análise de algoritmos tratadas no início do livro.

SUMÁRIO

Capítulo 1

INTRODUÇÃO

Este capítulo possui como objetivo proporcionar uma revisão de alguns conceitos que serão importantes para o melhor aproveitamento do texto. Nele você encontrará um panorama geral envolvendo aspectos de linguagens de programação e do estudo de algoritmos, bem como de estruturas de dados. Além desses aspectos, são propostos questionamentos de forma a estimular análises que visam identificar a necessidade de técnicas aqui estudadas e sua adequação para diversos contextos de uso. Utilize este capítulo como um ponto de início para despertar a sua curiosidade sobre os diversos aspectos relacionados com a Análise de Algoritmos e o uso de Estruturas de Dados.

O desenvolvimento de software é uma das áreas nas quais se observa um enorme crescimento, que pode ser associado com a evolução de recursos tecnológicos diversos. Alguns destes são os avanços na microeletrônica, que possibilitaram a construção de componentes miniaturizados cada vez mais poderosos, outros são os desenvolvimentos nas áreas de telecomunicação e redes de computadores ou as melhorias de tecnologias de digitalização de dados. Com esses recursos, tornou-se possível armazenar, processar e transmitir eficientemente uma quantidade enorme de dados. Os dispositivos para essas atividades tornam-se cada vez mais versáteis e miniaturizados, podendo, em muitos casos, ser utilizados de forma bastante flexível, em diversos dispositivos e não apenas em computadores de mesa.

A disponibilidade dessa infraestrutura para o armazenamento, transmissão e processamento de dados veio fomentar diversos tipos de necessidades, que agora encontram viabilidade prática. Desta forma, atividades que antes eram consideradas inviáveis por demandarem grande capacidade de processamento ou memória tornam-se realidade, tais como a síntese de voz, animações e simulações complexas por computador, manutenção de bases de dados extensas, processamento de imagens, integração de dados, entre outras.

Além disso, áreas que não eram consideradas aptas para uso de recursos de informática são agora possíveis de serem atendidas, com produtos customizados e descritos para sua automação. Exemplos são softwares de automação comercial, de apoio a vendas ou outros processos, softwares para gestão acadêmica, automatização de laboratórios, controle de usinas, monitoramento e controle de veículos ou de equipamentos médicos, entre tantos outros existentes.

Por fim, destaca-se que novas fronteiras foram delineadas, emergindo justamente da grande expansão na utilização de sistemas de informação, processamento e geração

de enormes volumes de dados. Áreas como Mineração de Dados, que utilizam bases de dados ou conjuntos de dados não estruturadas para a descoberta de padrões e informações úteis, são um exemplo bastante característico, relacionado não apenas com os volumes de dados gerados com os sistemas de automação comercial e industrial, mas também com a ampla utilização destes e de outros sistemas, como no caso de aplicações disponíveis na *web*, onde um enorme volume de dados de interação é gerado pelos usuários, contendo informações valiosas sobre suas intenções e interesses.

Neste breve contexto, pode-se destacar também a grande utilização de diferentes linguagens de programação, adequadas às tecnologias disponíveis em cada momento, algumas com ciclo de vida mais longo, outras nem tanto. As diferentes metodologias para a construção e manutenção de sistemas de informação, que constituem uma área em constante evolução, também representam um fator a ser destacado.

Desta forma, a implementação de sistemas que irão resolver problemas das diversas áreas é sempre realizada a partir de uma metodologia e de uma ou mais linguagens de programação. Certamente a carreira de um profissional de desenvolvimento de sistemas contará com alguns eventos de estudo de novas linguagens de programação, para adequação a novas necessidades ou a um contexto ou empresa específicos.

Este aprendizado e utilização de novas linguagens é tanto mais fácil quanto mais sólidos forem os conhecimentos do profissional em áreas como lógica de programação, descrição e avaliação de algoritmos, bem como na aplicação prática de estruturas de dados.

Portanto, você está diante de um cenário onde existe a possibilidade de manutenção e processamento de uma grande quantidade de dados, o que permite o atendimento de diversas aplicações, algumas destas sequer imaginadas até recentemente. As alternativas para desenvolvimento do programas de computador também são bastante variadas e evoluem constantemente. Esses fatores nos levam a enfatizar aqui a necessidade de uma ampla compreensão e domínio de conceitos que formam uma base para qualquer uma destas iniciativas de desenvolvimento. Para isso, você encontrará a seguir um breve resumo de alguns destes conceitos.

1.1 Algoritmos

De modo informal, pode-se considerar um algoritmo como uma sequência de passos para a resolução de um problema. Normalmente o problema sendo tratado é descrito em termos de um conjunto de dados, estruturados de forma a facilitar o seu tratamento. Existe uma associação importante entre o algoritmo e a estrutura de dados utilizada, normalmente com o objetivo de otimizar os aspectos de utilização de memória e de realização de operações. Na literatura relacionada com programação e

desenvolvimento de software é comum a utilização dos termos “entrada de dados” e “saída de dados”. O primeiro corresponde ao conjunto de dados originais (a “entrada”) que será processado segundo os passos descritos no algoritmo. Já o segundo é o conjunto de dados resultantes desse processamento, ou seja, os valores desejados.

Portanto, em um primeiro passo desta revisão, um algoritmo pode ser considerado como a descrição dos procedimentos computacionais para a transformação de um conjunto de valores iniciais (ou seja, a entrada de dados) em um determinado conjunto de valores considerados como os resultados, ou seja, as saídas de dados. A descrição do algoritmo pode ser feita de diversas formas e pode, inicialmente, ser bastante abstrata, sem o detalhamento necessário para sua implementação. Porém, em geral, a descrição do algoritmo é feita de modo detalhado, inclusive com a especificação dos dados tratados e do formato utilizado para tal.

Exemplos bastante corriqueiros, que podem ser trazidos aqui para ilustrar problemas atendidos por algoritmos, são os procedimentos de ordenação de valores, pesquisa de conteúdo ou cálculo de resultados. Todos esses procedimentos fazem parte do cotidiano do uso de recursos computacionais e estão disponíveis e incorporados tanto em sistemas operacionais como em aplicativos diversos.

Os algoritmos podem ser desenvolvidos para situações bastante específicas e típicas de apenas um contexto de aplicação. Por exemplo, você poderia imaginar um algoritmo para determinar a melhor alternativa em uma logística de transporte de determinados produtos entre estados. Ou então, um procedimento para otimizar a alocação de tripulantes em voos de uma companhia aérea. Nesses casos, é necessária uma análise específica para determinar todos os detalhes e requisitos envolvidos, que serão as bases para a definição da solução desejada.

Porém, durante o desenvolvimento de algoritmos para situações específicas, como essas exemplificadas, podem ser detectadas operações parciais, necessárias para a solução geral, mas que se repetem, seja em diversas etapas da solução para uma dessas situações, seja em situações diferentes. Esse é outro aspecto interessante dos algoritmos, relacionado não com situações específicas, mas com operações comuns a diversos contextos. Por exemplo, tanto na solução de logística para o transporte de cargas, como na organização da alocação dos tripulantes da companhia aérea, pode ser necessário que um conjunto de dados seja ordenado. No primeiro caso, ele pode representar a lista de alternativas para uma rota parcial de transporte. No segundo caso, pode indicar a lista de tripulantes aptos à seleção para um determinado voo.

Portanto, ao estudar algoritmos, é importante tratar desses dois aspectos. Nas situações em que uma análise e soluções particulares são necessárias, o desenvolvedor irá utilizar sua criatividade e capacidade de proposição de soluções a partir dos requisitos específicos identificados. Porém, essa mesma tarefa pode ser facilitada caso o desenvolvedor possua conhecimento sobre recursos já desenvolvidos para as diferentes classes de problemas que são comuns a diversos contextos. A habilidade de

pesquisa e identificação de algoritmos já desenvolvidos, que podem ser aplicados ou adaptados aos contextos específicos, é muito valiosa e permite que resultados mais consistentes, otimizados e econômicos sejam alcançados.

Um algoritmo é considerado correto quando ele resolve o problema da forma esperada para todas as situações de aplicação. Ou seja, quando ele fornece os dados de saída de forma correta, para todos os dados de entrada. A validação dos algoritmos para essa situação pode ser feita de diversas formas, dependendo dos conjuntos de entrada tratados. Nas situações em que o algoritmo não consegue resolver corretamente as situações em sua totalidade, normalmente ele é descartado, mas podem haver situações em que taxas baixas de erros, encontradas em algoritmos eficientes, podem ser consideradas aceitáveis.

1.2 Orientação a objetos

Alguns paradigmas de programação facilitam tarefas como o reuso de código e a adaptação de procedimentos já disponíveis, tais como exemplificado nas situações onde uma solução geral pode ser construída com alguns blocos ou etapas já previamente disponibilizadas. Na programação orientada a objetos, por exemplo, existem alguns conceitos que são bastante adequados a essas situações. A seguir, serão comentadas brevemente essas características. Ressalta-se que o objetivo não é realizar uma exposição aprofundada de programação orientada a objetos, pois considera-se que esses conceitos já foram trabalhados e praticados pelo leitor. Mas o objetivo é relacionar os conceitos com as necessidades e possibilidades da descrição e implementação de algoritmos e estruturas de dados.

As origens da programação orientada a objetos estão associadas justamente com necessidades de reaproveitamento de esforços já realizados por equipes de desenvolvimento de software. Assim, um de seus pontos básicos é o desenvolvimento de unidades lógicas de programação que podem ser facilmente reutilizadas, sem necessidade de alteração ou envolvimento em detalhes internos de seu funcionamento. Essa característica permite a construção de soluções para problemas complexos com base em componentes mais simples, que atuam em partes específicas do problema. Por exemplo, no caso em que a resolução de uma situação mais abrangente envolver a necessidade de um procedimento de ordenação, pode ser pesquisada a existência deste recurso específico e quando existir um número de alternativas, estas podem ser comparadas para verificação da mais adequada. O recurso adequado é então incorporado ao sistema em desenvolvimento como um componente isolado, sem necessidade de ajustes ou envolvimento em detalhes internos de seu funcionamento.

Essa característica é desenvolvida em linguagens de programação orientadas a objetos a partir de conceitos, tais como “objetos”, que podem ser utilizados para integrar os seguintes aspectos: a representação de dados específicos, a realização de

operações sobre estes dados e o estabelecimento de mecanismos de comunicação para o acesso e alteração dos mesmos. Assim, um programa orientado a objetos pode ser entendido como uma coleção de objetos que colaboram entre si para a resolução do problema tratado. Este recurso permite a abstração de partes de um determinado algoritmo, que podem ser implementadas por diversos objetos interdependentes. Também permite que etapas de algoritmos que podem ser reutilizadas de forma mais abrangente, em outros contextos, sejam isoladas e assim compartilhadas.

Os tipos abstratos de dados, em programação orientada a objetos, representam um conceito importante para essa situação, pois permitem descrever um conjunto de dados e operações sobre estes, de modo a gerar uma unidade que pode ser compartilhada com outros componentes em um programa. Um exemplo de tipo abstrato de dados poderia ser imaginado para a descrição de uma imagem, sendo que, além da representação interna dos dados desta imagem, seriam descritos para ela as operações mais simples, tais como abertura do arquivo de imagem, exibição da imagem, obtenção do tipo de imagem, verificação do tamanho, entre outras operações.

Outro conceito importante e relacionado é o conceito de classe. Uma classe, em programação orientada a objetos, permite a descrição dos dados (chamados de "atributos") e dos procedimentos computacionais (indicados como "métodos") que serão associados aos objetos. Portanto uma classe possui os requisitos necessários para a implementação dos tipos abstratos de dados. No contexto da implementação de algoritmos, as classes podem ser utilizadas para a descrição das estruturas de dados e das operações necessárias para etapas específicas desses algoritmos. Como a descrição de classes envolve um protocolo bem definido, isso facilita a reutilização de classes já desenvolvidas, contendo soluções necessárias para os sistemas em desenvolvimento.

A utilização do mecanismo de herança existente nas linguagens de programação orientada a objetos permite que classes já existentes sejam estendidas em novas classes, com o aproveitamento de seus atributos e métodos. Desta forma, uma determinada classe que realiza, por exemplo, um processo de ordenação para dados numéricos, pode ser utilizada como base para uma nova classe que deve realizar esta mesma ordenação em um conjunto de dados similar ou com algumas alterações e acréscimos no procedimento. Esse mecanismo possibilita que o desenvolvedor estenda algoritmos já implementados e testados, para que sejam acrescidas características necessárias em novos contextos de aplicação.

1.3 Estruturas de dados

Todo o desenvolvimento de algoritmos está associado com o conjunto de dados a ser manipulado ou gerado, sendo que este pode ser um fator determinante para o resultado observado quando da execução do algoritmo. Processos de divisão de problemas complexos em problemas menores podem envolver etapas de adequação

dos dados para que algumas operações sejam realizadas de forma mais eficiente. Ou, de modo inverso, também podem ser tratados os dados para que sejam preservados recursos eventualmente escassos, tais como a memória.

Em geral, as diferentes estruturas de dados são descritas com base na sua organização e utilização de memória, sendo que algumas operações serão mais adequadas ou eficientes, de acordo com a estrutura de dados utilizada. Uma habilidade importante para o desenvolvimento de programas de computador está relacionada justamente com a descrição ou escolha de uma estrutura de dados que seja eficiente para as tarefas a serem realizadas. Por exemplo, a listagem de uma sequência de números pode ser feita com base em uma varredura de posições contíguas de memória, com uso de uma estrutura de dados conhecida como *lista*. Já em um caso onde exista a necessidade do relacionamento de informações de localização, tal como em um mapa descrevendo cidades e as diversas conexões entre as mesmas, a estrutura em lista não será adequada, pois cada elemento (cidade) estará relacionado com diversos outros e não apenas com dois elementos como na lista de números. Portanto, no segundo caso, uma estrutura em forma de grafo seria mais adequada e permitiria operações de forma mais eficiente.

Outra situação bastante interessante é a existência de relações funcionais importantes de acordo com a estrutura de dados empregada. Em situações como o controle de processos ou requisições é comumente empregado o formato de filas ou pilhas como estruturas de dados para a manutenção das informações tratadas. Neste caso, a estrutura de dados está associada com o próprio formato do processamento e, portanto, com o projeto do algoritmo para atender ao problema. O formato no qual estas operações são realizadas está associado e define o formato da própria estrutura de dados em memória. No caso de uma pilha, por exemplo, existem duas operações padronizadas: uma para o armazenamento de dados e outra para a recuperação de dados. O armazenamento é sempre feito no topo da pilha, bem como a recuperação de dados. Portanto este procedimento define que o último dado recebido será sempre o próximo a ser processado. Já no caso de uma estrutura conhecida como fila, o armazenamento é sempre realizado no final da fila e a recuperação é feita no início da fila. Portanto o procedimento define que sempre o primeiro elemento armazenado será o primeiro a ser atendido e assim sucessivamente.

Em determinadas situações é possível prever exatamente a quantidade de dados que será processada, sendo que deste modo a definição de um problema e de sua solução computacional pode ser feita com base em estruturas de dados de tamanho fixo, ou estáticas. Porém, em grande parte das situações, é bastante difícil que a informação sobre quantidade de dados esteja disponível previamente. Além disso, a utilização de recursos de memória pode não ser tão eficiente no caso de alocação, de forma estática, da maior quantidade necessária, para aqueles casos em que apenas parte desta quantidade será efetivamente utilizada. Nestes casos são utilizados recursos para facilitar e tornar flexível a programação de soluções, conhecidos como

alocação dinâmica de memória. Utilizam-se operações de alocação de quantidades específicas de memória apenas durante o período em que se fazem necessárias para a solução dos problemas, sendo utilizados como apoio os recursos conhecidos como ponteiros, que permitem o tratamento de dados em memória de uma forma muito flexível.

Esta característica de utilização de memória de forma dinâmica pode ser aproveitada por procedimentos implementados com técnicas de recursividade. Com estas técnicas, um mesmo procedimento pode ser repetidamente aplicado em conjuntos diversos de dados, integrando diversos resultados parciais para a efetivação de uma solução de problemas com natureza dinâmica.

1.4 Conclusões

Nesta breve revisão foram destacados aspectos preliminares importantes para o desenvolvimento de soluções computacionais para os problemas que serão encontrados em diversas situações. Mesmo com o cenário atual indicando a melhoria contínua dos recursos de hardware e comunicação, que nos indicam o desenvolvimento de processadores cada vez mais velozes e capacidades de armazenamento cada vez maiores, existe a necessidade de avaliação e análise dos algoritmos utilizados, dado que os problemas a serem resolvidos tornam-se também cada vez mais complexos e demandam maiores quantidades de dados a serem tratadas de forma eficiente.

Alguns paradigmas de desenvolvimento, como programação orientada a objetos, são adequados para o desenvolvimento de algoritmos, por ampliarem algumas características como o reuso, abstração e encapsulamento. A utilização de algoritmos de forma eficiente está intimamente ligada com a descrição e escolha de estruturas de dados adequadas para cada situação tratada. Essas estruturas de dados podem atuar no sentido de melhoria de utilização de recursos, como memória, mas também podem estar associadas à eficiência dos algoritmos, quando permitem que as operações necessárias sejam realizadas de forma otimizada.

Os problemas sendo tratados podem envolver situações com ampla variação de demandas. Um profissional na área de desenvolvimento de software pode deparar-se com a necessidade de tratar situações como a identificação do maior valor inteiro em uma sequência de valores, a ordenação alfabética de uma sequência de palavras ou ainda a identificação de uma palavra dentro de um texto. Ou então, pode deparar-se com a necessidade de geração de roteiros eficientes entre diversas cidades, para atender a uma determinada necessidade de rota turística. Pode ainda necessitar definir qual o melhor fornecedor de determinado produto, de acordo com um conjunto variável de requisitos.

O desenvolvimento da indústria e a otimização de processos logísticos necessita da resolução de situações bastante dinâmicas e envolvendo grande quantidade de

variáveis. Áreas como a Biologia, Química ou Física apresentam diversas situações nas quais são necessárias análises e simulações de dados que possuem como requisitos tanto a quantidade de dados a serem tratados como a natureza complexa deste tratamento. A segurança de operações cada vez mais comuns em comércio eletrônico envolve diversos procedimentos, desde a geração de códigos de verificação até a criptografia de conjuntos de dados. Estas breves citações, dentre tantas outras existentes, servem para identificar a amplitude do campo de aplicação de algoritmos.

Portanto, o conhecimento de recursos de algoritmos, seu projeto, escolha e análise são de fundamental importância para o desenvolvimento de software eficiente e adequado. No texto a seguir serão apresentados ao leitor alguns destes recursos e será privilegiada a análise de sua aplicação em situações reais.

Capítulo 2

ANÁLISE DE ALGORITMOS

Este capítulo possui como objetivo introduzir os fatores envolvidos nas atividades de análise de algoritmos. Sabe-se que os diversos problemas computacionais envolvidos no desenvolvimento de aplicações podem ser implementados de forma a envolver um maior ou menor uso de recursos como memória e processamento. Mesmo com processadores velozes e grande capacidade de memória, a eficiência dos algoritmos é fundamental para a viabilização de aplicações em diversas áreas.

Tendo em vista as diversas possibilidades de solução possíveis para os problemas computacionais atuais, levando em conta a necessidade de performance e otimização que existe na maioria dos casos, podemos nos questionar quanto ao modo como os algoritmos podem ser comparados entre si e analisados. Em primeiro lugar, devemos ressaltar que os objetivos da análise de algoritmos são prover ferramentas para que o desenvolvedor possa comparar soluções, com base em critérios objetivos, que lhe permitam, por exemplo, indicar versões mais eficientes ou versões com diferentes necessidades de recursos de armazenamento.

Portanto a análise de algoritmos desenvolve-se de modo a permitir que sejam feitas previsões realistas acerca dos recursos necessários para a execução dos procedimentos descritos no algoritmo. Normalmente são grandes as diferenças entre os conjuntos de dados de testes utilizados na análise e desenvolvimento de soluções para os problemas e os dados reais, que deverão ser processados a partir da implementação das soluções. Portanto a identificação antecipada dos requisitos para o tratamento dos volumes reais de dados é fundamental para garantir a viabilidade das operações.

Vistas de outra forma, essas colocações implicam na utilização de mecanismos que tratem o processo de análise de forma consistente, gerando resultados confiáveis e que possam ser projetados para indicar o resultado que será obtido em diversos cenários. Também podem ser levadas em conta as características específicas dos ambientes nos quais os algoritmos serão executados, tais como no caso de algoritmos desenvolvidos para execução em ambientes de processamento paralelo ou com a utilização de recursos especiais para parte do processamento.

Em geral, são analisados dois aspectos principais de algoritmos, que consistem no espaço de armazenamento utilizado e na capacidade de processamento necessária. Ou seja, o tamanho de memória e o tempo de execução são os focos principais das análises. Para esta introdução ao assunto, devemos assumir alguns aspectos gerais, que podem apresentar algum impacto no processo de análise. Assim, como forma de

introdução a este assunto, será considerada, neste texto, uma situação de execução de programas sem uso de recursos, tais como paralelismo ou outros modelos de computação. Ou seja, considera-se os programas sendo executados de forma sequencial, de acordo com a sequência de instruções gerada pelo procedimento de compilação. Também considera-se a utilização da memória principal do computador (a memória RAM) como base para o armazenamento dos dados a serem processados, bem como das instruções do programa.

Um aspecto que também pode ser envolvido na análise de algoritmos, mas que não será tratado neste texto, é a verificação da correção do algoritmo, ou seja, a confirmação de que o algoritmo resulta em respostas corretas para o processamento de qualquer um dos valores de seu conjunto de possíveis entradas de dados.

2.1 Análise do tempo de execução

Para encaminharmos um exemplo das possibilidades para análise de algoritmos no que diz respeito ao tempo de execução, faremos a seguir o detalhamento deste processo, com base em alguns exemplos. Inicialmente, considera-se que a análise do tempo de execução levará em conta um modelo computacional utilizando apenas um processador, no qual os dados e instruções estejam dispostos em memória principal e onde as instruções são executadas sequencialmente, de acordo com a sua ordem no programa. Serão desconsideradas as diferenças que podem ocorrer no tratamento de tipos básicos de dados, tais como valores inteiros ou de ponto flutuante, que podem impactar nesta análise quando consideram-se processadores com capacidades diferenciadas. Nos exemplos tratados, também não serão analisadas especificidades de linguagens de programação, sendo que os exemplos foram projetados para não colocarem restrições de conhecimento prévio da sintaxe de linguagens de programação para o seu entendimento. Apesar de ser utilizada, em alguns casos, a sintaxe similar a linguagem de programação Java, os exemplos devem possibilitar a sua análise sem a dependência de conhecimento prévio da linguagem de programação e sua sintaxe.

Realizando a análise com o encaminhamento proposto, é possível chegar a uma conclusão a respeito de uma estimativa de tempo de execução, com base em alguns aspectos do código, tais como o tipo de instruções e sua frequência. Veja na sequência de códigos da figura 1, a seguir, como pode ser encaminhado este tipo de análise. Observe que nas linhas 1 e 2 é feita a declaração de duas variáveis do tipo inteiro e ao mesmo tempo estão descritas duas operações de atribuição para estas variáveis. Na linha 3 existe uma operação de multiplicação e outra de atribuição. Na linha 4 observa-se uma operação de adição e uma atribuição. Este trecho de código poderia ser parte de um algoritmo a ser analisado, em relação ao seu tempo de execução.

```
1. int aux1 = 1;
2. int aux2 = 1;
3. aux1 = aux1 * aux2;
4. aux2 = aux2 + 1;
```

Figura 1 – Trecho de código fonte em análise.
Fonte: o autor.

A partir do código descrito na figura 1, podemos identificar o tempo de execução necessário, com base no conhecimento das diferenças entre os tempos de execução de cada uma das instruções. Assim é possível identificar estes tempos individualmente, associando a atribuição com o termo *op1*, multiplicação com *op2* e adição com *op3*, podemos chegar à conclusão de que o trecho de código terá como tempo de execução o seguinte valor: T = 4 op1 + 1 op2 + 1 op3. Veja na figura 2 o detalhamento que levou a essa equação, observando as colunas "Quantidade" e "Operações", nas quais estão descritas as operações envolvidas em cada linha de código e a sua frequência. Na linha 1, por exemplo, está descrita uma operação de atribuição (op1) enquanto que na linha 3 estão descritas uma operação de multiplicação (op2) e uma operação de atribuição (op1).

Código fonte	Quantidade	Operações
1. int aux1 = 1;	1	op1
2. int aux2 = 1;	1	op1
3. aux1 = aux1 * aux2;	1	op2 e op1
4. aux2 = aux2 + 1;	1	op3 e op1

Figura 2 – Avaliação do tempo de execução.
Fonte: o autor.

Nesse exemplo está colocada uma situação bastante simples, onde o código indicado consiste em um trecho de quatro instruções, sem nenhuma operação condicional ou laço de repetição. Mas essa situação exemplifica o processo básico utilizado em situações mais complexas, que são encontradas em algoritmos para resolução de problemas computacionais. A próxima figura ilustra uma situação na qual também existe um laço de repetição de operações, sendo que neste caso a estimativa do tempo de execução deste trecho de código precisa levar em conta o número de

vezes que o código será executado, o que é dado pela variável “n”, recebida como parâmetro e considerada, neste exemplo, como contendo valores positivos. Analisando o exemplo da figura 3, observa-se que este parâmetro (“n”), recebido na linha 1 do exemplo, indica o número de vezes que deve ser repetido o trecho de código que vai da linha 5 até a linha 9, ou seja, o trecho de repetição, com a instrução “while”. Esta instrução irá resultar no teste da condição indicada na linha 5 (teste para verificar se o valor da variável “i” é menor que o valor da variável “n”) e no caso de um resultado verdadeiro, todo o trecho das linhas 6, 7 e 8 será executado, sendo que após esta execução a instrução da linha 5 será novamente executada, podendo ser repetido todo o ciclo novamente, até que uma condição falsa seja encontrada. Como a variável “i” possui inicialização com valor 0 e está sendo incrementada na linha 8, este contexto define que a instrução da linha 5 será sempre executada “n+1” vezes. Já o trecho das linhas 6 até 8 será executado “n” vezes. As demais operações, antes e depois do trecho de repetição, nas linhas 2, 3, 4, e 10, serão executadas apenas uma vez.

Código fonte	Quantidade	Operações
1. public void teste(int n){		
2. int aux1 = 1;	1	op1
3. int aux2 = 1;	1	op1
4. int i = 0;	1	op1
5. while (i < n) {	n + 1	op4
6. aux1 = aux1 * aux2;	n	op1 e op2
7. aux2 = aux2 + 1;	n	op1 e op3
8. i = i + 1;	n	op1 e op3
9. }		
10. i = 0;	1	op1
11. }		

Figura 3 – Avaliação do tempo de execução com laço de repetição.
Fonte: o autor.

Tendo sido analisadas essas informações, podemos utilizar o mesmo procedimento para a análise do exemplo anterior para a definição de uma equação que indique o

tempo de execução. No caso anterior, como não havia nenhuma variável definindo repetições de trechos de instruções, a equação consistia simplesmente na soma de todas as instruções e do tempo de cada uma. No exemplo da figura 3, devemos considerar esta mesma lógica e acrescentar na equação uma variável que irá representar o parâmetro "n". Neste caso, o tempo de execução não é mais fixo como no exemplo anterior, mas é variável, dependente do valor fornecido pelo parâmetro "n". Assim, uma primeira aproximação para esta análise do tempo de execução poderia ser a seguinte : $T(n) = 4\ op1 + (n+1)\ op4 + 3n\ op1 + n\ op2 + 2n\ op3$. Veja que as instruções nas linhas 2, 3, 4 e10 estão representadas no primeiro termo (4 op1) que identifica a operação de atribuição (op1) executada por 4 vezes, uma em cada linha de código. As instruções da linha 5 estão representadas no segundo termo, ou seja, (n+1) op4. Desta forma, este termo representa a operação de comparação (op4) executada por n+1 vezes. Nas linhas 6, 7 e 8 ocorre uma instrução de atribuição (op1), representada pelo terceiro termo, ou seja, 3n op1. A operação de multiplicação (op2) da linha 6 está representada pelo quarto termo, ou seja, n op2. As operações de adição (op3) das linhas 7 e 8 estão representadas pelo último termo, ou seja, 2n op3. Neste caso, pode-se organizar os termos para deixar o valor do parâmetro "n" em evidência, tal como $T(n) = 4\ op1 + op4 + n(3op1 + op2 + 2op3 + op4)$.

O aspecto importante a ser destacado na segunda análise é que tanto a visualização do código com os respectivos tempos e quantidades de operações, como a equação descrevendo o tempo de execução, indicam que alguns dos componentes individuais possuem pouca expressão, quando comparados com os componentes dependentes do parâmetro "n", que deve ser considerado como a tendência mais adequada para a análise do tempo de execução.

Essa afirmação fica mais evidente ao analisarmos o que ocorre com os valores de "n" variando em uma faixa ampla. Por exemplo, no caso do valor de "n" ser igual a 1, os tempos das instruções individuais, relacionadas nas linhas 2, 3, 4 e 10 equivaleriam a 4 instruções, sendo que as instruções afetadas pelo valor de "n", que estão nas linhas 5, 6, 7 e 8, equivaleriam a 7 instruções. Ou seja, as instruções fora do laço equivaleriam a cerca de 36% do tempo de execução e as instruções dentro do laço equivaleriam a 67% do tempo de execução. Porém aumentando os valores de "n" percebe-se que essa relação modifica muito rapidamente. No caso de "n" com valor igual a 10, as instruções fora do laço continuariam a contabilizar 4 instruções, enquanto que as instruções dentro do laço consistiriam de 70 instruções. Ou seja, nesse caso, as instruções dentro do laço já equivaleriam a 94% do tempo de execução. Aumentando o valor de "n" para 100, o percentual das instruções dentro do laço passa a equivaler a 99,4% de todas as instruções.

Essa consideração nos leva a indicar que a análise dos algoritmos com esta abordagem pode servir para demonstrar uma tendência de crescimento, tanto do tempo de execução como do uso de memória. Mais do que isso, pode-se determinar quais os parâmetros que realmente são importantes a considerar. Neste caso, pode-se

estimar o crescimento do tempo de execução do algoritmo com base no valor de entrada (“n”) e nas operações que são afetadas diretamente por este valor, tais como as operações em laços de repetição, visto que fica evidente o impacto bastante baixo das demais operações.

Considerando-se mais um exemplo, no qual existem laços de repetição aninhados, a noção inicial pode ser aprofundada. Veja a seguir, na figura 4, um trecho de código no qual existem dois laços de repetição de comandos, utilizados para percorrer uma matriz com valores inteiros, somando todos estes valores em uma variável auxiliar. A análise de operações realizadas indica a ocorrência de uma operação de atribuição (op1) na linha 2 e uma operação de retorno de valores (op4) na linha 8. Na linha 3 encontra-se uma operação de controle de repetições, que será considerada de forma simplificada como op2 e para este caso ela irá repetir-se por n+1 vezes. Essa instrução apresenta a mesma situação do exemplo anterior, na figura 3, onde encontra-se uma ocorrência de uma laço de repetição de comandos. Já na linha 4 encontra-se uma segunda instrução de controle de laços, porém em situação diferenciada, pois a mesma ocorre dentro do laço anterior, determinado pela instrução da linha 3. Desta forma a instrução será executada por n+1 vezes, tal como a instrução da linha 3, dado que consistem no mesmo formato de controle de laços (op2). Entretanto, por encontrar-se dentro do laço controlado pela instrução da linha 3, este número de n+1 execuções deve ser multiplicado ainda por n, dado que será repetido cada uma das vezes em que o laço for executado, gerando, portanto, uma quantidade de n(n+1) vezes para sua execução. Seguindo o mesmo princípio de análise, a instrução da linha 5, por estar dentro do segundo laço, irá executar por um número de n^2 vezes.

	Código fonte	Quantidade	Operações
1.	public int somatorio(int n, int v[][]){		
2.	int soma = 0;	1	op1
3.	for(int l = 0; l<n; l++) {	n + 1	op2
4.	for(int c = 0; c<n; c++) {	n(n + 1)	op2
5.	soma = soma + v[l][c];	n^2	op1 e op3
6.	}		
7.	}		
8.	return (soma);	1	op4
9.	}		

Figura 4 – Avaliação do tempo de execução com laços de repetição aninhados.

Fonte: o autor.

Uma primeira análise do tempo de execução neste caso poderia ser $T(n) = op1 + op4 + (n+1)\ op2 + n(n+1)\ op2 + n^2\ (op1 + op3)$. O primeiro termo corresponde à instrução da linha 2 e o segundo corresponde à instrução da linha 8. Já o terceiro e quarto termos originam-se nas linhas de código 3 e 4, respectivamente. O último termo corresponde à linha de código de número 5. Nesse caso, ao organizarmos os termos evidenciando o parâmetro "n" obtemos $T(n) = n^2\ (op1 + op2 + op3) + 2n\ op2 + op1 + op2 + op4$. Nessa descrição evidencia-se o termo n^2 como base para estimativas em relação ao tempo de execução do código respectivo. Com a mesma abordagem indicada para o exemplo anterior é possível verificar que a maior parte das execuções estão associadas com o valor do parâmetro "n" e que a existência de instruções executadas segundo o quadrado deste valor (n^2) são determinantes quanto ao tempo total do algoritmo, podendo, inclusive, ser descartado na análise, alguns dos outros fatores.

A partir destes exemplos, identifica-se uma forma de análise de algoritmos que permite a identificação de trechos de código, seu tempo de execução individual e a quantidade de vezes em que ocorrem em execuções típicas do programa. Com base nesta análise, pode-se verificar que em determinadas situações o tempo de execução será constante, ou seja, não existe nenhuma variação no tempo necessário para a execução do algoritmo. Essa situação é ilustrada na figura 2, porém não representa a maioria dos casos de aplicação real de algoritmos, dado que não apresenta nenhuma variação no conjunto de dados de entrada. Ela pode ser verificada no exemplo da figura 3, onde um parâmetro é recebido e determina o número de vezes em que um trecho de código é repetido. Nessa situação existe uma variação do tempo de execução de acordo com o valor recebido, sendo que o tempo de execução pode ser estimado com base em $T(n) = c1 + n\ c2$, onde c1 e c2 representam valores constantes associados com os tempos individuais das instruções envolvidas e "n" representa o número recebido como parâmetro para o trecho de código. Considera-se a situação como sendo de variação linear no tempo de execução, pois o valor "n" determina diretamente o crescimento do tempo de execução. Por fim, no terceiro exemplo, indicado na figura 4, existe um aninhamento de laços de execução, sendo que o tempo de execução pode ser estimado da seguinte forma: $T(n) = c1\ n^2 + c2\ n + c3$. Esse tipo de variação no tempo de execução é descrito como variação quadrática, pois o tempo de execução do algoritmo depende do valor de "n", utilizado em laços com aninhamento.

Essas considerações devem ser analisadas como taxas de crescimento, associadas com a quantidade dos valores de entrada a serem processados. Uma das vantagens da utilização dessa noção de taxas de crescimento é que elas podem ser determinadas com base na análise das operações descritas pelo algoritmo, portanto representam de forma consistente a sua execução. Durante esta análise podem ser descartadas operações de pouca influência no tempo total de execução, tal como exemplificado anteriormente, de modo que são possíveis identificações de padrões de crescimento bastante objetivos.

Outra vantagem desta abordagem é a possibilidade de comparação e avaliação da taxa de crescimento, de forma bastante clara. Um algoritmo com tempo de execução variando de acordo com a equação $T(n) = 5n$, outro com equação $T(n) = 5n^2$ e um terceiro com equação $T(n) = 2^n$ apresentam perspectivas radicalmente diferentes, tal como pode ser observado no gráfico a seguir, na figura 5.

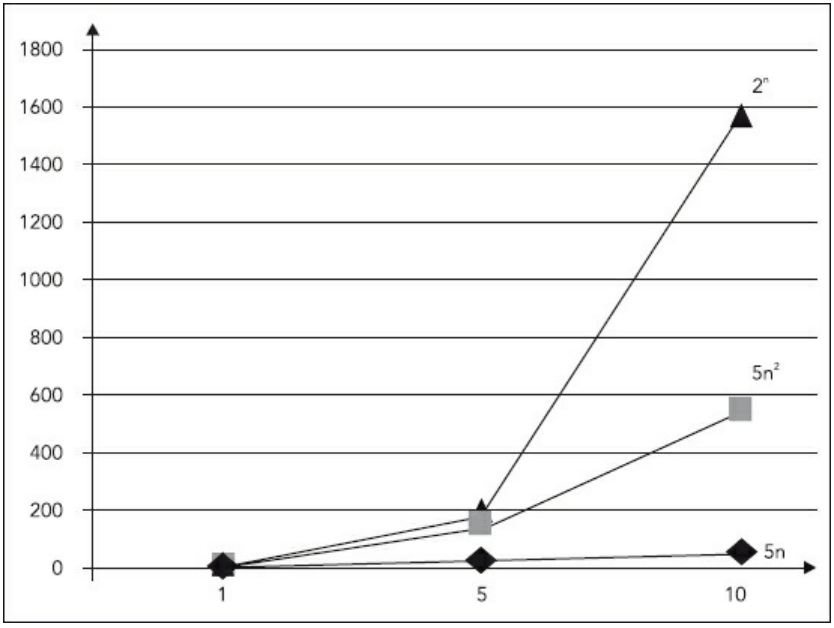


Figura 5 – Comparação de taxas de crescimento para tempos de execução.
Fonte: o autor.

Na figura 5 está identificado um gráfico contendo no eixo horizontal o número de entradas para um determinado algoritmo e no eixo vertical o tempo de execução associado. Ele permite visualizar a diferença enorme existente para os tempos de execução de acordo com taxas de crescimento linear ($T(n) = 5n$), quadrático ($T(n) = 5n^2$) e exponencial ($T(n) = 2^n$), mesmo com um pequeno número de valores de entrada. Observe na figura que mesmo com um pequeno número de entradas, igual a 10, já identifica-se grande variação entre tempos de execução, que seriam de 50 unidades para a taxa de crescimento linear, 500 unidades para o crescimento quadrático e 1.024 unidades para o crescimento exponencial. Porém essas diferenças podem

tornar-se bastante mais dramáticas quando utilizam-se valores maiores para os valores de entrada. Por exemplo, com as mesmas três taxas de crescimento, utilizando-se os valores de entrada indicados na tabela 1, que consistem em 10, 30 e 50, podem ser estimados os tempos de execução com base em segundos. Note que para a taxa de crescimento linear que consta na primeira linha, os valores variam apenas entre 0,05 segundos e 0,25 segundos. A taxa de crescimento quadrática, na segunda linha, varia entre 0,5 segundos e 12,5 segundos. Já a taxa de crescimento exponencial, descrita na terceira linha, inicia com 1024 segundos, passa para 298 horas e depois para 357 séculos. Essa comparação permite identificar situações em que é proibitivo o uso de determinados algoritmos.

Tabela 1 – Crescimento de tempos de execução

	10	30	50
$5n$	0,05	0,15	0,25
$5n^2$	0,5	4,5	12,5
2^n	1024	298 horas	357 séculos

Fonte: o autor.

Avançando um pouco mais na análise do tempo de execução de algoritmos, iremos agora avaliar as consequências da distribuição dos dados de entrada para os resultados finais obtidos. Existem algoritmos que possuem dependência apenas em relação à quantidade de dados, sem importar a sua distribuição ou valores. No caso de uma soma de valores de uma matriz, por exemplo, o tempo de execução do algoritmo para obtenção do resultado de duas matrizes diferentes, de mesmo tamanho, será exatamente o mesmo. As operações de adição a serem realizadas nesse caso são independentes da distribuição dos valores nas matrizes. Outros algoritmos possuem o mesmo comportamento.

O mesmo não ocorre em algumas outras classes de algoritmos. No caso de ordenação de valores, ou de busca sequencial de valores, por exemplo, a distribuição dos valores de entrada pode resultar em tempos de execução completamente diferenciados. Isso pode ser verificado de forma intuitiva, no caso de um algoritmo para a busca sequencial de palavras em um vetor. Existe a possibilidade da palavra desejada estar localizada na primeira posição do vetor, na última, ou na posição central. Caso o algoritmo percorra sequencialmente todas as posições do vetor até encontrar a palavra desejada, o tempo de execução para uma distribuição em que a palavra esteja na primeira posição será sempre o menor possível. Essa situação

também é conhecida como “melhor caso”, em contraposição ao “pior caso”, que seria ilustrado em nosso exemplo com a situação em que a palavra desejada está localizada na última posição do vetor. Esta situação se repete em casos de algoritmos de ordenação, o que também pode ser exemplificado e avaliado, informalmente, para o caso em que os dados de entrada apresentam-se parcialmente ordenados, situação em que diversas operações de movimentação destes dados não serão executadas pelo algoritmo.

Portanto, o tempo de execução dos algoritmos pode ser analisado em função dos valores de entrada, sendo que em uma primeira abordagem são descritas taxas de crescimento do tempo de execução, de acordo com o aumento do conjunto de entrada. A soma de valores em uma matriz de 10 linhas e 10 colunas é menor que a soma dos valores de uma matriz com 100 linhas e 100 colunas. Entretanto esta não é a única análise a ser feita, pois em alguns casos algoritmos podem apresentar tempos diferenciados de execução, de acordo com variações no conjunto de dados de entrada recebidos.

Usualmente, para tratar destas situações, utilizam-se avaliações levando em conta a taxa de crescimento, mas também os contextos de avaliação, que, via de regra, podem ser avaliados como o melhor caso, caso médio e pior caso. Cada uma dessas avaliações pode apresentar uma utilidade específica.

Quando se analisa o pior caso de um algoritmo, obtém-se a definição da situação em que todas as possibilidades de entrada serão atendidas, sendo que este valor pode ser utilizado como base para definir se o algoritmo é aceitável. Existem situações em que o tempo máximo de execução é fundamental, para que determinadas operações, tais como em áreas envolvendo segurança, por exemplo, sejam realizadas de forma adequada. Uma típica situação em que a análise de pior caso pode ser fundamental poderia ser o gerenciamento de rotas de aviação, situação em que existe um aspecto crítico relacionado com o tempo de resposta e encaminhamento de diretrizes.

Já a análise do melhor caso permite identificar a situação em que o algoritmo demonstra maior rapidez na sua execução, porém com a ressalva de que isso não irá ocorrer em todos os casos, não irá ocorrer na maioria dos casos. Portanto, como em geral esta análise não representa o comportamento real do algoritmo, possui pouca aplicação prática, em especial nas situações referidas, relacionadas à segurança.

O caso médio pode ser difícil de determinar, pois a sua identificação pode estar relacionada a fatores como a distribuição de dados de entrada, em geral apresentados de forma aleatória. Entretanto, pode ser interessante como uma medida do comportamento médio esperado do algoritmo, desde que seja analisada de forma correta, levando-se em conta as situações inadequadas que podem estar dependentes do tempo de execução.

2.2 Análise assintótica

Existe uma real necessidade de análise de algoritmos, como forma de garantir alguns requisitos associados com a performance desejada nas aplicações computacionais. Atualmente, apesar da constante melhoria dos recursos de processamento e armazenagem, observa-se um crescimento também constante nas demandas computacionais para as atuais aplicações. Um bom exemplo para essa avaliação são as atuais características de sistemas operacionais diversos, que proporcionam uma interação cada vez mais amigável e flexível aos usuários. Sabe-se que todos os recursos percebidos pelo usuário como facilidades são implementados por códigos que demandam uma grande quantidade de desenvolvimento e, por vezes, possuem grande complexidade.

Nas seções anteriores, foram destacados também diversos aspectos que podem ser utilizados para a análise de algoritmos, provendo uma boa base de comparação entre versões diversas, envolvendo aspectos como quantidade de dados de entrada e sua configuração. Com base nessa breve exposição, realizada nas seções anteriores, é possível identificar dois pontos importantes: a necessidade de mecanismos para avaliação de algoritmos e a dificuldade de tratamento de todos os aspectos envolvidos. Ao mesmo tempo, identifica-se com clareza atributos que são fundamentais no processo de análise, enquanto que outros podem ser desconsiderados, para fins práticos.

A análise assintótica dos algoritmos provê uma forma de atuação desconsiderando alguns aspectos específicos, relacionados com equipamentos ou constantes, e enfatizando a função de crescimento do tempo de execução dos algoritmos, com base no crescimento das entradas de dados.

Uma forma de analisar as implicações de desconsiderar algumas constantes ou detalhes do algoritmo pode ser bastante interessante neste ponto. Já foi identificado, na figura 5, o comportamento geral para três diferentes funções descrevendo o crescimento do tempo de execução de algoritmos. Na análise geral que foi feita, destacou-se a grande diferença no crescimento dos tempos de execução para o caso das funções quadráticas e exponenciais, comparadas com uma função de crescimento linear. Entretanto, ao analisarem-se alguns detalhes do conjunto de dados de entrada, pode-se verificar que, delimitado a um determinado conjunto de constantes, os resultados do tempo de execução são diferenciados. Veja, por exemplo, o conjunto de dados da tabela 2, onde estas três funções estão relacionadas, porém com valores menores, onde a variação dos valores de entrada (“n”) fica entre os valores 1 e 9. Cada uma das linhas descreve o resultado do crescimento do tempo de execução para as funções respectivas, assim na primeira linha estão os valores para o crescimento do tempo de execução segundo $T(n) = 5n$, na segunda linha o crescimento para $T(n) = 5n^2$ e na terceira para $T(n) = 2^n$. Apesar da função exponencial ($T(n) = 2^n$) apresentar tempos de execução muito maiores do que as outras duas, para grande parte das

situações, em determinados conjuntos de dados, isso não ocorre. Perceba-se que é apenas a partir de n = 5 que o tempo de execução da função exponencial é maior do que o da função linear. Isso está destacado na tabela 2, no item "a", que indica o valor de "n" no qual $2^n > 5n$. Portanto, pode-se dizer que no intervalo de "n" variando entre 1 e 4, a função exponencial apresenta tempo de execução menor do que a função linear. De forma similar, a tabela 2 identifica na coluna de "n" com valor 9 o ponto em que a função exponencial ultrapassa o tempo de execução da função quadrática, no qual $2^n > 5n^2$.

Tabela 2 – Crescimento de tempos de execução diferenciado entre constantes

	1	2	3	4	5	6	7	8	9	10	20
5n	5	10	15	20	25	30	35	40	45	50	100
$5n^2$	5	20	45	80	125	180	245	320	405	500	2000
2^n	2	4	8	16	32	64	128	256	512	1024	1048576

a) $2^n > 5n$ b) $2^n > 5n^2$

Fonte: o autor.

Apesar dos exemplos acima indicarem situações específicas onde apenas em um pequeno conjunto de valores a função exponencial apresenta um tempo de execução menor do que as outras duas, ele serve para destacar a existência dessas situações. No caso de um conjunto pequeno de dados, eventualmente pode ser interessante a utilização de um algoritmo que apresentaria maior taxa de crescimento, dada essa especificidade.

Na grande maioria das situações, com base nesse mesmo exemplo, a maior taxa de crescimento do tempo de execução está associada com a função exponencial. E é neste aspecto que a análise assintótica se baseia, para fornecer mecanismos de comparação que permitam uma ampla utilização. Ou seja, considera-se na análise assintótica uma série de notações que permitem a identificação da tendência de crescimento do tempo de execução dos algoritmos, com base em conjuntos de entradas de dados com tendência de crescimento e associados com grandes valores. A definição matemática dessas notações, que não será destacada neste livro, leva em conta as situações indicadas no exemplo da tabela 2, pois para cada uma dessas tendências de crescimento existe uma função que retorna determinados valores, sempre a partir de um ponto inicial para os valores de "n" ou, então, dentro de um determinado intervalo para os valores de "n".

A seguir são descritas resumidamente notações assintóticas utilizadas para a descrição do comportamento do tempo de execução de algoritmos, em função do

aumento do conjunto de entradas. As notações Ômega (Ω), Theta (Θ) e O (também referida como "Big O") descrevem conjuntos de funções com comportamento específico, que permitem a comparação de algoritmos. De uma forma simplificada, essas notações podem ser utilizadas associadas com as situações de melhor caso (Ω), caso médio (Θ) e pior caso (O) de execução do algoritmo. A utilização mais comum das notações pode ser encontrada na descrição de funções de crescimento do tempo de execução.

Assim, quando uma função f(n) descreve o limite máximo do crescimento para o tempo de execução de um algoritmo, pode-se dizer que este algoritmo está associado com o conjunto O(f(n)). Quando uma função f(n) descreve a menor taxa de crescimento, pode-se dizer que ela está no conjunto Ω(f(n)).

CAPÍTULO 3

TABELAS HASH

Neste capítulo serão apresentados os conceitos básicos de tabelas hash, um recurso utilizado em contextos de busca de informações. Também serão estimuladas as análises dos algoritmos de manipulação de dados com base em tabelas hash, de modo a proporcionar a aplicação dos conceitos do capítulo anterior. Serão apresentados os fundamentos de tabelas hash, exemplos e análises de casos de utilização e as principais operações envolvidas com o uso dessas estruturas.

Conforme analisado anteriormente, as necessidades de algoritmos ou classes de algoritmos estão sempre relacionadas com estruturas de dados e operações específicas. De acordo com as operações necessárias e com a natureza do problema, algumas opções de estruturas de dados podem apresentar características de excelente performance e adequação. As tabelas hash são um exemplo bastante interessante dessa situação, tendo sido projetadas para o atendimento das necessidades de operações de busca de informações.

Nas situações em que um conjunto extenso de dados precisa ser acessado para a verificação da existência ou localização de um determinado valor, algumas abordagens partem da varredura dos valores, empregando diversas otimizações, que possibilitam diminuir o tempo gasto na varredura. Mas, em geral, os tempos podem ser grandes, em especial quando o conjunto de valores é extenso. Implementações de algoritmos de busca binária apresentam complexidade na ordem de Θ ($\log_2 n$), enquanto que algoritmos mais simples, como busca direta apresentam ordem O(n).

A proposta da tabela hash, idealmente, apresenta-se com tempo de resposta na ordem O(1), ou seja, configura-se como uma método extremamente eficiente, com tempo de execução constante, o que significa que a quantidade de elementos armazenados não influencia o tempo gasto na execução de uma busca. Um conjunto de 10 valores e um conjunto de 20 mil valores apresenta o mesmo tempo de execução para as operações de busca. É claro que existem diversos detalhes e algumas situações onde a performance será modificada, mas essa abordagem é bastante eficiente. Algumas das restrições para o tipo de operações de busca também serão analisadas adiante. A seguir, serão introduzidas as noções gerais de tabelas hash.

Tabela 3 – Organização típica de um vetor com dados alfanuméricos

Operações típicas	Posição	Valor

a) Busca pelo valor: “ Rio de Janeiro”	0	Porto Alegre
b) Varredura de todas as posições	1	São Paulo
c) Comparação de valores armazenados	2	Belo Horizonte
d) Localização na posição “ 3”	3	Rio de Janeiro

Fonte: o autor.

Na tabela 3 pode-se observar a organização típica de um vetor em memória, contendo dados alfanuméricos. A coluna “Posição” corresponde ao endereço de memória no qual cada um dos valores está armazenado. Quando a busca por determinado valor é necessária, as operações executadas, de modo geral, são as descritas na tabela 3, na coluna esquerda. Mesmo com uso de estruturas mais otimizadas, o fator fundamental a ser destacado aqui é a necessidade de operações de acesso à memória para recuperação de valores armazenados e comparação destes valores com o valor desejado. Justamente essas duas etapas é que são suprimidas com o uso de tabelas hash.

A tabela 4 ilustra de forma simplificada esta estratégia, que consiste na criação de uma função de mapeamento para que os valores desejados sejam acessados diretamente, a partir de um cálculo feito com o seu conteúdo, cujo resultado indicará a posição de memória do registro desejado. Desta forma, quando a busca é feita por um determinado valor, tal como “Porto Alegre”, no exemplo da tabela 3, o mapeamento com base nesse valor irá gerar o índice “0”, que indica a posição de memória onde o valor está armazenado. Deve se ressaltar que o mapeamento com base no valor original sendo buscado não envolve a utilização de uma tabela para busca dos índices, tal como pode induzir uma análise mais superficial da tabela 4, pois neste caso o problema de varredura de um vetor apenas seria deslocado. Este exemplo serve apenas para identificar, de forma simples, a proposta geral de funcionamento de tabelas hash.

Tabela 4 – Simplificação da operação de mapeamento de chaves e índices

Valor	Posição
Belo Horizonte	2
Porto Alegre	0
Rio de Janeiro	3
São Paulo	1

Posição	Valor
0	Porto Alegre
1	São Paulo
2	Belo Horizonte
3	Rio de Janeiro

Fonte: o autor.

O ponto fundamental para o entendimento do modo como as tabelas hash possibilitam a implementação com tal eficiência está na utilização de uma função de mapeamento, que irá utilizar a chave de acesso procurada e transformará esta chave no índice correspondente à posição de memória na qual a chave que está sendo solicitada encontra-se armazenada. Para tal, considera-se um conjunto de chaves (C) correspondendo aos valores sendo buscados. O conjunto de posições de armazenamento (P) corresponde aos índices de acesso aos dados em memória. Desta forma, a função de mapeamento Hash realiza a associação entre os valores de C, o conjunto de chaves, com os valores do conjunto P, contendo a indicação das posições de armazenamento.

Com base nesta descrição resumida, é possível realizar algumas análises a respeito das suas possibilidades e implicações. Uma delas seria a observação de que os recursos de tabelas hash não são adequados para situações em que múltiplas ocorrências dos valores utilizados como chave são possíveis. Também observa-se que a identificação de faixas de valores não seria uma operação otimizada, como, por exemplo, em uma situação onde todas as chaves entre dois limites devem ser recuperadas. Ou então o caso de varredura dos valores em ordem de suas chaves. Por outro lado, quando a operação de busca envolve a identificação da posição de uma chave específica, o processo é extremamente eficiente.

3.1 Funções de dispersão

A função de dispersão, ou função hash, possui como objetivo receber uma chave de valor "v" e, depois de aplicado algum procedimento de transformação ou cálculo, gerar um resultado que consiste de um número inteiro dentro do intervalo definido pela tabela hash. Supondo que a tabela hash possua N posições, o resultado da função de dispersão deve estar localizado entre o intervalo de valores de 0 até N-1.

A forma mais simples de imaginar a implementação de uma função de dispersão é a associação simples de valores e posições, o que poderia ocorrer no caso de números, por exemplo. Essa associação corresponderia basicamente ao esquema de acesso existente com o uso de vetores. Assim, a chave de valor "i" ocuparia a posição "i" da tabela hash. No caso de um grande número de valores, isso implica em que um grande número de posições de memória seria necessário. No caso de um conjunto esparso de valores de entrada, a situação acarretaria um grande desperdício de posições de armazenamento.

Em geral, as implementações de funções de dispersão são realizadas com a premissa de mapear um grande conjunto de valores possíveis em uma tabela de tamanho reduzido, para economia de memória. Isso pode ser feito tanto com valores numéricos como com dados alfanuméricos. Na figura 6, por exemplo, está ilustrada, de forma simplificada, a relação entre o conjunto de chaves, a função de mapeamento ou

de dispersão e a tabela hash. Em uma situação como esta, a função de mapeamento poderia empregar heurísticas envolvendo os valores de codificação ASCII (*American Standard Code for Information Interchange*) para os dados alfanuméricos, em operações bastante variadas, tais como a simples montagem de uma sequência numérica com base nas letras de cada palavra, ou na geração de um somatório levando em conta a posição da letra e seu valor na codificação ASCII.

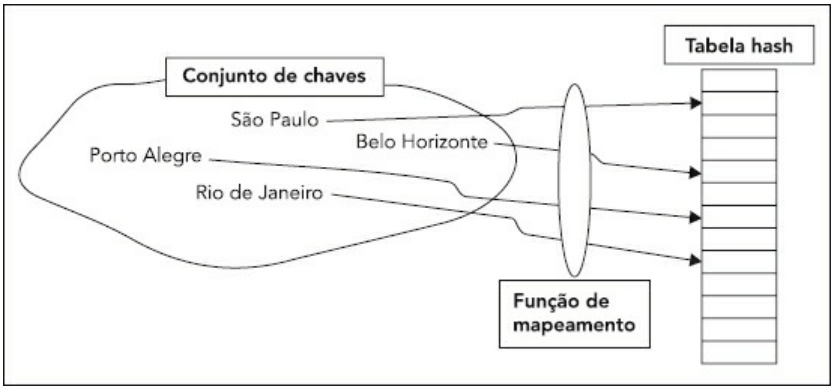


Figura 6 – Exemplo simplificado do mapeamento de chaves e índices.
Fonte: o autor.

Para exemplificar essa situação com um caso prático, vejamos qual seria o resultado possível da utilização de uma função simples de mapeamento, que apenas empregasse os valores da codificação ASCII para cada letra do dado alfanumérico, somando todos os valores para a geração de um índice. Utilizando como exemplos os nomes de cidades indicados na figura 6, poderíamos gerar os resultados abaixo, ilustrados na figura 7 caso o nome "Porto Alegre" fosse utilizado. A figura 7 descreve o valor da codificação ASCII para as letras da chave a ser mapeada. Após a identificação dos valores, a sua soma apresenta como resultado o valor 839, que poderia ser utilizado como índice para uma tabela hash, na qual a palavra "Porto Alegre" estaria armazenada. No exemplo, também está destacada uma possibilidade utilizada como forma de ajustes nas chaves geradas com quaisquer procedimentos e que precisem ser adequadas para tabelas hash de tamanho mais reduzido.

No caso de dados alfanuméricos, pode-se chegar a valores de soma grandes que ultrapassem as possibilidades de armazenamento da tabela e, nesse caso, operadores como mod (resto inteiro da divisão) podem ser empregados para o ajuste. Na figura 7 está ilustrada uma situação hipotética em que a tabela hash possui 128 posições e,

portanto, o primeiro valor obtido com a soma dos valores de codificação ASCII da chave é ainda utilizado com a operação mod, cujo resultado é utilizado como índice para a tabela hash. No exemplo, o valor da soma, igual a 839, gera como resultado da operação mod com o valor 128 o índice 71, utilizado para o acesso aos dados.

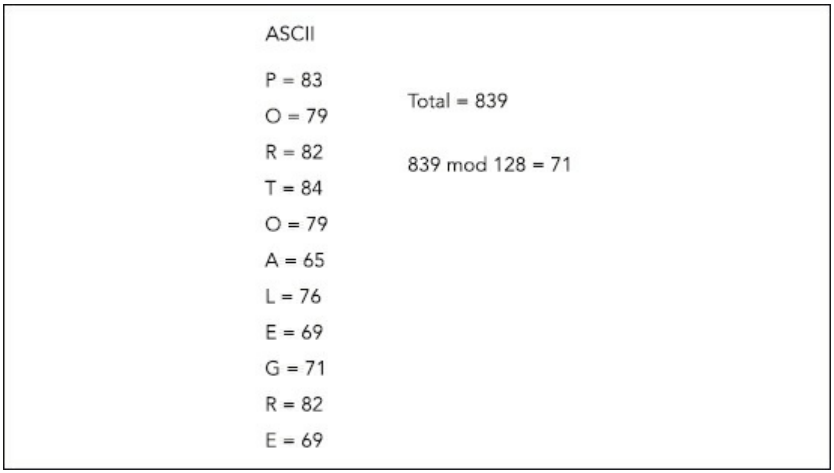


Figura 7 – Exemplo simplificado de função de mapeamento para dados alfanuméricos.
Fonte: o autor.

Do mesmo modo, se a palavra a ser mapeada fosse "NOMETESTE", a soma dos valores da codificação ASCII para suas letras seria igual a 692, a partir dos valores: 78, 79, 77, 69, 84, 69, 83, 84, 69. Utilizando o operador mod e considerando 128 posições na tabela hash, o índice gerado seria 52.

Uma das possibilidades existentes na aplicação das funções de dispersão é a colisão de valores, o que ocorre quando duas chaves diferentes geram o mesmo resultado para o índice. Veja que ao utilizar uma chave hipotética como "NOMEddP", os valores da codificação ASCII somariam 583 (78 + 79+77+69+100+100+80), número que aplicado ao operador mod, considerando-se 128 posições da tabela hash, resultaria no mesmo índice da palavra "Porto Alegre", cujo valor é 71. Esta situação é ilustrada na figura a seguir, de número 3.5, e representa a situação conhecida como colisão, que deve ser evitada, para garantir que o uso da tabela hash tenha condições de aproveitar a sua performance ideal. Uma das formas de evitar a ocorrência de colisão está associada com a escolha cuidadosa da função de dispersão, sendo que o balanceamento entre o tamanho da tabela hash e o número de possíveis chaves

também pode diminuir essa probabilidade.

Porém, no caso de ocorrência de colisão, esta pode ser tratada de diversas formas, algumas bastante intuitivas, tal como exibido na figura 8, onde cada índice da tabela hash aponta para elementos que podem constituir em uma lista, sendo utilizados quando houver colisão com este índice da tabela hash. Neste exemplo as chaves "Porto Alegre" e "Manaus" identificam uma situação de colisão, sendo que a partir da tabela hash estão dispostos, em uma lista, os nomes das duas cidades. Neste caso, a operação de localização de valores com uso da técnica de hash sofre uma pequena modificação, incluindo o tratamento da lista de valores, tanto para operações busca, como para inserção ou retirada de valores.

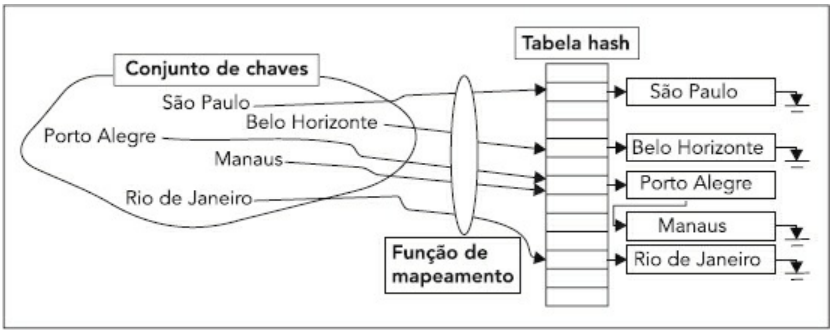


Figura 8 – Exemplo simplificado de tratamento de colisão.
Fonte: o autor.

De acordo com a taxa de colisões e o número de elementos constando nas listas, é possível a manutenção de tempos de execução bastante eficientes, quando comparados com outros métodos de acesso a dados. Algumas técnicas também permitem o ajuste de conjuntos já armazenados, de acordo com o acompanhamento de alguns parâmetros, tais como a taxa de colisões e a proporção entre o número de chaves e a tabela hash. Em algumas situações, quando existe uma configuração de ocorrências muito alta de colisões e o número de chaves cresce muito em relação ao tamanho da tabela, é possível a reorganização da mesma, com o aumento de seu tamanho e atualização de todos os índices, gerando uma nova configuração, na qual as ocorrências de colisão são reduzidas e a performance adequada é mantida. Essa técnica é referida como "rehashing" e é uma das alternativas encontradas em algumas implementações para manter um balanço adequado entre o espaço utilizado para a tabela hash e a sua performance.

Uma situação interessante ocorre nos casos em que as chaves são conhecidas

previamente e são estáticas, o que pode se verificar quando o conjunto de dados é descrito inicialmente, tal como em casos de um dicionário, um site da *web* ou de documentação técnica sobre determinados produtos. Nessas situações é possível implementar o que se denomina como hash perfeito. Dado que o conjunto de chaves é conhecido previamente, existe a possibilidade de utilização desse conhecimento para a definição da função de dispersão, que pode ser definida de modo a evitar qualquer tipo de colisão, garantindo que a eficiência seja a maior possível nas operações de busca de informações.

3.2 Hash aberto e hash fechado

Tendo sido apresentada a problemática envolvida com a definição das funções de dispersão e com o tratamento das situações de colisão, que em boa parte dos casos serão inevitáveis, aprofundaremos agora a análise sobre as possibilidades de implementação de tabelas hash, com a inclusão de políticas de tratamento para colisões. Dada a característica de alta performance esperada das implementações de tabelas hash, as políticas de tratamento de colisões são fundamentais para manter um bom desempenho nos diversos contextos de aplicação. Também é importante destacar que diversos fatores estão envolvidos no bom encaminhamento dessa situação, tais como o correto balanceamento entre o espaço utilizado pelas tabelas hash e a quantidade de chaves existentes, o comportamento destas chaves quanto a modificações, as demandas por utilização de memória, entre outras.

As duas linhas mais gerais de encaminhamento para essa situação são conhecidas como hash aberto e hash fechado. Já foi destacada anteriormente uma situação conhecida como hash perfeito, na qual todas as chaves são previamente conhecidas e com isso evita-se a colisão com a seleção de uma função de dispersão que garante esta característica para o conjunto de dados a ser mapeado. Entretanto, esta situação é bastante específica, não correspondendo à maioria das situações práticas de aplicação de hash, para as quais normalmente são empregadas variações das duas classes analisadas a seguir.

O hash aberto é também conhecido com o hash de encadeamento separado, pelo fato de tratar das colisões com o armazenamento dos dados fora da tabela hash, normalmente em listas encadeadas. A ideia mais geral dessa forma de resolução de colisões está ilustrada na figura 8. No hash aberto, cada entrada da tabela hash contém uma indicação sobre sua utilização e no caso de armazenar algum valor, este será encontrado a partir de uma estrutura de lista ligada. Essa estrutura proporciona o suporte necessário para os casos de colisão, onde os diversos elementos associados com o mesmo índice da tabela hash serão encontrados na lista ligada. Na figura 9, a seguir, está ilustrada uma situação de mapeamento de chaves utilizando a filosofia de hash aberto, para uma tabela hash de 13 posições e utilizando uma função de

dispersão H(x) = x mod 13.

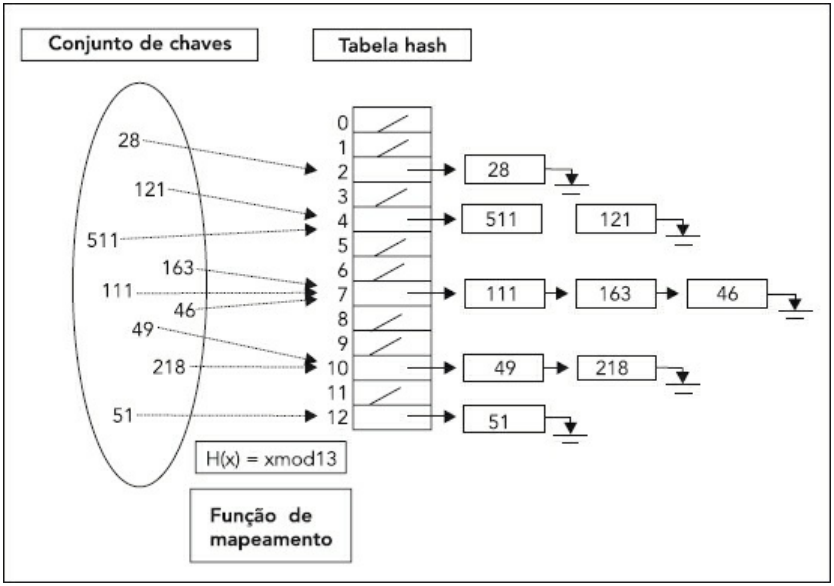


Figura 9 – Resumo do uso de hash aberto.
Fonte: o autor.

Em uma análise prévia sobre as possibilidades de performance neste contexto, deve ser destacado que grande parte do resultado está associado com a proporção entre a quantidade de chaves e o tamanho da tabela hash. No caso de uma distribuição adequada, é possível que poucas chaves apresentem situação de colisão, gerando uma possibilidade de performance na ordem de O(1). No caso hipotético extremo, em que todas as "n" chaves seriam mapeadas para o mesmo índice na tabela hash, entretanto, a performance seria da ordem de O(n), dado que estaria envolvida a pesquisa pela chave desejada na lista ligada contendo os valores.

Assumindo-se uma taxa de ocorrência aceitável para as operações de colisão entre chaves, existem alguns recursos que podem minimizar o tempo necessário para a recuperação dos dados nas listas ligadas. Por exemplo, é possível tratar estes dados de forma ordenada, tanto pelos valores armazenados como por outros parâmetros, tais como a quantidade de acessos ou datas de acesso. De acordo com a aplicação, cada uma dessas alternativas pode oferecer suporte adequado. Por exemplo, quando os

dados são ordenados por frequência ou data de acesso, podem ser privilegiadas algumas aplicações que possuem esse comportamento de forma predominante. Já no caso de ordenação de dados pelos valores armazenados, é possível abreviar algumas pesquisas, quando o valor buscado pode ser identificado como não existente na lista, através da comparação durante a pesquisa na lista.

O hash fechado também é conhecido como hash de endereçamento aberto, apesar desta denominação proporcionar alguma confusão quando trata-se de uma técnica que encaminha a solução das colisões de forma diferenciada da técnica conhecida como hash aberto. Neste caso, a abordagem do tratamento de colisões segue a proposta mais geral do uso de hash, através de uso de alguma forma de cálculo para determinar a posição onde os dados serão armazenados, mesmo no caso de alguma situação de colisão. Evita-se, aqui, o uso de estruturas complementares, como uma lista de valores, utilizado na abordagem de hash aberto.

Assim, nas implementações que seguem a proposta de hash fechado, todos os dados são armazenados na tabela hash, diretamente. Utiliza-se um passo adicional na função de dispersão para que, no caso de uma colisão, seja efetuado um cálculo ou uma atividade de pesquisa que irá determinar em qual posição da tabela hash a chave será armazenada. Além desse cálculo, deve ser realizada uma operação de comparação de valores, porém utilizando apenas os valores armazenados dentro da tabela hash. Deste modo, a abordagem de hash fechado evita o uso de estruturas auxiliares, tais como as listas ligadas usadas no hash aberto. Todos os procedimentos de manipulação dos dados, seja para inserção ou para consulta, seguem apenas os resultados gerados pelos cálculos da função de dispersão. Esse passo adicional pode ser implementado de diversas formas, sendo que, em geral, o objetivo destas implementações está concentrado na minimização das operações de comparação ou de varredura das posições da tabela hash.

Uma das implementações conhecidas para esta abordagem consiste em organizar a tabela hash em agrupamentos, dentro dos quais são tratadas as colisões de chaves. Por exemplo, no caso do uso do operador mod, é possível que diversas chaves gerem o mesmo valor. No caso deste valor indicar um grupo de posições e não apenas uma posição da tabela, o procedimento de inserção verificaria se a primeira posição do agrupamento está livre para o armazenamento do valor e, no caso de estar ocupada, passaria para a posição seguinte, realizando o mesmo teste. Na figura 10, abaixo, está descrito um exemplo de utilização da filosofia de hash fechado. Neste caso, a tabela hash é organizada em agrupamentos e estes são indicados pela função de dispersão ou de mapeamento. Quando ocorre uma colisão, a chave é armazenada no mesmo agrupamento, porém na primeira posição disponível. No caso do agrupamento não possuir posições disponíveis, existe uma área adicional sem restrição de espaço, que é utilizada para o armazenamento, na qual a pesquisa pelas chaves seria feita de modo sequencial. Para o exemplo foram utilizadas as mesmas chaves do caso anterior, descrito na figura 9, sendo que a tabela hash foi organizada em 4 agrupamentos e a

primeira parte da função de dispersão ou mapeamento é descrita como H´(x) = x mod 4. Deste modo as chaves são associadas com o agrupamento da tabela hash através de um cálculo simples. Quando a posição inicial do agrupamento indicado estiver disponível, o valor da chave será armazenado diretamente na primeira posição. No caso da posição estar ocupada, será iniciada uma busca sequencial para identificar a próxima posição disponível, onde será armazenado o valor da chave. No caso de operações de busca de valores, após o mapeamento para o agrupamento, será realizada uma comparação para verificar se a chave desejada encontra-se armazenada dentro do mesmo. Por exemplo, no caso da busca pela chave "49", a função de dispersão geraria o valor 1 (H´(49) = 49 mod 4). Ao acessar a primeira posição deste agrupamento, será encontrada a chave de valor "121", indicando a necessidade de pesquisa na próxima posição, onde será encontrada a chave desejada.

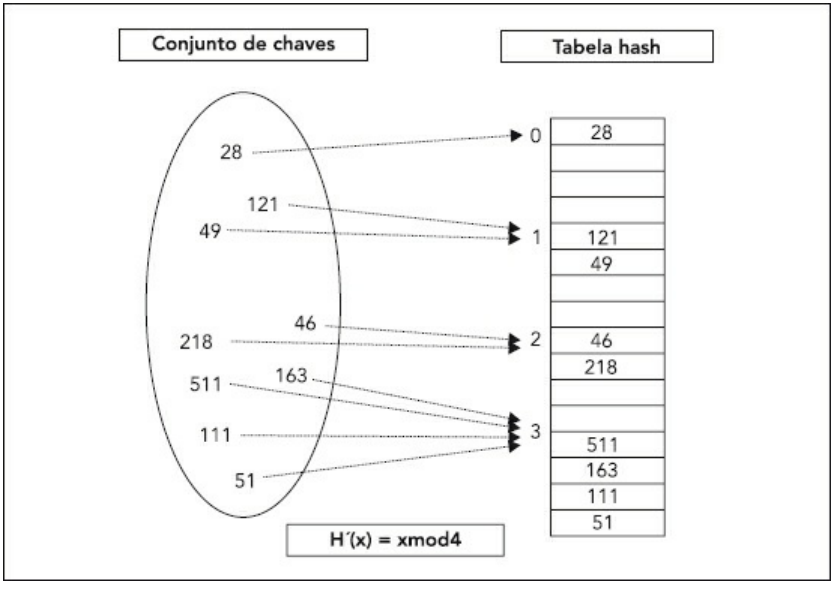


Figura 10 – Resumo do uso de hash fechado.
Fonte: o autor.

Capítulo 4

HEAP

Neste capítulo serão apresentados os conceitos básicos das estruturas de dados heap, que são uma adaptação de estruturas em árvore, com base em uma implementação eficiente, em um vetor. São apresentadas as noções gerais e os principais conceitos necessários para a utilização desta estrutura. São realizadas análises sobre suas possibilidades de utilização de forma eficiente, em especial em áreas como manutenção de filas de prioridades ou em ordenação de valores. Também são descritos algoritmos que utilizam a estrutura, tais como o heapsort.

Considerando-se que o leitor deste livro já possui conhecimentos sobre os recursos de estruturas de dados, tais como árvores, é descrita a seguir uma estrutura de dados baseada em uma estrutura do tipo árvore binária, sendo que um de seus diferenciais é justamente a implementação, que é realizada com base em vetores. A estrutura heap diferencia-se dessa forma das estruturas de árvore, normalmente implementadas com uso de ponteiros para a organização das relações entre seus nodos. As aplicações mais conhecidas e adequadas para esse tipo de estrutura de dados estão associadas com manutenção de filas dinâmicas de prioridades e também com operações de ordenação de dados, a partir de um algoritmo conhecido como heapsort.

Uma fila de prioridades pode ser aplicada em diversas situações nas quais seja importante a performance na recuperação dos itens e, em especial, a performance para a manutenção de novos itens, que devem ser armazenados de forma ordenada. Quando a tendência da aplicação é o tratamento de um grande número de novos itens, de forma frequente, torna-se fundamental uma boa solução para o processo de inserção ordenada de itens. Além disso, a busca pelos elementos de maior prioridade normalmente é de grande importância. Exemplos de filas de prioridades podem indicar necessidade de manter informações sobre importância ou urgência, tal como em uma fila de atendimentos médicos, onde a especificidade de cada doença pode indicar a necessidade de atendimento prioritário. Também podem ser utilizadas informações relacionadas com tempo, tal como no gerenciamento de um conjunto de aeronaves em operações de pouso e decolagem, onde os tempos de aproximação ou os horários de partida são determinantes para a indicação de atendimento.

A estrutura heap parte da representação lógica de uma árvore binária essencialmente completa, significando, desta forma, uma árvore binária onde os nodos estão completamente preenchidos, com eventuais exceções no nível mais baixo. Além deste aspecto, o heap mantém os elementos de seus nodos parcialmente ordenados, de

modo que sempre existirá uma relação de ordenação entre o nodo e seus nodos filhos.

Esta ordenação pode ser observada de duas formas, que definem duas abordagens para geração e manutenção de heaps. Quando cada nodo possui valores maiores ou iguais aos valores de seus nodos filhos, trata-se de uma abordagem conhecida como max-heap (ou heap máximo). Caso contrário, quando cada nodo possui valores menores ou iguais aos valores de seus nodos filhos, trata-se de uma abordagem conhecida como min-heap (ou heap mínimo).

Na figura 11, a seguir, estão destacados estes aspectos. São exibidos dois casos de heap, com as ordenações de max-heap e min-heap para um mesmo conjunto de dados.

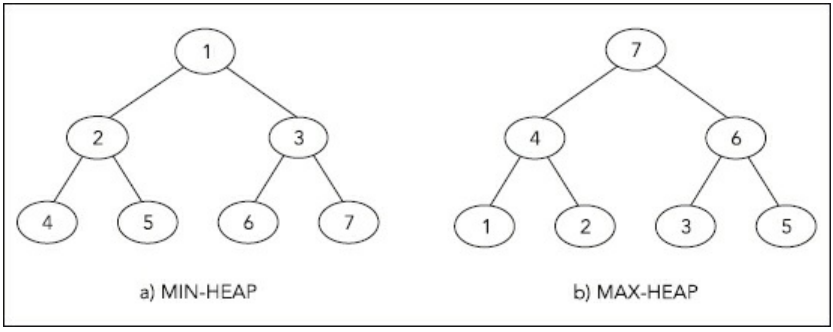


Figura 11 – Exemplos de max-heap e min-heap.
Fonte: o autor.

4.1 Implementação de heap

A implementação física do heap ocorre de forma diferenciada da maior parte das implementações de árvores, pois ele não utiliza organizações de ponteiros para indicar as relações entre os nodos. No seu lugar é empregada uma organização vetorial, sendo que os dados dos nodos são armazenados na memória sequencialmente, de acordo com a ordem determinada pela navegação na árvore binária, a partir do nodo pai, varrendo sempre todos os nodos filhos, da esquerda para a direita, esgotando sempre a navegação em um nível antes de acessar o nível seguinte.

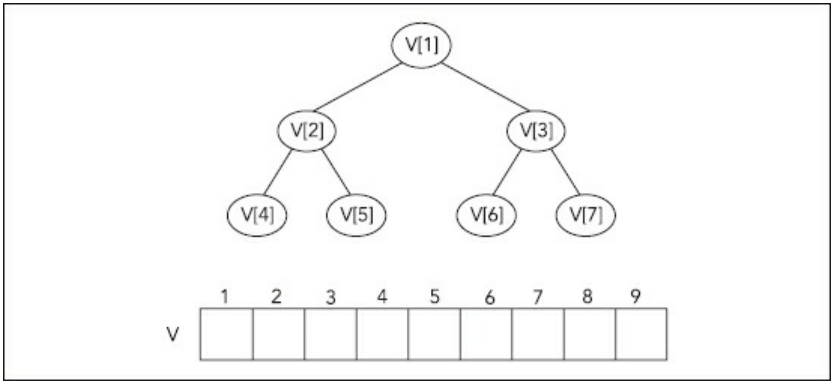


Figura 12 – Implementação de elementos do heap em um vetor.
Fonte: o autor.

Com a organização ilustrada na figura 12, percebe-se que o valor do nodo raiz será sempre utilizado para armazenamento na primeira posição do vetor, sendo seguido pelos valores dos nodos filhos do próximo nível, selecionados da esquerda para a direita, seguidos, por sua vez, pelos nodos do nível seguinte e assim sucessivamente. Note que no caso do max-heap, o valor armazenado na primeira posição do vetor será sempre igual ao maior valor. No caso do min-heap, o primeiro valor do vetor será sempre o menor valor. Os demais valores do vetor seguirão a sua disposição de forma ordenada, de acordo com o formato utilizado.

Outro aspecto a destacar é a forma facilitada de acesso aos nodos relacionados (pai ou filhos) a partir de um nodo qualquer. De acordo com esta organização, o pai de um nodo *i* estará sempre armazenado neste vetor, na posição i/2, considerando-se o resultado inteiro da divisão. Por exemplo, tomando como base a figura 12, o valor do nodo pai do valor armazenado na posição 2 do vetor é o valor armazenado no nodo 1, dado o cálculo indicado (2/2 = 1). Do mesmo modo, com o resultado inteiro da divisão de 3 por 2, temos que o valor do nodo pai do nodo armazenado na posição 3 é o nodo 1. A localização dos nodos filhos também se beneficia da organização vetorial, de modo que sempre é possível identificar o filho na posição esquerda ou direita para um nodo na posição *i* a partir dos seguintes cálculos: posição 2*i* para o filho na posição esquerda e 2*i*+1 para o filho na posição direita. Isso pode ser verificado no caso dos filhos do nodo na posição 3, que se encontram nas posições 6 (2*3) e 7 (2*3+1), respectivamente. A figura 13 ilustra graficamente esta relação.

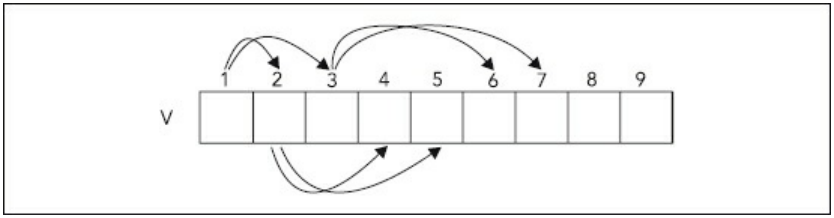


Figura 13 – Cálculos para acesso aos elementos do heap em um vetor: Pai(i) = i/2, FilhoEsquerdo(i) = 2i, FilhoDireito(i) = 2i+1

Fonte: o autor.

4.2 Operações fundamentais

A estrutura heap não se mostra adequada para a realização de algumas operações, tais como a navegação ordenada por seus valores componentes e mesmo a localização de um determinado componente. Neste caso, como a estrutura é fracamente ordenada, a estimativa de localização de um elemento, heap com "n" valores, seria da ordem O(n), portanto não adequada para estas operações.

Entretanto, algumas operações, tais como a identificação do elemento de maior prioridade, são bastante simples e eficientes, pois, como visto, sempre o primeiro elemento do vetor utilizado para a implementação do heap contém o menor ou o maior valor, de acordo com a organização utilizada. Caso seja utilizado para armazenar uma fila de prioridades, por exemplo, a operação de localização do valor de maior prioridade é simplesmente a leitura do primeiro elemento deste vetor.

Além dessa característica, o tratamento de filas de prioridades envolve mais duas operações fundamentais, que são a alteração de valores armazenados, como no caso em que é alterada a prioridade de um determinado item e ainda a inserção de novos valores, além da operação complementar, de retirada de valores, que é sempre realizada ao ser acessado o valor mais prioritário. Estas operações de alteração e inserção de valores também são implementadas de forma eficiente no heap, com as operações de comparação e troca de valores entre nodos pai e filho. A seguir são descritas em mais detalhes e exemplificadas essas operações.

No caso da troca de um determinado valor, ou da inserção de um novo valor na estrutura, serão realizadas, sucessivamente, as operações de comparação entre os valores do nodo alterado ou inserido e seu nodo pai. No caso de uma ordenação do tipo max-heap, o valor inserido ou alterado é comparado com o nodo pai. Caso seja maior, haverá a troca desses valores. O nodo alterado repetirá a troca com o nodo pai, caso se mantenha esta situação, repetindo a operação até que seja alcançado o nodo raiz. Deste modo, será mantida a organização do heap em suas propriedades. Veja na

figura 14 um exemplo deste processo, considerado a partir da inserção de um novo nodo no heap. Destaca-se que a inserção de um novo nodo é sempre realizada colocando-se o novo valor no último nodo da estrutura.

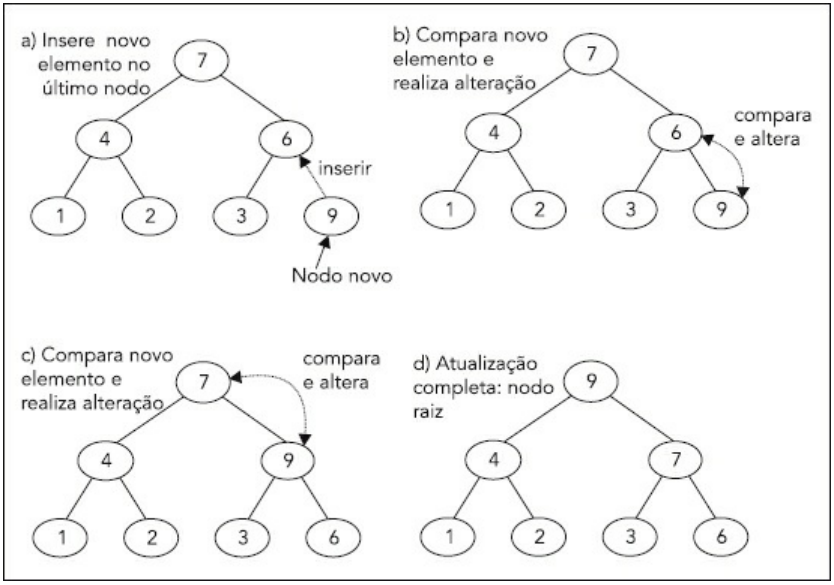


Figura 14 – Resumo do processo de inserção de novo valor no heap.
Fonte: o autor.

Um processo similar ocorre quando por algum motivo um determinado elemento possui seu valor alterado. Neste caso, o mesmo processo de comparações e trocas ocorre, a partir do elemento modificado. A condição de parada deste processo pode ser o ajuste do nodo raiz ou então a identificação de uma posição em que não ocorre a troca do valor alterado com o valor do nodo pai. Veja um exemplo na figura 15, a seguir.

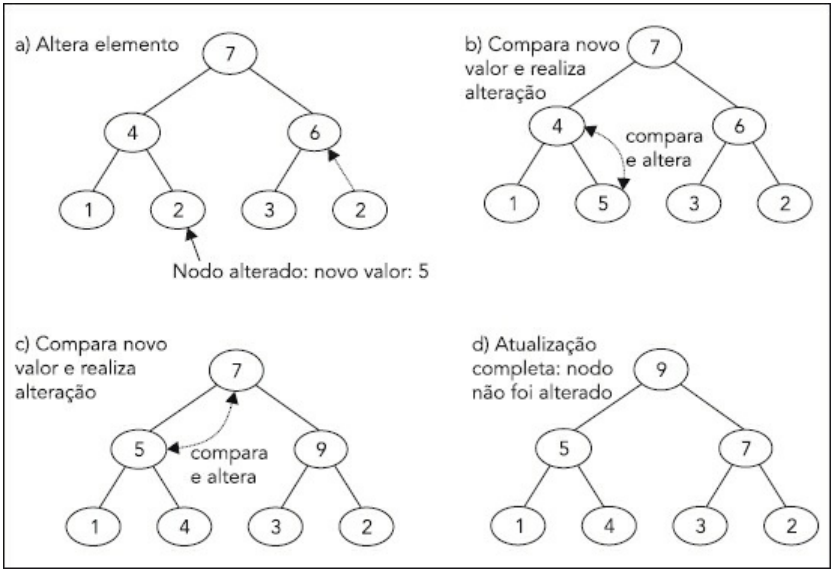


Figura 15 – Resumo do processo de alteração de valor no heap

Fonte: o autor.

Quando ocorre a recuperação do valor mais prioritário, existe a necessidade do processo de ajustes, pois neste caso o valor do primeiro nodo será retirado do heap. O procedimento de atualização, neste caso, envolve, em algumas implementações, a utilização do último nodo para a substituição do nodo retirado, na posição raiz. Após a troca, o processo de ajuste será realizado, de forma a promover a atualização do heap. A figura 16 ilustra esse processo.

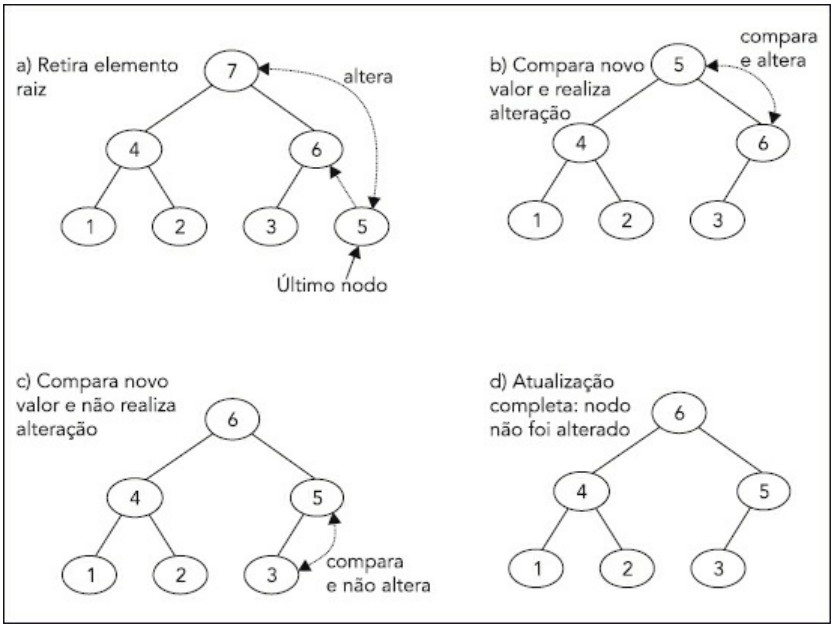


Figura 16 – Resumo do processo de remoção do elemento raiz do heap e sua atualização.

Fonte: o autor.

Durante o processo de atualização do heap, tal como ilustrado na figura 16, durante as comparações de um nodo pai com os nodos filhos, a troca deve sempre observar o nodo filho que possua o maior valor, e ser realizada com este elemento, pois, do contrário, existe um resultado indesejado, no qual ocorre uma violação da estrutura de heap, com um nodo pai armazenando um valor menor que um nodo filho. Por exemplo, supondo que durante o processo de atualização do heap um nodo de valor 22 seja pai de um nodo de valor 33 e outro de valor 11. Neste caso, considerando uma organização max-heap, o nodo escolhido deve ser sempre o nodo filho de maior valor, gerando o resultado de uma configuração onde o nodo pai 33 possui um filho com valor 22 e outro com valor 11.

4.3 Implementação de operações com heap

A seguir são descritas e analisadas as implementações necessárias para as operações com estruturas heap.

A figura 17 descreve o pseudocódigo para a atualização recursiva de um heap máximo, sendo que esta mesma estrutura pode ser utilizada em implementações diferenciadas, como o heap mínimo, com o ajuste de alguns componentes. Considera-se que o procedimento recebe o vetor que armazena o heap e um índice do ponto de início do ajuste. Este ponto de início pode ser qualquer posição do heap, o que será utilizado para o procedimento recursivo descrito.

```
void Atualiza_Heap_Maximo(Vetor h, int inicio){
1.   int esq = filho_esquerdo(inicio); //identifica índice do filho esquerdo
2.  int dir = filho_direito(inicio); //identifica índice do filho direito
3.  int maior;
4.  Se (esq <= tamanho_do_heap(v)  e h[esq] > h[inicio])
5.       maior = esq; //filho esquerdo é maior que valor em índice inicio
6.  Senão
7.       maior = inicio; //filho esquerdo não é maior que valor em índice inicio
8.  Se (dir <= tamanho_do_heap(v)  e h[dir] > h[maior])
9.       maior = dir; //filho direito é maior que valor em índice inicio e filho esquerdo
10.  Se maior ≠ inicio
11.      troca_valores_heap(inicio, maior); //troca valores no HEAP
12.      Atualiza_Heap_Maximo(h, maior); //invoca procedimento recursivamente
13  }
```

Figura 17 – Pseudo código para atualização de heap máximo.
Fonte: o autor.

A figura 18, a seguir, ilustra a utilização do pseudocódigo para a atualização de um exemplo de heap no qual o elemento no índice 1 está em desacordo com a propriedade do heap máximo. No primeiro elemento da figura 18 está indicado o estado inicial do heap, sendo que sobre cada elemento está indicado o número que representa o seu índice no vetor de armazenamento do heap. Estes números de índice do vetor são os utilizados pelo procedimento. A descrição da primeira etapa indica o resultado para uma chamada do procedimento de atualização do heap com a indicação do índice inicial de valor 1. Neste caso, ele será comparado com os elementos de índice 2 e 3, sendo descoberto que o maior elemento é o de índice 3, portanto, como o maior

elemento é diferente do elemento inicial, haverá a troca desses elementos e uma chamada recursiva para o mesmo procedimento, agora com a indicação do elemento atualizado será realizada. A chamada recursiva é quem garante que os valores serão atualizados corretamente, de acordo com ajustes realizados no heap.

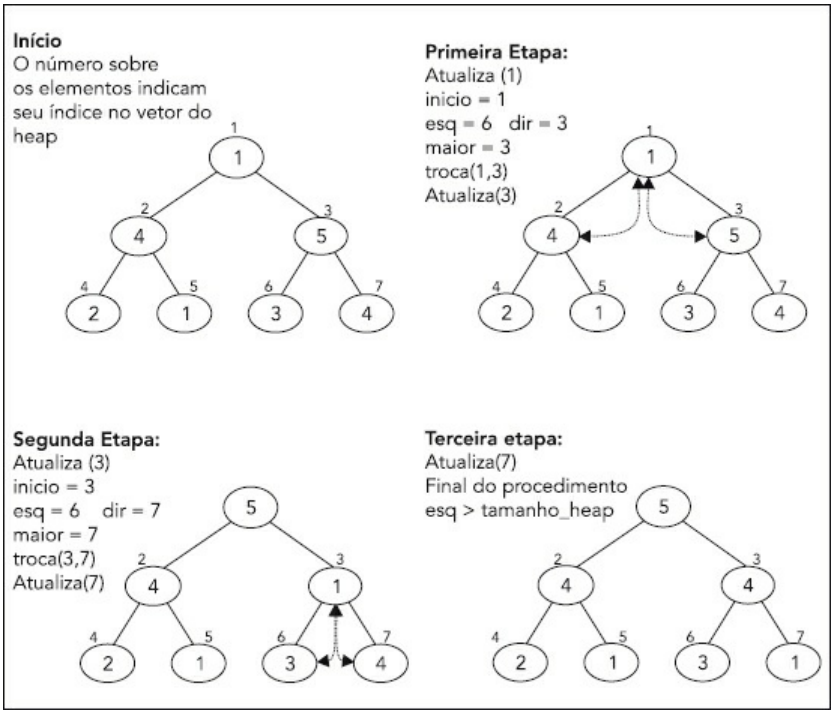


Figura 18 – Exemplo de atualização de heap máximo.
Fonte: o autor.

Conforme pode ser observado pelo leitor, este mesmo procedimento permite a construção de um heap, caso seja utilizado de acordo com uma organização que garanta a verificação de todos os elementos. Isso é possível com base na própria estrutura do heap, pois no caso de um tamanho igual a n, todos os itens de índice ((n/2)+1) até n constituem situações onde existem heap de tamanho 1. Portanto, utilizando este procedimento repetidamente, com base no índice inicial variando entre n/2 até 1, garante-se a atualização de todos os seus elementos. A figura 19 ilustra esta

situação, apontando a sequência de índices visitados pelo procedimento de atualização, o que demonstra que todos os elementos do heap serão analisados. No caso de alterações, o mecanismo recursivo garante que todos os elementos relacionados também serão analisados e ajustados.

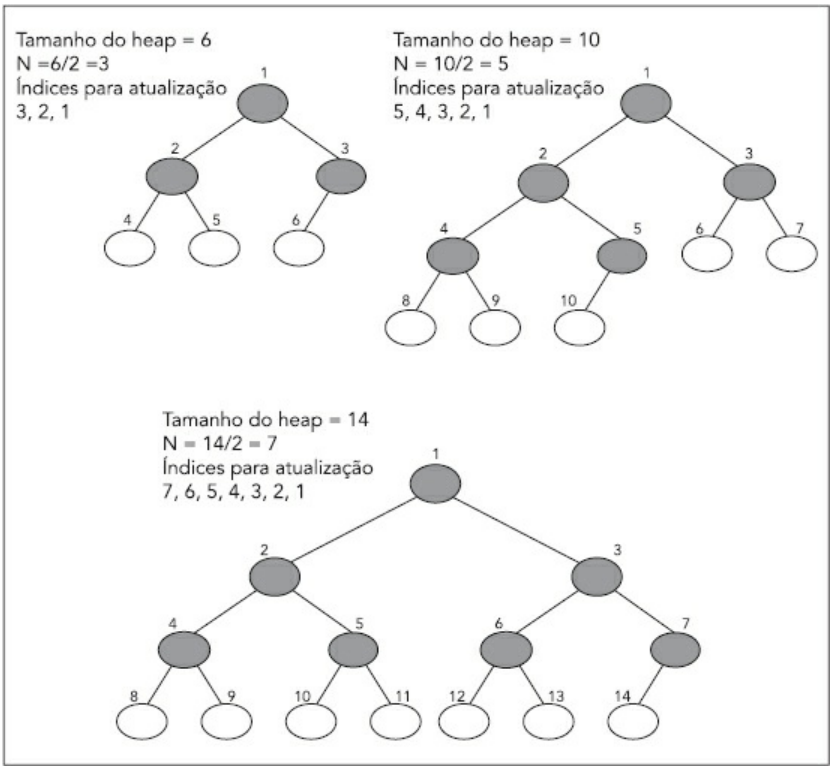


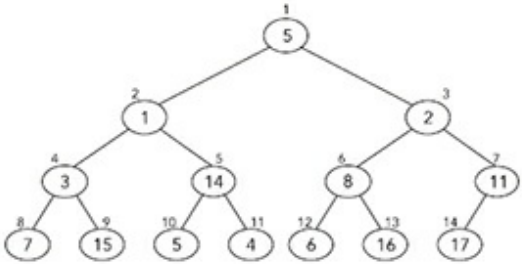
Figura 19 – Exemplo de atualização de heap máximo.
Fonte: o autor.

A partir desse contexto, permite-se a geração de um novo heap, com base em um vetor de valores não ajustados inicialmente. A figura 20 demonstra detalhadamente esse processo, inclusive destacando as situações em que o procedimento recursivo é acionado para atualização de porções internas do heap. A figura indica os elementos contendo um índice que representa a posição do elemento no vetor, sendo que o valor

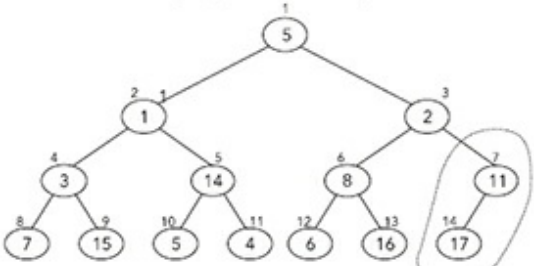
dentro de cada elipse representa o valor armazenado.

Tamanho do heap = 14
Valores inicialmente dispostos no vetor

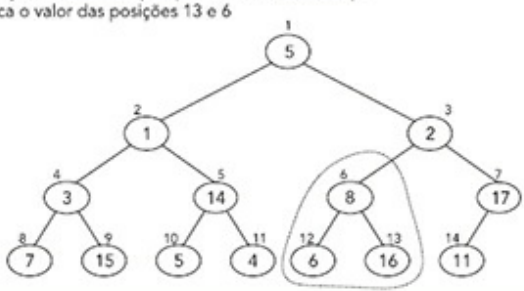
5	1	2	3	14	8	11	7	15	5	4	6	16	17



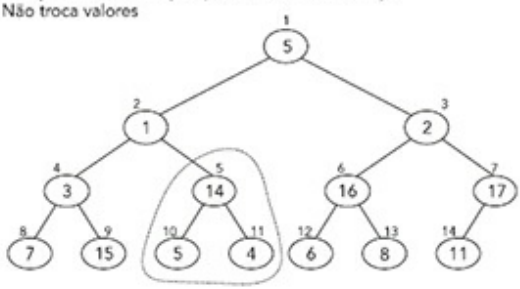
Etapa 1
Índice = 7
Compara e troca valores nas posições 7 e 14 do heap



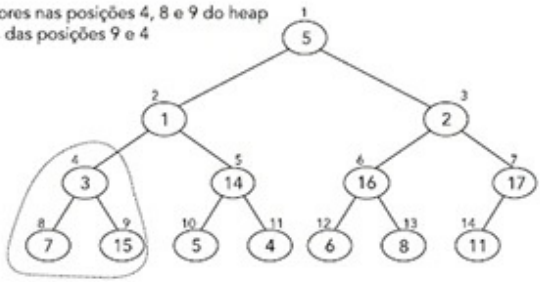
Etapa 2
Índice = 6
Compara valores nas posições 6, 12 e 13 do heap
Troca o valor das posições 13 e 6



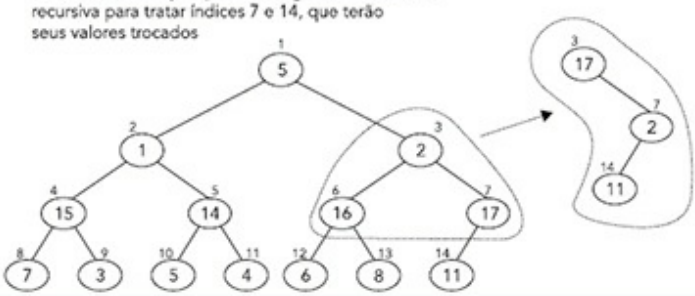
Etapa 3
Índice = 5
Compara valores nas posições 5, 10 e 11 do heap
Não troca valores



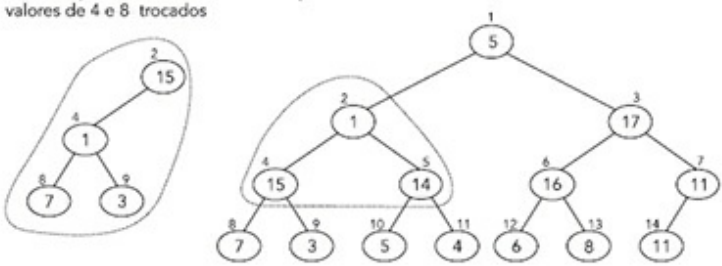
Etapa 4
Índice = 4
Compara valores nas posições 4, 8 e 9 do heap
Troca valores das posições 9 e 4



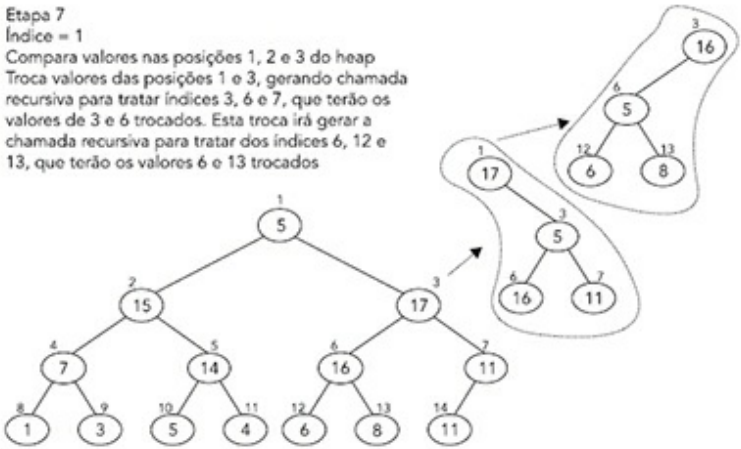
Etapa 5
Índice = 3
Compara valores nas posições 3, 6 e 7 do heap
Troca valores das posições 7 e 3, gerando chamada recursiva para tratar índices 7 e 14, que terão seus valores trocados



Etapa 6
Índice = 2
Compara valores nas posições 2, 4 e 5 do heap
Troca valores das posições 2 e 4, gerando chamada recursiva para tratar índices 4, 8 e 9, que terão os valores de 4 e 8 trocados



Etapa 7
Índice = 1
Compara valores nas posições 1, 2 e 3 do heap
Troca valores das posições 1 e 3, gerando chamada recursiva para tratar índices 3, 6 e 7, que terão os valores de 3 e 6 trocados. Esta troca irá gerar a chamada recursiva para tratar dos índices 6, 12 e 13, que terão os valores 6 e 13 trocados



Resultados finais

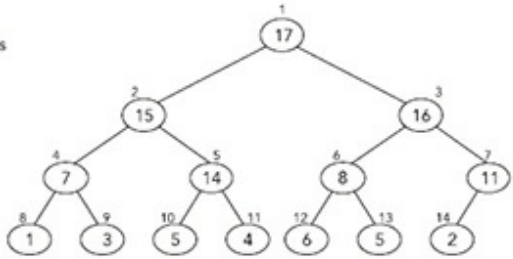


Figura 20 – Exemplo de atualização de heap máximo.
Fonte: o autor.

4.4 Algoritmo heapsort

O procedimento de organização de heap pode ser usado de forma eficiente para a geração de um procedimento de ordenação, sendo que a avaliação de performance indica a possibilidade de ordenação de acordo com um tempo de execução na ordem de O(n log n) mesmo com diferentes distribuições de dados, sendo n o número de elementos.

A abordagem utilizada emprega a troca e ajustes sucessivos de elementos de um heap máximo. Como o elemento da primeira posição é o elemento de maior valor, o algoritmo realiza a troca deste valor com o valor que se encontra na última posição do heap. Após esta troca, o número de elementos do heap é diminuído em um elemento. O procedimento de ajuste do heap irá garantir que o elemento de maior valor seja colocado na posição inicial. Com a repetição desse processo, garante-se a ordenação dos elementos inicialmente no heap.

A figura 21 descreve o pseudocódigo necessário para a implementação do algoritmo heapsort.

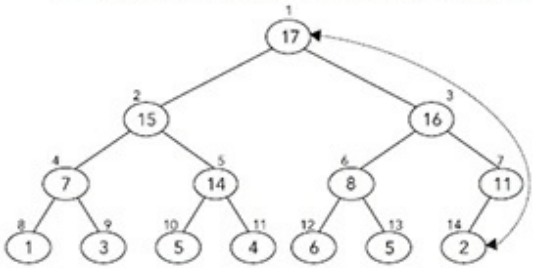
```
void HeapSort(Vetor h){
1. Constroi_Heap_Maximo(h);  // construção do HEAP Máximo
2. Para índice variando entre numero_de_elementos(h) até 2
3.       troca_elementos(h, 1, índice);
4.       diminui_tamanho_heap(-1);
5.       Atualiza_Heap_Maximo(h, 1);
6. }
```

Figura 21 – Pseudocódigo para o algoritmo heapsort.
Fonte: o autor.

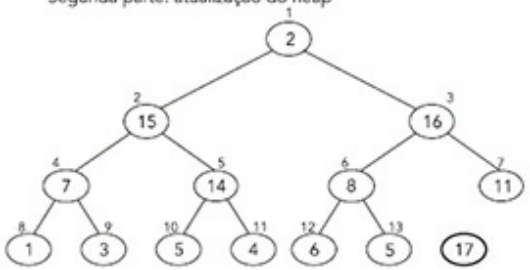
O pseudocódigo descrito na figura 21 utiliza procedimentos já descritos, como o procedimento para a construção do heap e a atualização do mesmo. O funcionamento do algoritmo pode ser observado no detalhamento da figura 22, que ilustra a sua

operação com um exemplo.

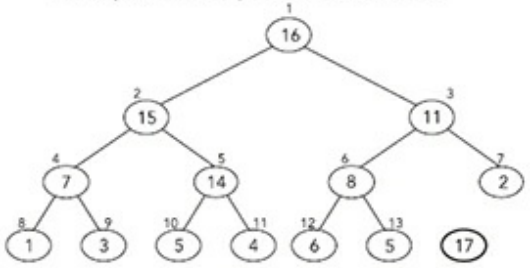
Ciclo 1
Número de elementos n = 14
Primeira parte: troca do valor 1 e n e diminui número de elementos



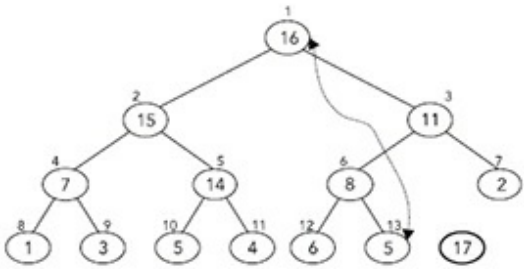
Ciclo 1
Número de elementos n = 13
Segunda parte: atualização do heap



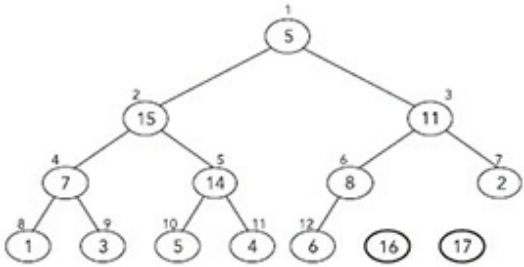
Ciclo 1
Número de elementos n = 13
Terceira parte: novo heap e elementos ordenados



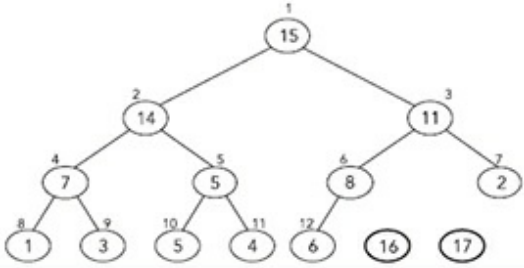
Ciclo 2
Número de elementos n = 13
Primeira parte: troca do valor 1 e n e diminui número de elementos



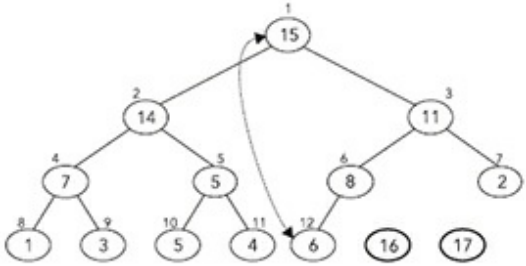
Ciclo 2
Número de elementos n = 12
Segunda parte: atualização do heap



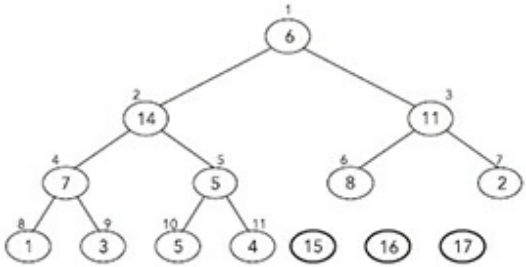
Ciclo 2
Número de elementos n = 12
Terceira parte: novo heap e elementos ordenado



Ciclo 3
Número de elementos n = 12
Primeira parte: troca do valor 1 e n e diminui número de elementos



Ciclo 3
Número de elementos n = 11
Segunda parte: atualização do heap



Ciclo 3
Número de elementos n = 11
Terceira parte: novo heap e elemento ordenado

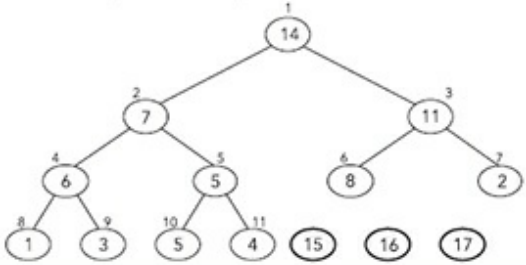


Figura 22 – Exemplo de utilização do algoritmo heapsort.
Fonte: o autor.

Apesar de o exemplo descrito na figura 22 não apresentar todo o processo de ordenação, observa-se claramente os resultados obtidos a cada ciclo, que correspondem à identificação do próximo valor, na forma da ordenação desejada. Assim, observa-se na última parte da figura 22 os seguintes elementos já ordenados: 15, 16, 17. Caso os processos descritos continuassem a se repetir, o próximo ciclo identificaria o elemento 14, depois o elemento 11, e assim sucessivamente, até a obtenção de todo o conjunto organizado de forma ordenada.

Capítulo 5

TRIEs

Em alguns contextos de recuperação de informações, em um conjunto definido de dados, é possível o tratamento prévio de dados, com operações de pré-processamento que garantem o ajuste das informações de modo a proporcionar maior eficiência nas operações de recuperação efetuadas posteriormente. A estrutura trie baseia-se em árvores e permite a localização eficiente de cadeias de caracteres e prefixos, com base em uma organização prévia dos dados. Neste capítulo serão apresentados os conceitos básicos das estruturas de dados tries e os conceitos necessários para a utilização dessa estrutura. São realizadas análises sobre suas possibilidades de utilização de forma eficiente, em especial em áreas como recuperação de informações.

Em algumas situações específicas, a maior eficiência nas operações pode ser obtida com operações conhecidas como pré-processamento, que consistem na análise do conjunto de dados e na sua organização de acordo com critérios que facilitam as operações mais comuns da aplicação. Essa é uma abordagem largamente utilizada em diversas áreas, entre elas a computação gráfica e a recuperação de informações. Baseia-se no conhecimento prévio do conjunto de dados e na sua organização prévia, ou na sua decomposição em formatos de dados adequados à execução das operações realizadas pela aplicação. Essa abordagem envolve o gasto de um tempo, por vezes grande, na etapa de pré-processamento, para a análise e organização dos dados. De acordo com o tamanho do conjunto de dados, esta tarefa pode ser dispendiosa, motivo pelo qual, em geral, essa abordagem é mais adequada para situações em que o conjunto de dados está definido e vai permanecer com pouca ou nenhuma alteração durante a sua utilização. Alguns exemplos dessas situações seriam a descrição geométrica de uma cena modelada em computação gráfica, ou um conjunto de palavras de um dicionário, ou, ainda, as informações de um determinado livro ou site da web.

Nos exemplos citados, os volumes de dados são grandes e as operações de acesso seriam bastante dispendiosas, em geral, caso não fossem organizados de modo a otimizar o tempo de execução. Como os dados tratados no pré-processamento não serão alterados, a etapa inicial, mesmo que dispendiosa, proporciona um resultado posterior bastante positivo, pois todas as operações de acesso e manipulação ocorrem em tempo adequado.

Trie é uma estrutura de dados baseada em implementações de árvores, voltada para a busca rápida de padrões em cadeias de caracteres. Considera-se, na montagem de uma árvore trie, a existência de um alfabeto e de um conjunto de palavras a serem consultadas, sendo que esse conjunto de palavras será descrito na árvore trie. As

operações de busca de padrões, ou eventualmente a busca de prefixos a partir desta organização, é bastante otimizada, podendo, em geral, ser feita com tempo de execução na ordem O("d"), sendo "d" o número de palavras do dicionário.

Alguns aspectos iniciais importantes para a compreensão de árvores trie são a existência de um dicionário, que determina os valores possíveis para os nodos da árvore. A árvore é ordenada com base no conjunto de palavras a ser representado e na utilização do alfabeto. Consideram-se dois tipos de nodo nessa árvore: nodos internos e externos. Os nodos internos são utilizados para o armazenamento das letras do alfabeto, de acordo com a estrutura das palavras representadas. Cada nodo interno pode ter de 1 até "d" nodos filhos, sendo "d" o número de letras do alfabeto utilizado. Os nodos externos indicam o final de uma palavra representada na árvore trie e em algumas implementações são utilizadas para otimizações específicas. Assim, cada cadeia de caracteres representada está associada a um nodo externo e os padrões somente serão considerados a partir de sua navegação até um nodo externo. Padrões formados por elementos de nodos internos não são considerados como cadeias de caracteres válidas para o dicionário.

Nas operações de busca de padrões, são empregadas comparações entre o padrão desejado e as cadeias de caracteres descritas na árvore trie, sendo que algumas implementações específicas como trie compactado e trie de sufixos permitem pequenas variações na performance, que pode ser bastante boa, conforme indicado anteriormente, da ordem de O(n), sendo "n" o número de letras do dicionário.

Para analisarmos a utilização de árvores trie de forma prática, vamos considerar, a seguir, um exemplo, no qual o conjunto de palavras a ser representado é o seguinte: caro, cavar, cal, cem, cela, belo, bica, bicho. A montagem de uma árvore trie, de forma intuitiva, deve iniciar com a análise das letras iniciais de cada palavra do conjunto. Identifica-se duas letras diferentes como iniciais de todas as palavras do exemplo: "c" e "b". Com base nestas duas letras, será montado o primeiro nodo interno da árvore. Depois disso, consideram-se dois subconjuntos de palavras, de acordo com as duas letras iniciais. As palavras que iniciam com a letra "c" apresentam duas opções como segunda letra: "a" em caro, cavar e cal; "e" em cem e cela. Desta forma, serão gerados dois nodos internos a partir do nodo com a letra "c", que são os nodos com estas duas letras "a" e "e". Com este mesmo procedimento, a montagem prossegue até a identificação dos nodos externos, que consistem das letras finais de cada palavra.

A figura 23, a seguir, ilustra a montagem da árvore trie simplificada, com base no conjunto de palavras indicado. Observa-se que sempre, a partir do nodo raiz, existe a possibilidade de recuperação de cadeias de caracteres que, quando seguidas até o nodo externo, formam uma das palavras representadas. Deve ser destacado que padrões formados com letras armazenadas apenas em nodos internos não são considerados como palavras do dicionário, tais como no caso de "bic" ou "bel", no exemplo da figura 23.

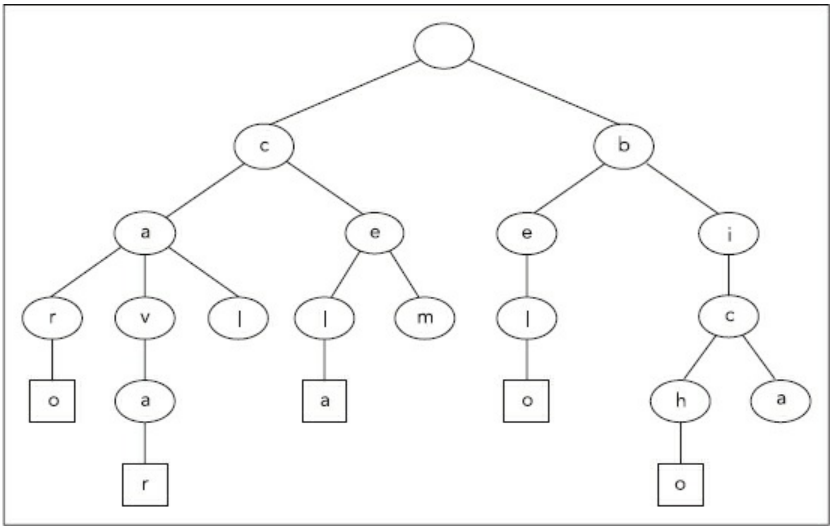


Figura 23 – Exemplo de trie para as palavras caro, cavar, cal, cem, cela, belo, bica, bicho.

Fonte: o autor.

Na figura 24 está destacado um exemplo simplificado de busca de palavras, sendo utilizada como exemplo a localização da palavra “cavar”. Para sua localização na árvore trie, parte-se do nodo raiz, com a verificação da primeira letra e este mesmo processo é repetido sucessivamente até a localização de um nodo externo com a última letra da palavra desejada.

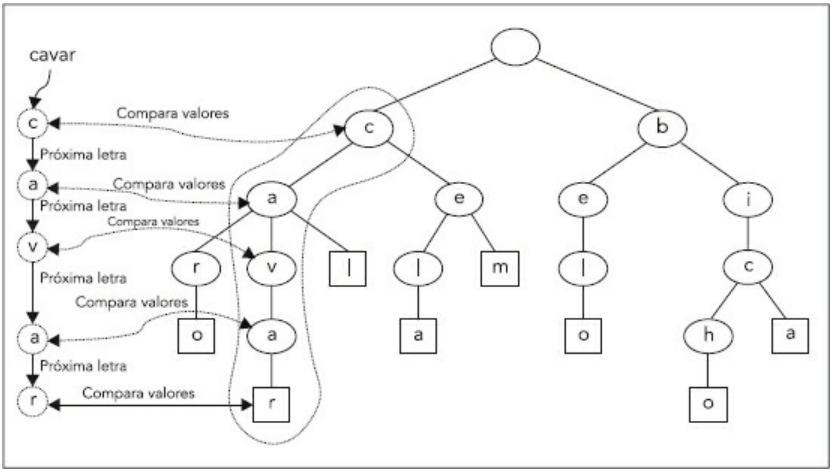


Figura 24 – Busca da palavra “cavar” em trie.
Fonte: o autor.

5.1 Tries comprimidos

Em diversas situações ocorrem possibilidades de otimização no espaço utilizado pelas árvores trie, sendo que uma destas situações é empregada na variação conhecida como trie comprimido. Para esta implementação, consideram-se os nodos definidos como redundantes, que seriam todos aqueles nodos, exceto o nodo raiz, que possuem apenas um filho e não são nodos externos. Estes nodos constituem possibilidades de otimização que implicam em economia de espaço e na melhoria de performance.

Na figura 25 está descrito um exemplo de resultado obtido com a representação comprimida de trie. Note que os nodos redundantes foram agrupados em nodos externos. Assim, os nodos com os valores “r” e “o” foram agrupados no nodo com o valor “ro”, completando a palavra “caro”. De modo similar, os nodos com os valores “v”, “a” e “r”, que estavam separados no exemplo da figura 23, foram agrupados em um nodo com o rótulo “var”, completando a palavra “cavar”.

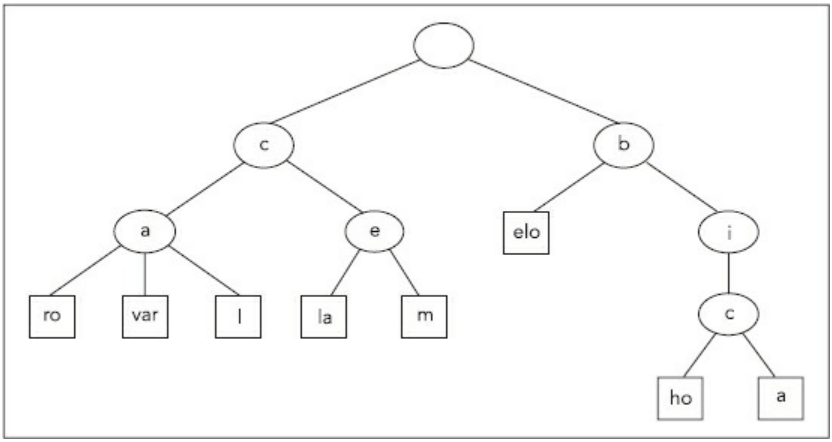


Figura 25 – Árvore trie comprimida para as palavras caro, cavar, cal, cem, cela, belo, bica, bicho.
Fonte: o autor.

O formato comprimido exibido na figura 25 serve para ilustrar o princípio geral de utilização de tries com característica de compressão, porém não implica em geração de maior eficiência diretamente. Com a compressão dos dados, os rótulos possuem tamanho maior e, além disso, tamanho variável, o que leva à diferenças no modo de utilização da árvore. Entretanto, quando utilizados com uma estrutura auxiliar, que armazene os termos a serem pesquisados e uma otimização dos valores relacionados na árvore, o processo de compressão torna-se mais eficiente.

Assim, a figura 26 descreve uma organização na qual as palavras a serem representadas na árvore trie comprimida estão armazenadas inicialmente em uma estrutura externa, consistindo de um conjunto de vetores, para que as letras de cada palavra possam ter acesso individual. No caso do exemplo, as palavras representadas anteriormente na árvore trie estão agora representadas em um vetor P com uma cadeia para cada palavra. Além desta característica, também é modificada a representação interna dos nodos da árvore, pois ao invés de armazenarem nodos com as letras ou conjuntos de letras, cada nodo representa uma tripla de valores inteiros, que representam o índice do vetor onde a palavra está armazenada e os índices internos, indicando a posição inicial e final da palavra neste vetor.

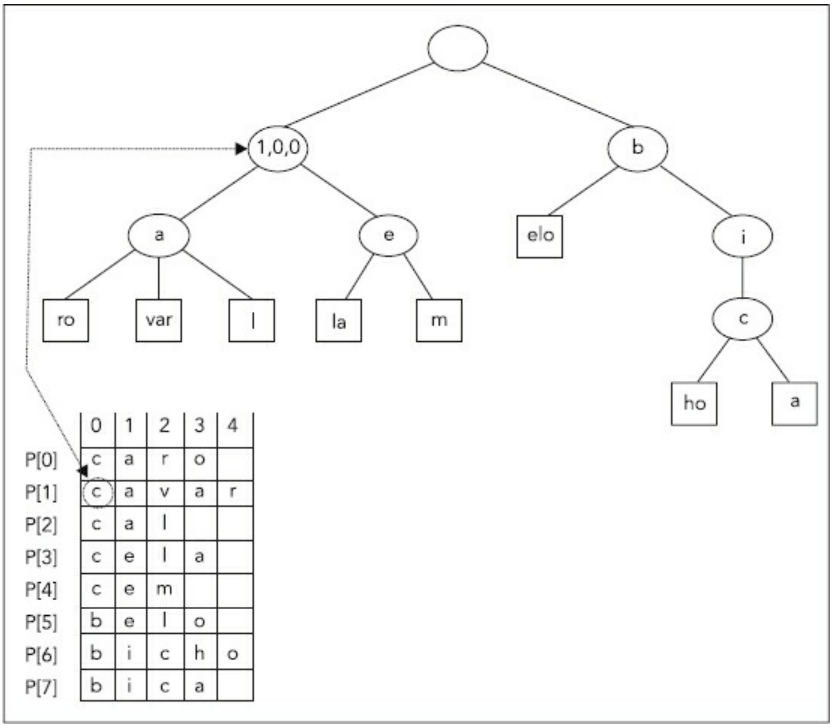


Figura 26 – Processo de montagem de árvore trie compacta.
Fonte: o autor.

O formato de exibição identificado na figura 26, onde está sendo exibido o processo de ajuste, transforma a representação anterior com caracteres em uma representação mais eficiente, com as triplas de inteiros. Note que a entrada de índice 1 do vetor P contendo as cadeias de caracteres está com a sua primeira letra destacada, associada ao nodo que anteriormente possuía a letra "c" e agora está representado com a tripla 1,0,0. Esta tripla representa, portanto, a entrada 1 do vetor de palavras, com o conteúdo desta entrada entre os índices 0 e 0. Portanto representa a letra "c".

Na figura 27, observe que todos os nodos foram ajustados conforme este processo. No exemplo foram destacados o nodo com a tripla 1,2,4, que representa o nodo contendo as letras com índice 2, 3 e 4 da entrada de índice 1 do vetor de palavras. De modo similar, o nodo com a tripla 5,0,0 está representando a entrada da

lista de palavras com índice 5, com as letras entre os índices 0 e 0, portanto a primeira letra da palavra, ou seja, a letra “b”.

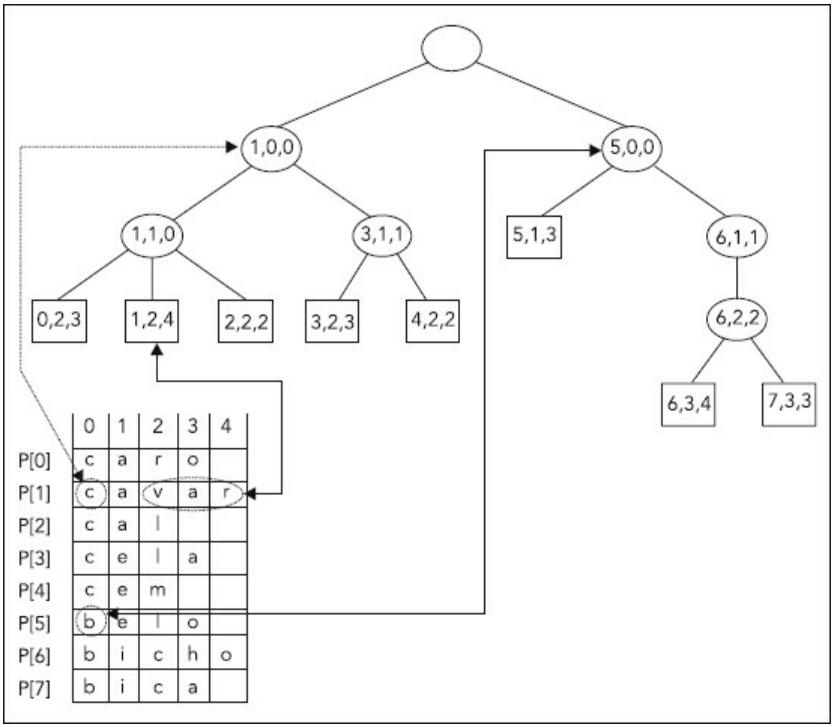


Figura 27 – Árvore trie compacta para as palavras caro, cavar, cal, cem, cela, belo, bica, bicho.

Fonte: o autor.

A representação descrita na figura 27 ilustra também algumas possibilidades de otimização em relação ao espaço utilizado por árvores trie padrão. Além disso, a localização de elementos passa a ser realizada com algumas adaptações em relação ao procedimento utilizado em árvores trie padrão.

A próxima seção descreve brevemente um formato ligeiramente adaptado de árvores trie que utiliza os recursos de representação já descritos e avança para uma configuração adequada à realização de algumas operações específicas.

5.2 Tries de sufixos

Uma das trie de sufixos é uma variação de árvores trie que permite com que todos os sufixos de palavras sejam representados, sendo que, com este recurso, algumas aplicações interessantes são possíveis de forma eficiente, tais como a localização de padrões de caracteres nas palavras representadas na estrutura de dados. Deste modo, seria possível localizar não apenas as cadeias completas de caracteres, que são a aplicação mais direta de árvores trie, mas também a localização de todas as cadeias de caracteres que possuam uma determinada sequência de letras em comum.

Além da localização das possibilidades, a árvore de sufixos permite exibi-las, de forma que possam ser utilizadas como apoio para situações diversas relacionadas ao processamento de textos, por exemplo. Outra aplicação importante de árvores de sufixos pode ser encontrada na Bioinformática, onde aplicações de busca em conjuntos de dados como DNA e proteínas podem envolver uma grande quantidade de dados e configurações bastante complexas, sendo necessária a identificação de partes de sequências destas cadeias de forma eficiente.

A árvore de sufixos pode ser implementada com os mesmos recursos descritos anteriormente, com objetivo de otimizar tanto a utilização de espaço em disco como as operações de busca de informações. Uma estrutura de árvore de sufixos possibilita a representação de todas as combinações possíveis de determinada *string* de tamanho t em uma árvore com complexidade de espaço na ordem de $O(t)$. Um fator importante é a possibilidade de geração da árvore em tempo linear, o que permite uma boa performance em situações de aplicações práticas. A busca por padrões também mostra-se eficiente, pois é possível a localização de um padrão de tamanho p em operações que envolvem uma ordem $O(p)$, sendo que a identificação de n ocorrências deste padrão de tamanho p também podem ser obtidas em operações envolvendo tempos na ordem de $O(n+p)$.

Um exemplo de árvore de sufixos está descrito a seguir, para a *string* “omooomo”, que permite avaliar alguns aspectos da geração das árvores de sufixos. A figura 28 utiliza a representação compactada de árvores trie para implementar a descrição de uma árvore de sufixos, na qual são encontradas todas as possibilidades de sufixos para a *string* utilizada como exemplo. A construção de árvores de sufixos pode ser realizada com base em diversos algoritmos, que de modo geral seguem o mesmo procedimento de análise das *strings* e subdivisão, de modo a gerar a cada passo uma ou mais opções de sufixo, até que todas as *strings* tenham sido analisadas.

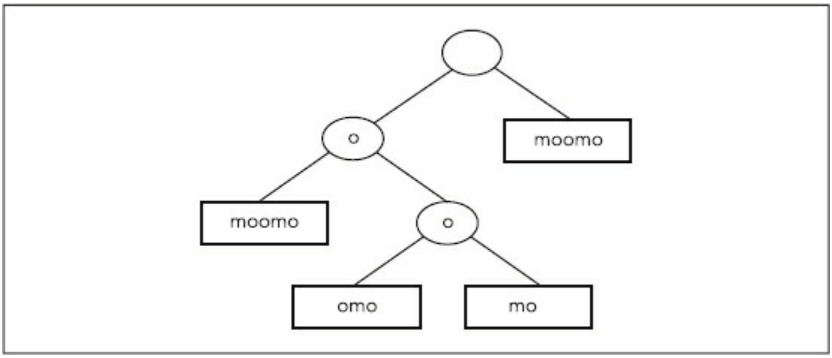


Figura 28 – Exemplo de árvore de sufixos para string “omooomo”.
Fonte: o autor.

Capítulo 6

B-TREEs

Árvores conhecidas como b-trees implementam uma abordagem eficiente para a localização de informações com uso de armazenamento secundário, tais como discos magnéticos. Esta estrutura é bastante importante em diversas situações de aplicações em que a localização de informações em grandes conjuntos de dados é necessária. Neste capítulo são descritos os conceitos básicos desta estrutura de dados e analisadas as suas aplicações e características de performance.

Árvores B (ou, em inglês, b-trees) são árvores de pesquisa balanceadas, desenvolvidas para uso em uma área específica, na qual os dados possuem um grande volume e são armazenados em dispositivos de armazenamento secundário, tais como discos magnéticos. Dada a natureza desta operação de acesso aos dispositivos de armazenamento, quando comparadas com o acesso aos dispositivos de memória principal, que é bastante lento, a estrutura de dados Árvore-B apresenta uma solução para otimizar o tempo gasto nas operações de acesso e permitir uma boa performance na localização de informações.

Portanto essa estrutura de dados possui um nicho de aplicação bem definido, sendo que diversos sistemas gerenciadores de banco de dados a utilizam como alternativa para implementação de operações de localização de informações.

Esse tipo de árvore diferencia-se de algumas outras versões de árvores balanceadas por permitir que cada nodo armazene um número muito grande de chaves, gerando, assim, uma árvore com um número alto de ramificações. O tempo de acesso pode ser descrito como na ordem O(log n), sendo que a altura da árvore é delimitada por restrições da estrutura de dados e observa uma ordem de crescimento logarítmica, de acordo com o número de chaves.

Um breve exemplo permitirá uma visão mais concreta da estrutura de dados resultante e também de suas possibilidades. Observe que cada nó desta árvore pode apresentar um número grande de chaves, sendo que existe uma regra de formação que indica que cada nodo, se possuir n chaves armazenadas, apontará para n+1 nodos filhos. Cada conjunto de chaves armazenado em um nodo da árvore B deve ser considerado como um indicador para a localização do nodo filho a ser utilizado na identificação de determinado valor. Isso ocorre porque cada valor de chave armazenado em um nodo está ordenado e os intervalos entre as chaves são utilizados como indicativos para os nodos contendo os valores do intervalo. Assim, cada intervalo entre chaves do nodo sendo manipulado está associado com um dos seus nodos filhos.

Esta organização pode ser observada de forma mais concreta no exemplo descrito na figura 29, a seguir, que descreve uma subárvore, segundo as regras de árvores B, na qual estão organizados valores numéricos ordenados de forma crescente. O nodo x possui dois valores, organizados de forma crescente (“4” e “8”) e, portanto, estas duas chaves do nodo x indicam três apontadores para nodos filhos contendo as chaves, também ordenadas de forma crescente. Isso está posto de modo que as chaves com valor inferior ao valor da primeira chave do nodo *x* estejam armazenadas no seu primeiro nodo filho, indicado na figura como o nodo *xf1*, no qual estão armazenados os valores “1”, “2” e “3”. O segundo nodo filho do nodo x está indicado na figura como *xf2* e armazena os valores “5”, “6” e “7”, que estão justamente entre os valores das duas chaves armazenadas no nodo *x*. Por fim, o terceiro nodo filho do nodo *x*, indicado na figura como o nodo *xf3*, armazena os valores ordenados acima da última chave do nodo *x*, ou seja, os valores acima de “8”, que constam no exemplo, como “9”, “10” e “11”.

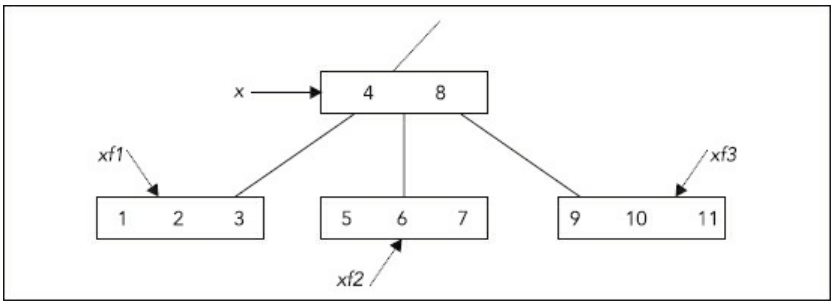


Figura 29 – Exemplo simplificado de árvore B.
Fonte: o autor.

No exemplo da figura 29, o intervalo anterior à primeira chave do nodo *x* está associado com o seu primeiro filho, contendo os valores anteriores ao valor de sua primeira chave. De modo similar, o intervalo entre as duas chaves do nodo *x* está associado com o seu segundo nodo filho, contendo os valores entre as suas duas chaves. O intervalo após a última chave do nodo *x* está associado com o seu terceiro nodo filho, contendo os valores maiores que o valor de sua última chave.

Conforme indicado, essa estrutura de dados é definida especificamente para uso com recursos de memória secundária, tais como discos magnéticos, portanto as próprias operações envolvidas em algoritmos para seu processamento estão associadas às restrições da estrutura.

Para proporcionar uma melhor compreensão desta estrutura, suas restrições e

aplicações, serão retomados alguns dos aspectos básicos do armazenamento em dispositivos secundários de memória, tais como discos magnéticos. Em geral, os discos magnéticos são compostos por um conjunto de lâminas organizadas em um eixo central, sendo que cada uma destas lâminas é acessada com um cabeçote de leitura e gravação de dados, movimentada por um braço mecânico de alta precisão. Os dados são gravados nos discos em trilhas concêntricas, de modo que um conjunto de trilhas de mesma posição, em diversas lâminas, é também chamado de cilindro, possuindo a propriedade de permitir a leitura simultânea de seus dados a partir dos diversos cabeçotes de leitura operando em conjunto.

As operações de gravação e leitura de dados são, portanto, dependentes de componentes que possuem tempo de operação bastante maior que os tempos de operação observados para o caso de componentes de memória primária, tais como circuitos de memória, que podem operar na faixa de 10 a 100 nanossegundos. No caso dos discos magnéticos, a velocidade de acesso aos dados está associada à velocidade de rotação, que determina a velocidade de localização e recuperação dos dados armazenados nas trilhas do disco. Algumas tecnologias de construção atualmente permitem que sejam utilizadas velocidades com base em variações como 5.400 RPM, 7.200 RPM, 10.000 RPM e 15.000 RPM. Estas velocidades de rotação possibilitam que o acesso a dados no disco envolvam tempos próximos de 8 milissegundos, portanto bastante maiores que os tempos verificados no acesso à memória primária.

Para um melhor aproveitamento do procedimento que envolve a leitura, com o posicionamento mecânico de cabeçotes de leitura e gravação de dados e a necessidade de rotação do disco para o acesso deste elemento à uma área de dados, as operações de armazenamento em disco magnético são organizadas em blocos localizados fisicamente, de forma contígua, em um conjunto de trilhas, possibilitando que o tempo envolvido na leitura seja, portanto, a soma do tempo de movimentação do cabeçote para a trilha desejada, seguido do tempo necessário para a rotação do disco, de modo que a área contendo os dados seja acessada pelo cabeçote de leitura e gravação de dados. A definição do tamanho dos blocos segue um compromisso entre a otimização de operações de gravação, com tamanhos extensos o suficiente para o aproveitamento das operações envolvidas, mas também levando em conta a otimização do espaço de disco, o que pode ser alcançado com tamanhos reduzidos de modo a evitar o desperdício interno aos blocos.

Em geral, busca-se uma configuração em que uma página de dados em disco envolva pelo menos uma rotação para leitura dos dados. Esta organização é configurada de modo a estar adequada com o uso dos dados pela memória principal, situação em que frequentemente é necessário o tratamento de blocos de dados maiores que o espaço físico disponível. O uso de conjuntos de dados maiores que o possível na memória principal é possível com alternância de armazenamento dos dados entre memória principal e memória secundária, de acordo com a demanda. Para que este mecanismo seja eficiente, o tamanho dos blocos transferidos e o momento de seu

processamento precisam ser cuidadosamente projetados.

No caso do uso de árvores B associadas com o tamanho das páginas armazenadas em disco, as operações de leitura e escrita devem ser organizadas para atualizar blocos por completo, otimizando as operações lentas de cópia ou leitura de dados em dispositivos secundário.

Esta necessidade pode ser atendida a partir de uma das características das árvores B, que é a possibilidade de implementação de um grande número de ramificações. Se um nodo da árvore possuir n chaves, então um número de n + 1 nodos filhos serão apontados no nível seguinte da árvore. Isso permite que uma grande quantidade de valores esteja acessível em um número pequeno de níveis, evitando que uma grande quantidade de acessos seja realizada para a recuperação de valores. Veja, por exemplo, os dados exibidos na tabela 5, que resumem a quantidade de chaves disponíveis com árvores B em dois níveis.

Tabela 5 – Relação entre o número de chaves por nodo, altura da árvore B e quantidade total de chaves

Chaves por nodo	Profundidade	Nodos	Total de chaves
10	nodo raiz	1	10
	1	11	110
	2	121	1.210
100	nodo raiz	1	100
	1	101	10.100
	2	10.210	1.020.100
1000	nodo raiz	1	1000
	1	1.001	1.001.000
	2	1.002.001	1.002.001.000

Fonte: o autor.

Conforme a descrição na tabela 5, observa-se que a quantidade de valores armazenados cresce rapidamente. No caso de apenas 10 chaves por nodo, chega-se a um número de 1.210 chaves em uma árvore B de dois níveis. Este número passa para mais de um milhão no caso de aumentar o número de chaves para 100 por nodo. Por fim, com 1.000 chaves por nodo, apenas dois níveis de representação garantem mais de um bilhão de valores. Esse exemplo destaca a possibilidade, portanto, de acesso a mais de um bilhão de chaves diferentes com apenas dois acessos diferentes ao

armazenamento em memória secundária, um dos pontos importantes desta estrutura de dados.

6.1 Definições e operações básicas

Alguns detalhes de implementação serão destacados a seguir, de modo a formalizar a descrição de árvores B. Alguns modelos de implementação consideram que todas as informações associadas com as chaves são armazenadas no mesmo nodo em que esta se encontra, como forma de otimizar operações de acesso. Já uma variação dessa implementação, conhecida como árvore B^+, armazena as informações associadas apenas nos nodos folha da árvore, com o objetivo de otimizar a localização dos nodos, armazenando nos nodos internos os ponteiros para essas chaves. Em alguns casos, também são utilizados ponteiros para indicar o nodo localizado à direita do nodo atual, em um mesmo nível da árvore.

Alguns itens são formalizados a seguir, em relação ao modelo mais geral de implementação de árvores B. A partir da existência de uma raiz para a árvore, todos os nodos da mesma possuem elementos de composição, tais como o número de chaves armazenadas, os valores das chaves (e, dependendo da implementação, as informações associadas), um valor descrevendo a característica do nodo, de forma a diferenciar nodos internos e nodos folha. Os valores das chaves são armazenados no nodo de forma ordenada, não decrescente. Além disso, o nodo armazena um número de ponteiros para os nodos filhos, composto pelo número de chaves mais um. Esses ponteiros para nodos filhos são organizados de modo que as chaves armazenadas em cada nodo filho estejam no intervalo dos valores das chaves do nodo atual.

Esses detalhes podem ser observados na figura 30, abaixo, na qual está ilustrada uma situação em que a árvore B é implementada com duas chaves por nodo. Em cada um dos nodos observam-se as duas chaves e os ponteiros para os nodos filhos. Neste caso, como o nodo possui duas chaves, apontará para três nodos filhos. Os valores de cada nodo filho seguem a definição de ordenação não decrescente, portanto, conforme indicado na figura, abaixo do diagrama, as chaves “C´1” e “C´2” estarão ordenadas de forma não decrescente entre si e também serão menores ou iguais à chave “C1”. Já as chaves “C´´1” e “C´´2” estarão entre as chaves “C1” e “C2”. Por fim, as chaves “C´´´1” e “C´´´2” estarão ordenadas com valores maiores que o valor da chave “C2”.

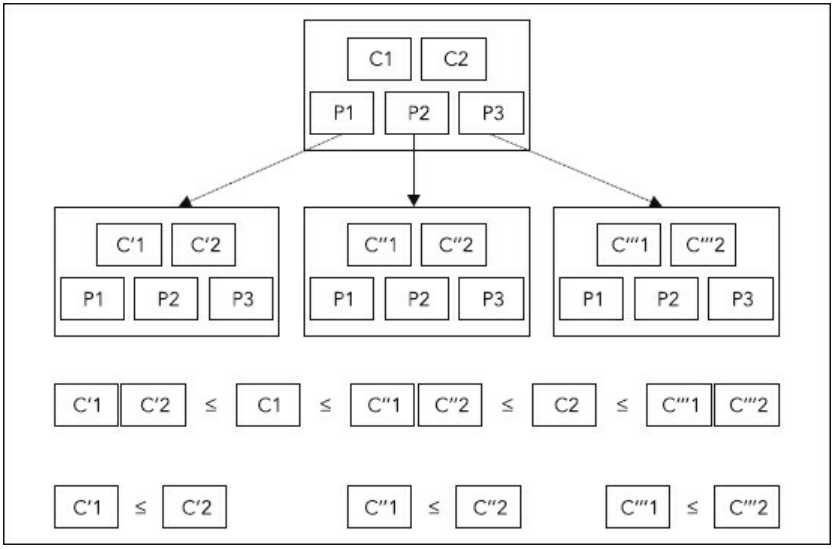


Figura 30 – Detalhamento de árvore B com duas chaves por nodo.
Fonte: o autor.

Para garantir a sua operação de forma otimizada, são observadas também algumas restrições, mencionadas a seguir. Todas as folhas de uma árvore B possuem exatamente a mesma altura nessa árvore, cujo valor equivale à própria altura da árvore. Cada nodo pode armazenar um número variado de valores de chaves, dentro do limite descrito inicialmente pela implementação. Porém, mesmo dentro deste contexto, existem limites mínimos e máximos que devem ser observados para o número de chaves por cada nodo da árvore. Esses limites são delimitados a partir de um número conhecido como o grau mínimo da árvore, que deve ser obrigatoriamente maior do que o valor 2. Caso o grau mínimo for representado por *M*, então todos os nós diferentes da raiz da árvore devem possuir pelo menos *M*-1 chaves e no máximo 2*M*-1 chaves. Ou seja, caso o valor de *M* para dada árvore seja igual a 2, seus nodos podem ter entre 1 e 3 chaves cada. O objetivo do controle de um intervalo de valores armazenados em cada nodo permite que as operações de acesso e recuperação de dados sejam mantidas otimizadas.

A busca em árvores B é bastante similar à busca em árvores binárias. O algoritmo geral para o procedimento pode ser descrito a partir de uma pesquisa no nodo raiz da árvore B, por uma determinada chave, cujo valor será utilizado para comparação entre

os valores de chave armazenados no nodo. Caso a chave não seja encontrada neste nodo, seu valor será utilizado para determinar o índice do ponteiro para o nodo filho que deve ser localizado. Essa operação pode envolver a busca em dispositivo de armazenamento secundário. Tal procedimento pode ser implementado de forma recursiva, tendo como pontos de parada a localização, o valor da chave ou a não localização do mesmo em um nível dos nodos folha, situação em que a pesquisa deve ser concluída sem sucesso.

CAPÍTULO 7

GRAFOS

Neste capítulo serão apresentados os conceitos básicos de grafos, bem como as principais operações e algoritmos associados a estas estruturas. Os grafos possibilitam que uma grande quantidade de problemas seja modelada computacionalmente e que soluções eficientes sejam desenvolvidas. Tanto problemas associados com situações de implementação, levando em conta aspectos reais de certas áreas, como problemas puramente abstratos podem ser modelados com essa estrutura. Ao longo do capítulo, também serão destacados diversos exemplos de utilização, em situações que permitam ao leitor uma visão geral e abrangente de suas possibilidades.

A estrutura de dados conhecida como grafos apresenta uma grande flexibilidade para a modelagem de situações e, além disso, possibilita a geração de operações eficientes, com base nas possibilidades de implementação. Desta forma, é natural que sejam encontradas aplicações bastante numerosas utilizando essa estrutura como base, dada a sua versatilidade e capacidade de modelagem de problemas complexos.

Possibilidades de aplicação de grafos podem ser estimadas pelo leitor a partir de alguns exemplos de situações típicas, nas quais estes podem ser utilizados de forma eficiente: modelagem de mapas entre localizações, descrição de redes de comunicação, modelagem de cenários de jogos de entretenimento, descrição e simulação de processos complexos envolvendo subprocessos e execução simultânea de etapas, simulação de roteamento urbano, otimização de projeto de placas de circuito integrado, recuperação e comparação de imagens, análise e simulação de centrais hidrelétricas.

Uma das grandes vantagens do uso de grafos para modelagem de problemas complexos está na sua capacidade de abstração de elementos destes problemas e de relacionamento dos diversos elementos entre si, de forma flexível e dinâmica. A seguir, serão descritos os elementos básicos de grafos e as formas de representação.

7.1 Estrutura e terminologia

Um grafo é definido, de forma intuitiva, como um conjunto de objetos e ligações entre estes. Pode também ser descrito, matematicamente, como sendo formado por dois conjuntos, denominados de conjunto de nodos (ou nós, ou ainda, vértices) e arestas (ou conexões, ou ainda, arcos). As arestas conectam os nodos entre si, permitindo que qualquer nodo seja conectado com qualquer outro.

Uma notação comumente utilizada para a descrição dos grafos é a seguinte: G = (V, A), onde V representa o conjunto de vértices e A representa o conjunto de arestas. A figura 31 identifica dois grafos diferentes, ambos contendo letras identificando cada vértice. Observe que a lista de arestas de cada um dos exemplos organiza os vértices em formatos que devem ser familiares, pois são os formatos observados em árvores e listas. Desta forma, o exemplo serve para indicar que as estruturas de dados, tais como árvores ou listas, podem ser consideradas como grafos gerados a partir de determinadas restrições quando às ligações entre vértices e arestas.

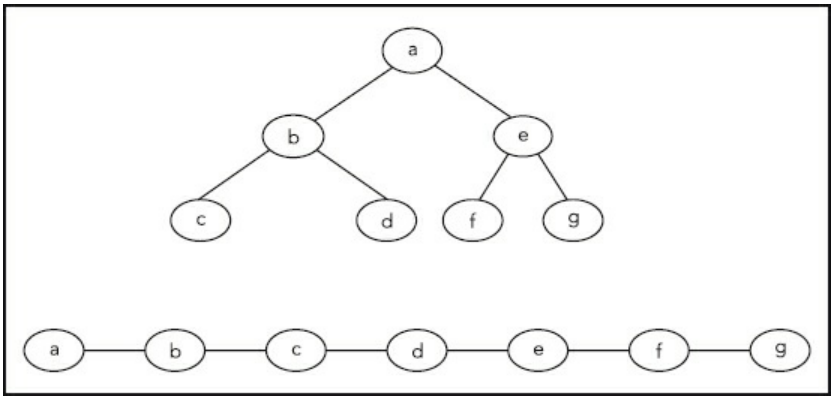


Figura 31 – Dois exemplos de grafos, similares à estruturas de árvores e listas.
Fonte: o autor.

A grande flexibilidade encontrada na estrutura de dados com base em grafos está relacionada com a utilização de vértices e arestas para representar objetos quaisquer e para indicar ligações necessárias entre esses objetos. Portanto, seguiremos com a exposição de maiores detalhes sobre os componentes de um grafo.

Os vértices de um grafo (também referidos como nós ou nodos) podem representar elementos de um problema real, com base em informações simples, como valores numéricos, dados alfanuméricos ou ainda em organizações mais complexas de dados. Cada vértice pode também ser rotulado com nomes para facilitar o acesso e a utilização.

As arestas (também referenciadas como arcos ou conexões) indicam o relacionamento entre os vértices, sendo que o número de arestas originada em um vértice, originalmente, não possui limitações, com exceção daquelas definidas pelo problema sendo modelado. As arestas podem ser orientadas ou não, ou seja, podem ou não indicar um ponto de origem e outro de destino. Um vértice pode receber uma

ligação para ele próprio, quando uma aresta possui origem e destino no mesmo vértice. Além disso, as arestas podem ser utilizadas para indicar atributos, úteis para modelagem de alguns problemas.

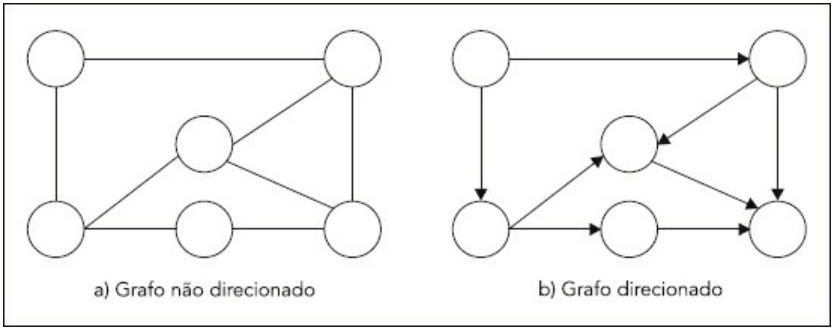


Figura 32 – Grafo não direcionado e grafo direcionado.
Fonte: o autor.

Na figura 32 estão exemplificados dois aspectos na definição de grafos. Na primeira parte da figura (a), está exemplificado um grafo não direcionado, no qual as arestas apenas indicam a conexão entre os vértices do grafo. Já no segundo grafo (b), também conhecido como dígrafo, cada aresta possui uma indicação de sentido, indicando a conexão com base em uma informação de origem e destino.

Já na figura 33, abaixo, os mesmos dois grafos receberam rótulos, que permitem a descrição precisa de sua composição. Deste modo, seria possível descrever os grafos abaixo a partir da indicação de um conjunto de vértices V = (a, b, d, e, f, g) e de um conjunto de arestas A = {(a, b), (a, e), (b, d), (b, f), (d,e), (g, d), (g, f), (g, e)}.

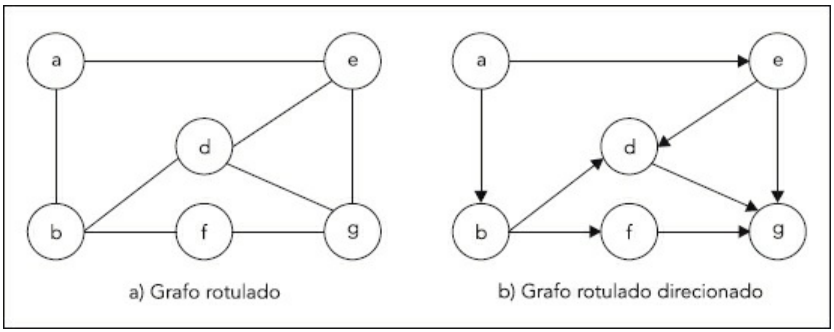


Figura 33 – Grafo rotulado e grafo direcionado.
Fonte: o autor.

No caso de um grafo direcionado, como no item b da figura 33, devem ser seguidas algumas orientações gerais na descrição do mesmo, para representar corretamente as ligações entre os vértices e o seu sentido. Por exemplo, supondo que a descrição das arestas em um grafo direcionado seja descrita com a indicação da aresta de origem seguida da aresta de destino, deveria ser ajustada a descrição anterior, para indicar corretamente as arestas entre os seguintes vértices: "e" e "d", "d" e "g", "f" e "g". No exemplo de descrição anterior, essas arestas não estão indicadas de acordo com a origem e destino das setas. Desta forma, o conjunto de arestas deve ser descrito da seguinte forma: A= {(a, b), (a, e), (b, d), (b, f), (e, d), (e, g), (d, g), (f, g)}.

Uma das aplicações desse tipo de grafo pode ser o de orientação de trânsito, por exemplo. Neste caso, pode-se supor que cada vértice do grafo indica um cruzamento em uma cidade e que as arestas indicam o sentido em que é permitido o tráfego de veículos. Desta forma, para identificar as possibilidades de roteiros entre os vértices "a" e "g" seria possível, intuitivamente, utilizarmos o grafo e as arestas para identificar os possíveis caminhos, representados pelos vértices conectados no sentido desejado. Para tal, deve-se partir do ponto de origem, ou seja, o vértice de rótulo "a", e devem ser examinadas as arestas e os vértices de destino das mesmas, de forma sucessiva, até o momento em que seja detectado como vértice de destino de uma das arestas o vértice indicado como destino final do roteiro desejado.

Este processo é também conhecido como caminhamento em grafos e permitiria, por exemplo, identificar a sequência de vértices (a, b, d, g) como uma das soluções para o problema. Esta sequência está destacada na figura 34 abaixo. A solução completa, com todos os itens desse conjunto de caminhos, pode ser descrita como S = {(a, b, d, g), (a, e, g), (a, e, d, g), (a, b, f, g)}.

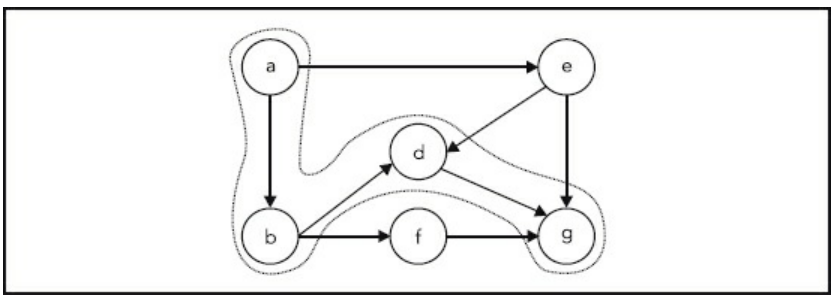


Figura 34 – Utilização de grafo direcionado para identificação de roteiros.

Fonte: o autor.

Além da identificação de roteiros, pode ser importante para alguma classe de aplicações identificar também o menor caminho, dentre os roteiros identificados como possíveis. Uma solução para esse tipo de abordagem é a utilização de pesos para as arestas, de modo que elas identifiquem as grandezas necessárias, tais como, neste caso, a distância entre os vértices. Este tipo de grafo, como pesos para as arestas, é dito grafo ponderado. No exemplo da figura 35 estão identificadas, para cada aresta, as distâncias entre os vértices. Desta forma, a caminho identificado na figura 35 envolve os nodos "a", "b", "d", e "g", sendo que a soma das distâncias para estas arestas é igual a 350 (200 + 100 + 50). A menor distância entre os quatro caminhos descritos está associada ao caminhamento entre os vértices "a", "e", "d" e "g", que é igual a 110 (50 + 10 + 50).

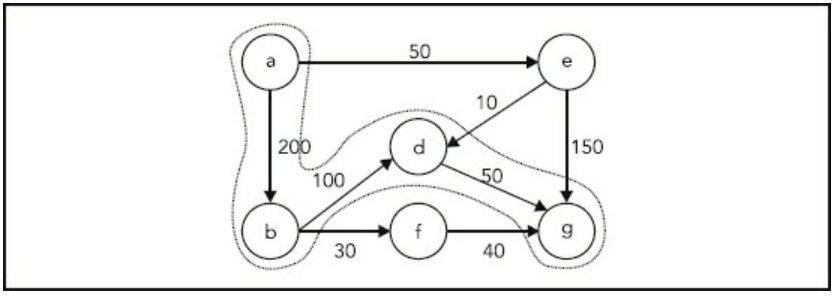


Figura 35 – Utilização de grafo com pesos para identificação de menor roteiro.
Fonte: o autor.

Um caminho é uma sequência de vértices conectados entre si por arestas, sendo considerado o tamanho do caminho igual ao número de arestas que o compõe. Considera-se um ciclo em um grafo quando um caminho de tamanho maior ou igual a três conecta o vértice inicial. De acordo com esta descrição, portanto, um grafo pode descrever ciclos, a partir das opções de navegação pelas arestas. Quando o grafo não possui ciclos, é denominado grafo acíclico.

Quando um grafo possui pelo menos um caminho entre todos os vértices, ele é dito um grafo conectado. Um grafo é dito completo quando todos os seus vértices estão interligados com pelo menos um caminho. Quando dois vértices são conectados por uma aresta, eles são ditos vértices adjacentes, ou vizinhos. Um grafo também pode apresentar vértices sem nenhuma conexão com os demais, tais como os vértices "e", "f" e "g" da figura 36. Também podem existir vértices conectados com eles

próprios, como no caso do vértice “c” do exemplo descrito na figura 36. Outro conceito importante é o de subgrafos, tal como pode ser considerado o conjunto formado pelos vértices “b”, “c” e “d” da figura 36.

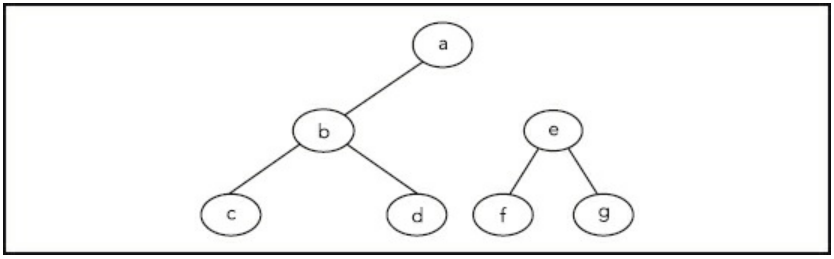


Figura 36 – Utilização de grafo com pesos para identificação de menor roteiro.
Fonte: o autor.

7.2 Representação de grafos

A representação de grafos é usualmente realizada com base em dois formatos gerais, que possuem maior ou menor adequação de acordo com as necessidades de aplicações. Existem diversas implementações com variações para melhoria de performance ou economia de espaço, mas, em geral, são derivadas das duas opções gerais: a representação por matrizes de adjacência e a representação por listas de adjacência.

A representação por matrizes de adjacência é normalmente mais adequada para situações onde os grafos são densos, ou quando operações específicas dos algoritmos utilizam detalhes desta representação. Para representar um grafo com número de vértices $n = |V|$, utiliza-se uma matriz de n x n posições. Os vértices devem ser identificados com índices variando entre 0 e n-1. Cada linha l da matriz deve representar todas as informações relacionadas com o vértice l. Assim, dada uma linha l, a coluna c irá receber um valor positivo caso exista uma aresta entre os vértices v_l e v_c. Um valor nulo na posição (l, c) da matriz indica a não existência de uma aresta entre estes dois vértices. Quando apenas a existência da aresta entre os dois vértices deve ser indicada, a matriz pode ser definida com um espaço de apenas 1 bit para cada uma das posições. Quando trata-se da representação de grafos ponderados, onde devem ser armazenados valores para cada aresta, a matriz deve armazenar um valor nulo para indicar que não existe aresta ou então, caso exista a aresta, deve armazenar o valor da mesma.

Como exemplo, a figura 37 descreve um grafo, o conjunto de vértices e o conjunto

de arestas. Com base no conjunto de arestas, é gerada a matriz de adjacências, de modo que para cada aresta entre dois vértices seja armazenado o número 1 na posição correspondente da matriz. Os vértices podem ser referenciados pelos rótulos, ou pelo seu índice na lista de vértices (V). Assim, a lista de arestas pode ser indicada também com uso dos rótulos ou dos índices dos vértices envolvidos, como na representação da aresta entre o vértice de rótulo "a" e o vértice de rótulo "e", que podem ser indicados também pelos respectivos índices na lista de vértices, ou seja, 0 e 3. Essa aresta será representada como o valor 1 na posição da matriz indexada pelos valores dos índices dos vértices, ou seja, na linha 0 e coluna 3.

Deve ser destacado que, neste caso, o exemplo trata de um grafo não direcionado e que não utiliza pesos para as arestas. Desta forma, deve ser representada tanto a aresta do vértice "a" ao vértice "e", como a aresta entre estes dois vértices, na direção contrária ("e", "a"). Este fato gera a descrição de duas representações por aresta na matriz de adjacências, abordagem que pode ser modificada de acordo com detalhes da implementação sendo desenvolvida ou do algoritmo que utilizará a estrutura de dados gerada.

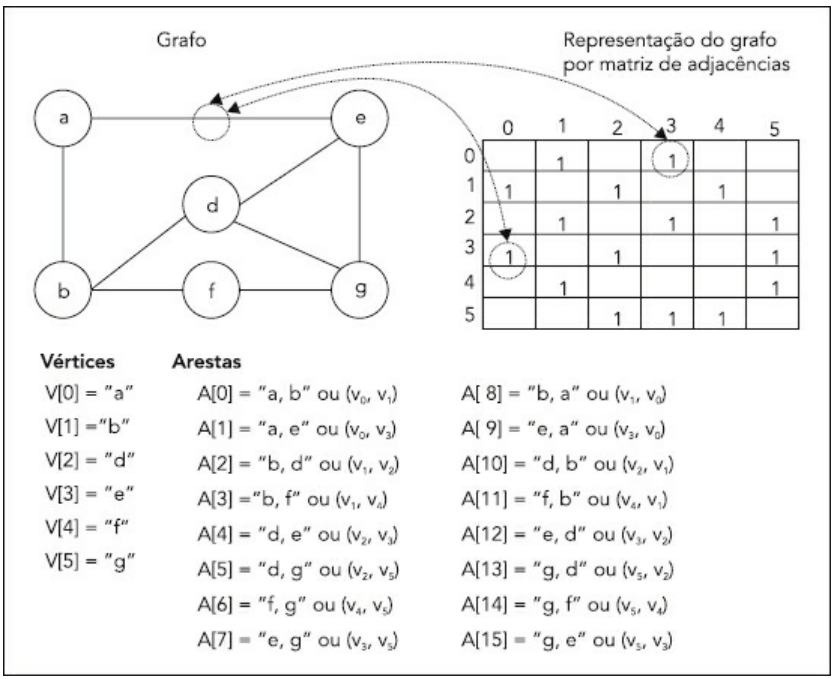


Figura 37 – Representação de grafos por matrizes de adjacência.
Fonte: o autor.

Como descrito anteriormente, em alguns casos é importante a representação de um grafo com arestas direcionadas e contendo pesos ou atributos. Deste modo podem ser representadas correta e eficientemente algumas aplicações.

No caso de uma aplicação demandar um grafo ponderado e direcionado, a representação por matrizes de adjacência também pode ser utilizada para a sua descrição, com pequenas alterações no formato. Em primeiro lugar, deve ser determinado um padrão para indicar a origem e o destino de cada aresta, o que pode ser facilmente resolvido com determinações simples, tais como a indicação de que o índice da matriz relacionado à linha indicará o vértice de origem da aresta e o índice da coluna indicará o vértice de destino. Assim, uma posição qualquer (l, c) na matriz será interpretada como representando a aresta com origem no vértice de índice l, cujo destino é o vértice de índice c. A ponderação pode ser indicada com os valores armazenados na matriz. Por exemplo, pode ser adotada outra definição simples, indicando que valores nulos indicam a não existência de arestas entre os vértices representados pela posição da matriz de adjacência. Já no caso de uma aresta, o seu valor será armazenado diretamente na matriz, cumprindo tanto a função de indicação de existência da aresta como a função de descrição do valor da mesma.

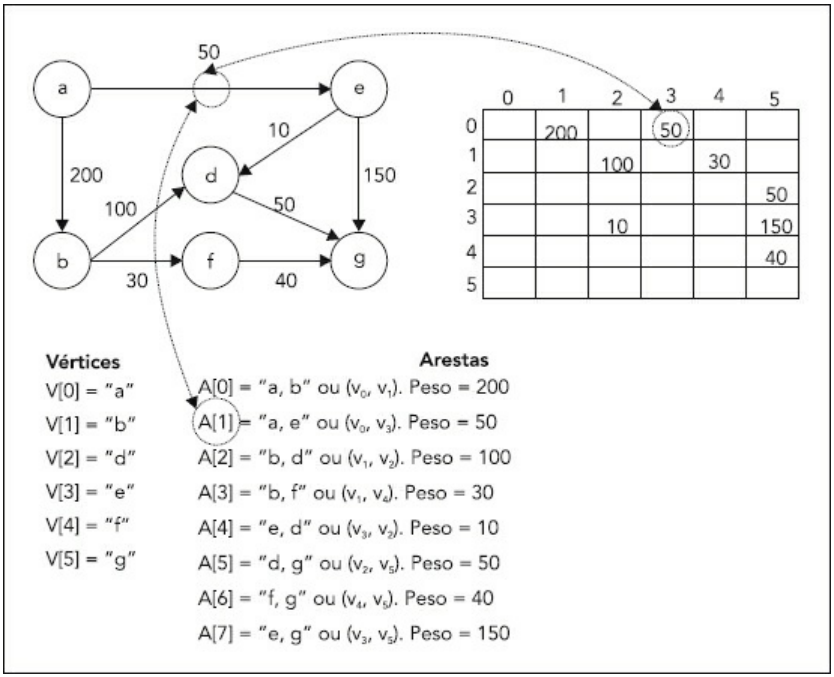


Figura 38 – Representação de grafos ponderados e direcionados, com matrizes de adjacência.

Fonte: o autor.

Na figura 38 está ilustrado um grafo ponderado e direcionado, sendo que a listagem de suas arestas inclui o peso de cada uma. Foi adotada, neste exemplo, a convenção de indicação da origem da aresta com o índice da linha e destino com o índice da coluna da matriz. Assim, por exemplo, a aresta de índice 1 na lista de arestas (A[1]), que representa uma aresta direcionada do vértice "a" (índice 0 na lista de vértices, V[0]) para o vértice "e" (índice 3 na lista de vértices, V[3]) será representada indicando o seu peso (valor 50) na posição (0,3) na matriz de adjacência.

Outra forma de representação de grafos parte do uso de listas, com isso apenas as arestas existentes são representadas, o que permite economia de memória em alguns casos. Com esta forma de representação, cada aresta é representada como um elemento em uma lista, sendo utilizadas as propriedades das listas para o tratamento das situações descritas anteriormente, para o caso de representação em matriz de

adjacência. Assim, a representação de grafos direcionados ou não direcionados pode utilizar as listas com alguma definição prévia, indicando, por exemplo, no caso de grafos direcionados, que sempre a origem da aresta está descrita no nodo inicial da lista. Grafos ponderados podem ser representados armazenando-se os valores nos próprios elementos da lista, com base na descrição de tipos de dados adequados a cada situação.

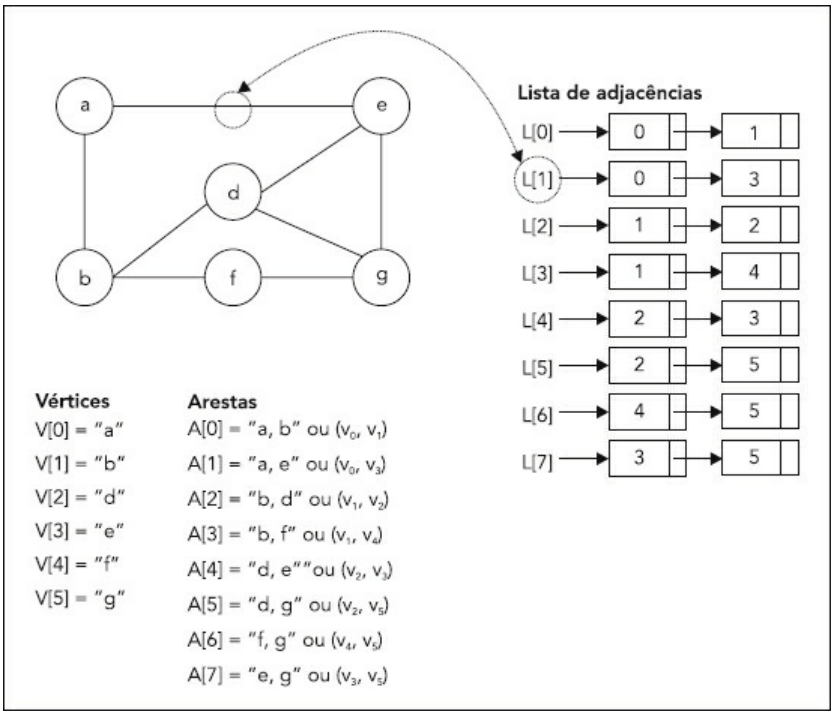


Figura 39 – Representação de grafos com listas de adjacência.
Fonte: o autor.

A figura 39 ilustra um exemplo de representação de grafo não direcionado com base em listas de adjacência. Neste caso foi definido o uso de um elemento de um vetor de listas para representar cada aresta do grafo, assim o primeiro elemento do vetor L representa a aresta entre os vértices de índice 0 e 1 (“a”, “b”) e o segundo elemento deste vetor representa a aresta entre os vértices de índice 0 e 2 (“a”, “d”).

Casos de representação de grafos ponderados também podem ser tratados com esta abordagem, como no exemplo da figura 40, em que o elemento da lista de adjacências foi modificado para conter um espaço numérico que pode ser utilizado como valor da aresta.

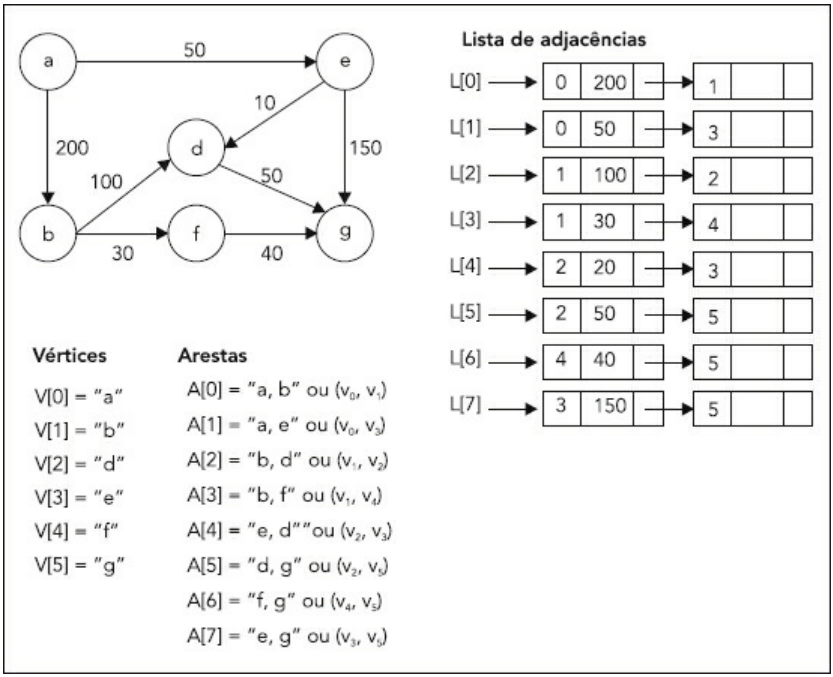


Figura 40 – Representação de grafos ponderados com listas de adjacência.
Fonte: o autor.

7.3 Caminhamento em grafos

O caminhamento em grafos pode ser necessário em situações nas quais valores devem ser localizados ou alguma outra situação deve ser verificada. Em algumas aplicações em que estados e relações entre estes são representados por grafos, o caminhamento permite a obtenção da solução de determinados problemas. Algumas situações relacionadas com a descrição dos grafos devem ser tratadas, ocasionalmente,

sendo que tais situações estão relacionadas com os grafos cíclicos e com os grafos não conectados.

Quando um grafo possui ciclos, o algoritmo de caminhamento deve prever esta situação e evitar a repetição de caminhos já visitados durante caminhamentos prévios. No caso de grafos não conectados, existe a necessidade de verificação diferenciada, que trate também os elementos ou eventualmente os subgrafos não conectados e não apenas os resultados obtidos com o caminhamento para a resolução dos problemas de varredura de seus elementos.

Em geral, existem duas abordagens largamente utilizadas: o caminhamento em profundidade e o caminhamento em largura. De acordo com a natureza dos problemas tratados e da configuração dos grafos é que deve ser definida a melhor escolha para o caminhamento. Os dois procedimentos compartilham alguns aspectos. Em geral, parte-se do vértice inicial do grafo ou subgrafo sendo analisado e repete-se o procedimento de verificação de vértices vizinhos e escolha de uma sequência de visitas a estes vértices. Para a identificação dos elementos já analisados, tendo em vista os cuidados necessários com grafos cíclicos ou com grafos não conectados, em geral utiliza-se um elemento auxiliar para cada vértice, que indica se ele já foi analisado ou não. Esta informação pode ser apenas um bit, indicando uma situação mais simples (visitado, não visitado) ou pode ser outro parâmetro, no caso de situações que exigem o armazenamento de mais de um estado para cada vértice.

Os algoritmos de caminhamento devem inicialmente definir os valores de verificação como nulos, para garantir que todos os vértices serão corretamente analisados. Durante os procedimentos de caminhamento, cada operação de acesso a um vértice deve ser registrada, marcando-se o valor de verificação relativo ao mesmo. Durante a verificação, os valores podem ser consultados, evitando as situações de laço infinito, no caso de grafos cíclicos. Além disso, esses valores podem ser utilizados ao final dos processos de caminhamento, para garantir que todos os vértices tenham sido consultados, o que pode não ocorrer no caminhamento em grafos não conectados.

De acordo com o modelo de representação dos grafos e dos algoritmos sendo desenvolvidos, podem ser utilizadas diferentes alternativas para a manutenção dos valores de verificação. No caso do uso de matrizes de adjacência, por exemplo, podem ser utilizadas matrizes auxiliares, contendo o mesmo tamanho da matriz de representação do grafo, apenas para a manutenção dos valores de verificação. Já com o uso de listas de adjacência, os próprios elementos das listas podem ser adaptados para armazenarem informações sobre o acesso aos elementos do grafo. A seguir, estes dois mecanismos serão detalhados e exemplificados.

No caminhamento em profundidade, o grafo será visitado com base em um vértice inicial e na escolha de um dos vizinhos deste vértice, até que não exista mais possibilidade de acesso a novos vértices vizinhos, seja pela inexistência de arestas ou por já ter ocorrido a visitação de todos os vizinhos do vértice atual. Podemos

identificar esta situação como um ponto de parada do caminhamento. A cada etapa é escolhida uma aresta indicando um vértice vizinho a visitar, mas também é armazenado o vértice atual em uma pilha. Quando o caminhamento identificar um ponto de parada, será consultada a pilha e o vértice disponível no topo desta pilha será usado para a continuidade do processo. O caminhamento termina quando todos os vértices tiverem sido visitados. Este comportamento gera um resultado equivalente ao caminhamento por todos os vizinhos de um determinado ramo do grafo, seguido do retorno a vértices com vizinhos não visitados e da repetição do processo. Esta é, portanto a origem do nome do caminhamento, pois a dimensão de profundidade é privilegiada.

No caminhamento em largura, o grafo será visitado de forma ligeiramente diferente, com o esgotamento de todas as arestas de um determinado vértice antes da continuidade do caminhamento. Com isso, obtém-se como resultado um caminhamento por todos os vizinhos de um vértice inicial, seguindo níveis de aprofundamento segundo os níveis de conexão existentes. O nome desse caminhamento está associado com esse resultado, ou seja, em primeiro lugar, todos os elementos de cada nível no grafo serão analisados.

Veja no exemplo da figura 41 os diferentes resultados obtidos com os dois formatos de caminhamento do grafo empregado como exemplo.

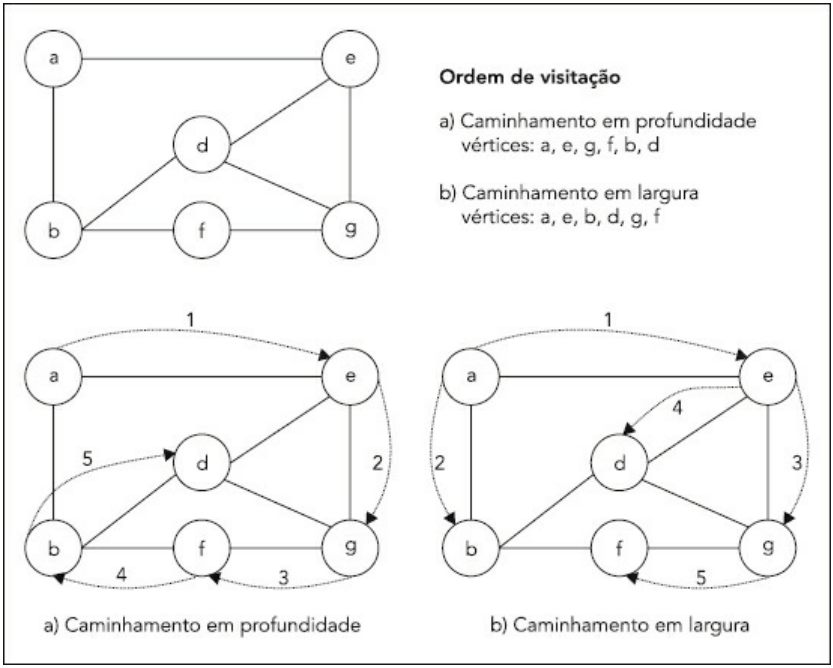


Figura 41 – Representação de grafos ponderados com listas de adjacência.
Fonte: o autor.

De acordo com especificidades do problema sendo modelado ou de restrições para o algoritmo, a escolha de qual das arestas deve ser seguida durante um processo de caminhamento pode levar em conta algum tipo de ordenação prévia dessas arestas, ou seus atributos, no caso de grafos ponderados, por exemplo. Note que as abordagens podem ser utilizadas tanto em grafos direcionados como em grafos não direcionados.

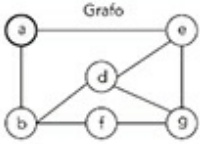
Esses exemplos de caminhamento podem ser gerados com algoritmos bastante similares, com base na anotação de vértices visitados e uma pilha para manter os pontos de continuidade de caminhamento no grafo. Para a implementação do caminhamento em profundidade pode ser utilizada como base o seguinte pseudocódigo, descrito na figura 42. O procedimento considera as seguintes estruturas de dados: uma pilha de valores inteiros para anotação do último vértice visitado, um grafo e um vértice inicial a partir do qual deve ser iniciado o caminhamento. Considera-se que os vértices do grafo estão anotados como nodos não visitados, sendo usada no

procedimento a função "anota" para marcar um determinado vértice como já visitado. A função "armazena_caminho" guarda o rótulo de um vértice escolhido para o caminhamento e a função "busca_vizinho" recebe um índice para um nodo e retorna o índice de um nodo vizinho que não tenha sido ainda visitado, ou então, caso não exista nenhum vizinho com esta condição, retorna um valor nulo (-1). A pilha é utilizada com as funções "empilha", "desempilha" e "retorna_topo". As duas primeiras armazenam e removem um valor da pilha, conforme os seus nomes indicam. A terceira copia o valor do topo da pilha, sem retirar da mesma.

```
void Busca_em_profundidade(Grafo * G, int inicio, Pilha * P){
14.    int va; /* vertice auxiliar */
15.    P->empilha(inicio); //empilha o vértice inicial
16.    G->armazena_caminho(inicio); //guarda primeiro vértice visitado
17.    G->anota(inicio, VISITADO); //marca vértice de início como  visitado
18.    Enquanto ( ! P->vazia() ) {//enquanto a pilha não está vazia
19.          va  = G->busca_vizinho( P->retorna_topo());//copia vértice no topo da pilha
20.          Se ( va == -1)  //se não existe vizinho não visitado
21.                P->desempilha();  //remove da pilha
22.          Senão{
23.                P->empilha(va); //empilha vértices para análises posteriores
24.                G-> armazena_caminho(va); //guarda vértice do caminhamento
25.                G->anota(va, VISITADO);//marca vértice como visitado
26.          }
27.    }
28.    }
```

Figura 42 – Pseudocódigo para caminhamento em profundidade.
Fonte: o autor.

O acompanhamento detalhado da execução desse algoritmo com o grafo exemplo pode servir para facilitar a compreensão de alguns detalhes do funcionamento. A figura 43 ilustra os passos de execução do procedimento descrito na figura 42, sobre um grafo de exemplo.



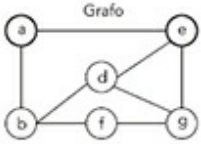
Etapa 1:

Início do algoritmo, com linhas 2 até 4 executadas.

Vértice de início do caminhamento: "a"

Pilha recebeu vértice "a"

Grafo marcado com vértice "a" como visitado



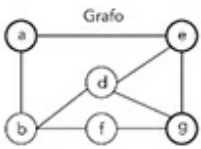
Etapa 2:

Primeiro laço do algoritmo, com linhas 6, 10, 11 e 12 executadas.

Vértice auxiliar: "c"

Pilha recebeu vértice "c"

Grafo marcado com vértice "c" como visitado



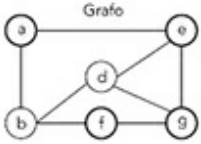
Etapa 3:

Segundo laço do algoritmo, com linhas 6, 10, 11 e 12 executadas.

Vértice auxiliar: "g"

Pilha recebeu vértice "g"

Grafo marcado com vértice "g" como visitado



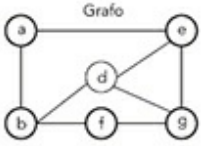
Etapa 4:

Terceiro laço do algoritmo, com linhas 6, 10, 11 e 12 executadas.

Vértice auxiliar: "f"

Pilha recebeu vértice "f"

Grafo marcado com vértice "f" como visitado



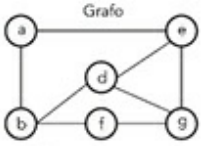
Etapa 5:

Quarto laço do algoritmo, com linhas 6, 10, 11 e 12 executadas.

Vértice auxiliar: "b"

Pilha recebeu vértice "b"

Grafo marcado com vértice "b" como visitado



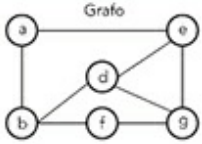
Etapa 6:

Quinto laço do algoritmo, com linhas 6, 10, 11 e 12 executadas.

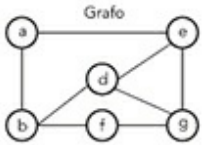
Vértice auxiliar: "d"

Pilha recebeu vértice "d"

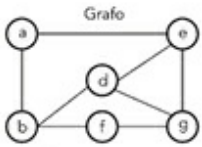
Grafo marcado com vértice "d" como visitado



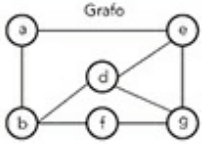
Etapa 7:
Sexto laço do algoritmo, com linhas 6, 7 e 8 executadas.
Vértice auxiliar: valor nulo
Pilha remove vértice "d"



Etapa 8:
Sétimo laço do algoritmo, com linhas 6, 7 e 8 executadas.
Vértice auxiliar: valor nulo
Pilha remove vértice "b"



Etapa 9:
Oitavo laço do algoritmo, com linhas 6, 7 e 8 executadas.
Vértice auxiliar: valor nulo
Pilha remove vértice "f"

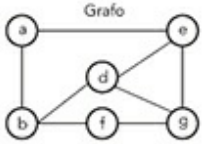


Etapa 10:

Nono laço do algoritmo, com linhas 6, 7 e 8 executadas.

Vértice auxiliar: valor nulo

Pilha remove vértice "g"

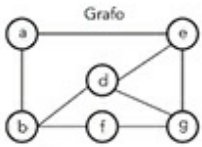


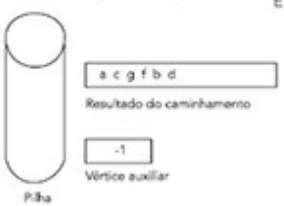
Etapa 11:

Décimo laço do algoritmo, com linhas 6, 7 e 8 executadas.

Vértice auxiliar: valor nulo

Pilha remove vértice c"





Etapa 12:

Décimo-primeiro laço do algoritmo, com linhas 6, 7 e 8 executadas.

Vértice auxiliar: valor nulo

Pilha remove vértice "a"

E com isso finaliza a execução no próximo laço.

Figura 43 – Etapas de execução de caminhamento em profundidade.
Fonte: o autor.

Para a implementação do caminhamento em largura, pode ser utilizada como base o seguinte pseudocódigo, descrito na figura 44. O procedimento considera as seguintes estruturas de dados: uma lista de valores inteiros para anotação dos vértices a serem analisados em cada interação, um grafo e um vértice inicial a partir do qual deve ser iniciado o caminhamento. Considera-se que os vértices do grafo estão anotados como nodos não visitados inicialmente, sendo usada no procedimento a função "anota" para marcar um determinado vértice como já visitado. A função "armazena_caminho" armazena os rótulos dos vértices escolhidos durante o caminhamento e a função "busca_vizinho" recebe um índice para um nodo e retorna o índice de um nodo vizinho que não tenha sido ainda visitado, ou então, caso não exista nenhum vizinho com esta condição, retorna um valor nulo (-1). A fila é utilizada com as funções "armazena_valor", e "recupera_valor". A primeira armazena um valor na fila e a segunda copia o valor no início da fila e o marca como valor lido.

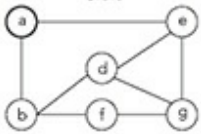
```
void Busca_em_largura(Grafo * G, int inicio, Fila * F){
1.      int va, va2; /* vertices auxiliares */
2.      F->armazena_valor(inicio); //armazena o vértice inicial na fila
3.      G->armazena_caminho(inicio); //guarda primeiro vértice visitado
4.      G->anota(inicio, VISITADO); //marca vértice de início como visitado
5.      Enquanto ( ! F->vazia() ) {//enquanto a fila não está vazia
6.          va = F->recupera_valor();//busca valor no início da fila e o remove da fila
7.          Enquanto ((va2=G->busca_vizinho(va)) != -1) { //busca vizinho não visitado
8.              F->armazena_valor(va2); //guarda vértices para análises posteriores
9.              G-> armazena_caminho(va2); //guarda vértice do caminhamento
10.             G->anota(va2, VISITADO);//marca vértice como visitado
11.         }
12.     }
13. }
```

Figura 44 – Pseudo código para caminhamento em largura.
Fonte: o autor.

O acompanhamento detalhado da execução desse algoritmo com o grafo exemplo

pode servir para facilitar a compreensão de alguns detalhes do funcionamento. A figura 45 ilustra os passos de execução desse procedimento com uso de um grafo de exemplo.

Grafo



a

Fila de valores auxiliares

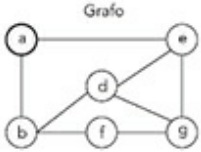
a

Resultado do caminhamento

Vértice auxiliar Vértice auxiliar 2

Etapa 1:
Início do algoritmo, com linhas 2 até 4 executadas.
Vértice de início do caminhamento: "a".
Fila recebeu vértice "a".
Grafo marcado com vértice "a" como visitado.

Grafo



~~a~~

Fila de valores auxiliares

a

Resultado do caminhamento

a Vértice auxiliar 2

Etapa 2:
Início do primeiro laço do algoritmo, com linha 6 executada.
Vértice auxiliar: "a"
Fila vazia, com valor "a" removido

Grafo

~~a~~ b

Fila de valores auxiliares

a b

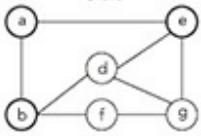
Resultado do caminhamento

a b

Vértice auxiliar Vértice auxiliar 2

Etapa 3:
Execução do segundo laço do algoritmo, com linhas 7, 8 9 e 10 executadas.
Vértice auxiliar não foi alterado: "a"
Segundo vértice auxiliar: "b"
Fila recebeu vértice "b"
Grafo marcado com vértice "b" como visitado

Grafo



b e

Fila de valores auxiliares

a b e

Resultado do caminhamento

a | e

Vértice auxiliar | Vértice auxiliar 2

Etapa 4:

Nova execução do segundo laço do algoritmo, com linhas 7, 8 9 e 10 executadas.

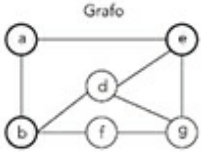
Vértice auxiliar não foi alterado: "a"

Segundo vértice auxiliar: "e"

Fila recebeu vértice "e"

Grafo marcado com vértice "e" como visitado

Grafo



b e

Fila de valores auxiliares

a b e

Resultado do caminhamento

b |

Vértice auxiliar | Vértice auxiliar 2

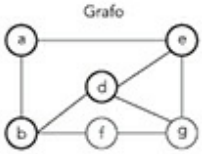
Etapa 5:

Nova execução do primeiro laço do algoritmo, com linha 6 executada.

Vértice auxiliar: "b"

Fila com valor "b" removido

Grafo



e d

Fila de valores auxiliares

a b e d

Resultado do caminhamento

b | d

Vértice auxiliar | Vértice auxiliar 2

Etapa 6:

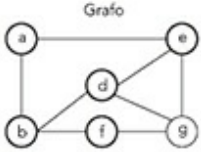
Execução do segundo laço do algoritmo, com linhas 7, 8 9 e 10 executadas.

Vértice auxiliar não foi alterado: "b"

Segundo vértice auxiliar: "d"

Fila recebeu vértice "d"

Grafo marcado com vértice "d" como visitado



~~a~~ ~~b~~ e d f

Fila de valores auxiliares

a b e d f

Resultado do caminhamento

b | f

Vértice auxiliar | Vértice auxiliar 2

Etapa 7:

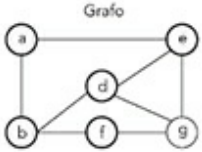
Nova execução do segundo laço do algoritmo, com linhas 7, 8 9 e 10 executadas.

Vértice auxiliar não foi alterado: "b"

Segundo vértice auxiliar: "f"

Fila recebeu vértice "f"

Grafo marcado com vértice "f" como visitado



~~a~~ ~~b~~ ~~e~~ d f

Fila de valores auxiliares

a b e d f

Resultado do caminhamento

e |

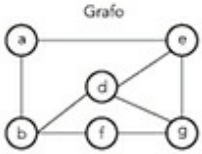
Vértice auxiliar | Vértice auxiliar 2

Etapa 8:

Nova execução do primeiro laço do algoritmo, com linha 6 executada.

Vértice auxiliar: "e"

Fila com valor "e" removido



~~a~~ ~~b~~ ~~e~~ d f g

Fila de valores auxiliares

a b e d f g

Resultado do caminhamento

e | g

Vértice auxiliar | Vértice auxiliar 2

Etapa 9:

Execução do segundo laço do algoritmo, com linhas 7, 8 9 e 10 executadas.

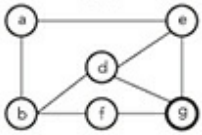
Vértice auxiliar não foi alterado: "e"

Segundo vértice auxiliar: "g"

Fila recebeu vértice "g"

Grafo marcado com vértice "g" como visitado

Grafo



f f f f f g

Fila de valores auxiliares

a b e d f g

Resultado do caminhamento

d

Vértice auxiliar Vértice auxiliar 2

Etapa 10:

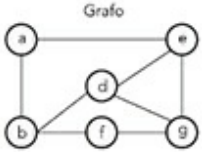
Nova execução do primeiro laço do algoritmo, com linha 6 executada.

Vértice auxiliar: "d"

Fila com valor "d" removido

Laço interno não será executado pois vértice "d" não possui vizinhos ainda não visitados

Grafo



f f f f f g

Fila de valores auxiliares

a b e d f g

Resultado do caminhamento

f

Vértice auxiliar Vértice auxiliar 2

Etapa 11:

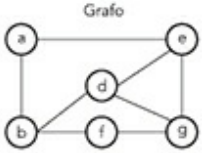
Nova execução do primeiro laço do algoritmo, com linha 6 executada.

Vértice auxiliar: "f"

Fila com valor "f" removido

Laço interno não será executado pois vértice "f" não possui vizinhos ainda não visitados

Grafo



f f f f f f

Fila de valores auxiliares

a b e d f g

Resultado do caminhamento

g

Vértice auxiliar Vértice auxiliar 2

Etapa 12:

Nova execução do primeiro laço do algoritmo, com linha 6 executada.

Vértice auxiliar: "g"

Fila com valor "g" removido

Laço interno não será executado pois vértice "g" não possui vizinhos ainda não visitados

Figura 45 – Etapas de execução de caminhamento em largura.
Fonte: o autor.

Os procedimentos de caminhamento descritos são abordagens de cunho genérico, pois existem diversas possibilidades de implementação de aplicações que necessitam de sua utilização e que possuem detalhamentos particulares. Estes detalhes podem resultar em alterações na forma de implementação, que também pode estar associada com requisitos de performance ou de uso de recursos de memória.

Desta forma, é bastante comum a situação onde os requisitos de um determinado problema definem aspectos do procedimento a ser adotado, para caminhamento em grafos. A seguir, estão descritas duas situações que exemplificam este contexto, uma voltada para grafos não direcionados e outra para grafos direcionados e ponderados.

7.4 Árvore de amplitude mínima

Em determinadas situações, pode ser importante otimizar um projeto complexo, no que diz respeito ao número de conexões entre seus elementos. Sendo este problema descrito por grafos, existem possibilidades de geração de resultados com base nos procedimentos de caminhamento, com algumas restrições e ajustes. Um dos exemplos mais encontrados para ilustrar esta situação é o projeto de circuitos integrados, pois este normalmente envolve uma grande quantidade de conexões ligando os componentes do circuito. A otimização dessas conexões, com a identificação de conexões redundantes e que possam ser simplificadas, é bastante interessante. Outra situação que permite ilustrar o problema é a geração de roteiros com um conjunto de percursos mínimo, o que possibilita apoio em tarefas de planejamento.

É importante destacar que nesta abordagem não estão sendo considerados os percursos em seus atributos, como a contagem do tamanho das conexões no projeto de circuitos ou a distância dos roteiros gerados. Apenas a quantidade de conexões é foco de atenção neste caso, em uma primeira abordagem. Deste modo, considera-se a aplicação desta técnica em um grafo não ponderado, pois este aspecto não será tratado. Além disso, considera-se que o procedimento será aplicado em grafos conectados e simples, ou seja, um vértice não possui conexão com ele mesmo. Na seção seguinte serão tratados também os problemas relacionados com grafos ponderados, que possibilitam atender às necessidades de contextos diferentes.

Portanto, considera-se uma árvore de amplitude mínima, ou, em inglês, *Minimum Spannin Tree* (MST), como sendo o grafo com o número mínimo de arestas necessárias para conectar todos os seus vértices. A figura 46, a seguir, ilustra com um exemplo simples o resultado obtido com esse procedimento.

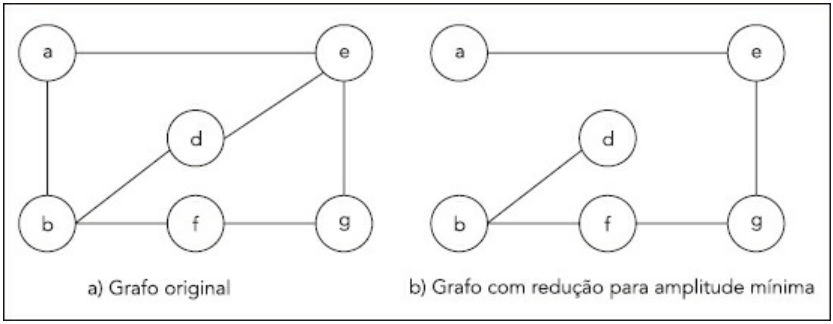


Figura 46 – Árvore de amplitude mínima.
Fonte: o autor.

Observa-se na figura 46 que o resultado proposto para este caso é composto pelas seguintes arestas: {(a, e), (e, g), (g, f), (f, b), (b, d)}. Este conjunto, apesar de correto, não é o único possível para a geração de uma árvore de amplitude mínima para este grafo. Por exemplo, o conjunto a seguir também proporciona esta propriedade: {(a, b), (b, d), (d, e), (e, g), (g, f)}. Portanto, existem, em geral, mais de uma opção de resultado correto.

Além disso, pode ser observado que nos dois exemplos destacados, o número de arestas é igual a cinco, sendo que o número de vértices é igual a seis. Esta é outra observação geral importante, sendo que o número de arestas para a árvore de amplitude mínima de um grafo deve ser igual ao número de vértices menos um.

A implementação de procedimentos para a obtenção da árvore de amplitude mínima pode ser feita de diversas formas, conforme os detalhes de implementações, performance desejada e detalhes específicos da situação em que se aplica o procedimento, conforme já foi comentado. Apesar disso, existe uma semelhança muito grande entre a geração de árvore mínima e os caminhamentos vistos anteriormente. No caso do caminhamento em profundidade apresentado, basta a organização dos resultados de forma a indicação de sequência de arestas e serão obtidos os elementos que formam a árvore de amplitude mínima desejada.

Para o caso de grafos ponderados, uma abordagem de árvore de amplitude mínima pode ser considerada como o grafo que realiza a conexão entre todos os vértices, com o mínimo de arestas e considerando o menor peso possível entre estas aresta, para cada grupo de vértices. Este problema possui alguns algoritmos dedicados à sua solução, tais como o algoritmo de Kruskal ou o algoritmo de Prim-Jarnik. O algoritmo de Kruskal possibilita a construção de árvores de amplitude mínima em grafos conectados ponderados com tempo de execução na ordem de $O(n^3)$, sendo n o número

de vértices do grafo. O algoritmo trabalha com a construção deste resultado gerando grupos de árvores e depois conectando-as, levando em conta os pesos mínimos das arestas. Já o algoritmo de Prim-Jarnik utiliza um vértice raiz e segue seu procedimento analisando a agregando interativamente a aresta de menor peso. Seu tempo de execução é similar ao do algoritmo de Kruskal.

7.5 Ordenação topológica

As técnicas utilizadas na classe de problemas, conhecida como ordenação topológica, partem de descrições de grafos que são orientados e ponderados, portanto, utilizados, em geral, para descrever elementos com requisitos entre si. Tais grafos poderiam, por exemplo, ser empregados para descrever uma sequência de disciplinas de um curso de graduação com seus requisitos, ou, possibilitariam a descrição de um projeto complexo em etapas menores, delimitadas pelas dependências existentes entre si. Tais grafos são bastante úteis para a descrição e simulação, em especial de situações associadas com projetos, pois possibilitam a identificação de situações conhecidas como caminhos críticos, que relacionam as demandas do projeto e seus requisitos.

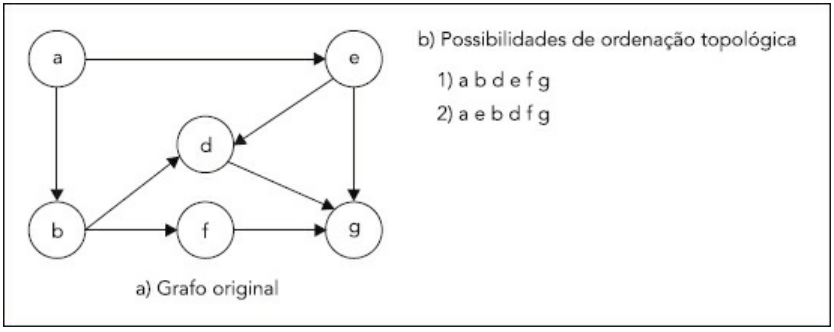


Figura 47 – Exemplos de ordenação topológica.
Fonte: o autor.

No caso de um grafo orientado, os nodos devem ser ordenados com algum critério, dado pela descrição do problema sendo encaminhado. Supondo que no exemplo do grafo da figura 47 as arestas indicam precedência e cada vértice indica uma atividade que deve ser realizada completamente antes da atividade representada pelo vértice à qual está conectada, isso nos indicaria que a atividade representada pelo vértice “a” deve ser vista como requisito para as atividades indicadas nos vértices “b” e “e”. De

forma similar, o vértice “g” possui como requisitos as atividades “f”, “d”, e “e”. Por sua vez, as atividades “d” e “f” estão relacionadas com a atividade “b”, que deve ser considerada como requisito para essas duas.

Ainda na figura 47, também estão relacionadas duas sugestões de solução para a ordenação topológica neste caso, considerando a direção das arestas como indicações de requisitos. Note que mais de uma alternativa existe, de modo a atender aos requisitos indicados.

A resolução de situações de ordenação topológica pode ser realizada com recursos já observados, utilizados para o caminhamento em grafos. Por exemplo, o procedimento de caminhamento em largura, onde todos os vértices vizinhos de um vértice de origem são visitados, em etapas que se repetem, permite solucionar diversos contextos de ordenação topológica. Outra possibilidade de resolução segue com base na análise dos vértices do grafo, localizando os vértices que não possuem sucessores, retirando-os da descrição do grafo e anotando os vértices retirados do grafo em uma fila. Após a varredura sucessiva de todos os vértices com este critério, a fila gerada possuirá a sequência dos vértices ordenada de forma invertida, de modo que todos os vértices serão sempre associados com a existência de requisitos.

7.6 Caminhos mais curtos

No caso de um grafo ponderado, as arestas podem ser utilizadas para a descrição de elementos do problema sendo modelado. Por exemplo, podem ser utilizadas para descrever a distância entre cidades, a capacidade de um fluxo de condutores ou o tempo de execução de tarefas. Ou então, no caso de apoio para resolução de problemas na área de inteligência artificial ou no apoio a jogos de entretenimento digital, os grafos podem representar contextos e restrições entre elementos destes contextos.

Veja um exemplo na figura 48 a seguir. Nesta figura observa-se um grafo contendo arestas com pesos e direcionadas. No caso de um problema associado à localização geográfica, os valores nos pesos das arestas poderiam estar associados com as distâncias entre cidades identificadas nos vértices do grafo. Assim, para o deslocamento entre a cidade representada pelo vértice “A”, até a cidade representada pelo vértice “B” existe um trajeto, com distância igual a 14. Um trajeto alternativo entre o vértice “A” e o vértice “B” pode envolver um caminho passando pelo vértice “C”, situação em que a distância total seria igual a 16. Já no caso de vértices representando tarefas, o valor das arestas poderia significar o tempo de execução da mesma, sendo que a direção permite associar prioridades entre as tarefas. Por fim, caso este grafo represente um conjunto de conexões, o valor das arestas pode estar associado com a capacidade de transferência entre as conexões.

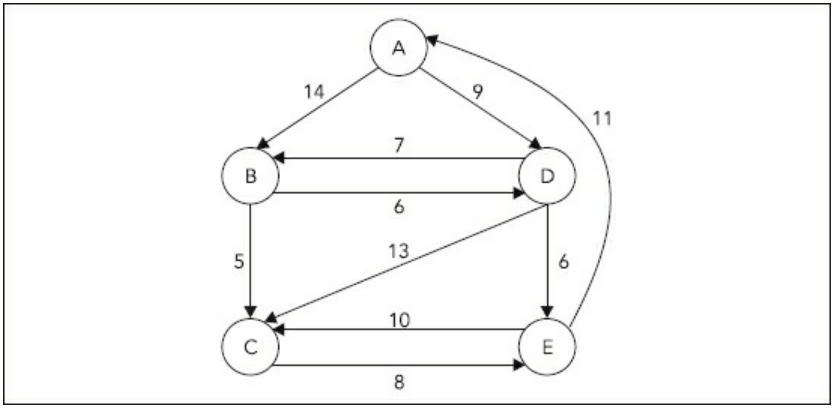


Figura 48 – Modelagem de problemas com grafos ponderados.
Fonte: o autor.

A questão de identificação de caminhos mais curtos em grafos, a partir dos exemplos mencionados, pode estar associada com uma diversidade de situações. Especificamente, considera-se que estas situações serão modeladas em grafos dirigidos, não cíclicos e ponderados, de modo que a cada descrição de grafo o significado dos atributos esteja relacionado com o problema sendo modelado.

A descrição de caminhos mais curtos pode ser considerada a partir de um determinado ponto de origem do grafo, o que poderia ser descrito como a análise de um grafo ponderado G = (V, A), sendo que o ponto de origem do caminho mais curto é o vértice *v* pertencente ao conjunto de vértices V. Assim, o caminho mais curto entre vértice *v* e um determinado vértice v_i, pertencente ao conjunto de vértices V, seria o caminho formado por n vértices vizinhos, com início em *v*, até o vértice v_i, de modo que a soma dos valores de suas arestas seja o menor possível.

A resolução desse problema pode ser adaptada, com pequenas variações, à situações ligeiramente diferenciadas. Além da identificação de caminhos mais curtos, considerando-se um ponto de origem único, pode-se imaginar uma situação em que o destino único é fornecido, sendo necessária a identificação de forma inversa dos caminhos mais curtos, ou seja, cada vértice deve ser usado para calcular o seu caminho mais curto até o ponto de destino. Ou então pode ser necessária a identificação de um caminho mais curto entre um par de vértices, em uma situação com maior dinamicidade, onde necessidades diversas precisam ser acomodadas a partir de informações mantidas no grafo. Esta situação pode também ser resolvida com uma adaptação do cálculo com base em origem única. Ou, por fim, pode haver necessidade

na identificação de caminhos mais curtos entre todos os pares de vértices, o que implicaria na execução do mesmo algoritmo de origem comum para n ocorrências de vértices do grafo.

Em todas essas abordagens, é possível a identificação de mais de uma solução para o caminho mais curto, que pode não ser único, sendo esta situação tratada em geral nas implementações dos algoritmos.

Um formato interessante para a implementação de soluções para este problema é a geração de uma árvore de caminhos mais curtos, com base em um determinado grafo e um vértice de origem. A partir dessa árvore são descritos os caminhos mais curtos entre o vértice de origem utilizado no cálculo e todos os demais vértices existentes no grafo original. Um dos algoritmos que podem ser utilizados nesta operação é o algoritmo de Dijkstra, descrito e exemplificado a seguir.

7.7 Algoritmo de Dijkstra

O algoritmo de Dijkstra permite a geração de uma árvore com caminhos mais curtos a partir de um grafo G, dado um vértice de origem. O resultado será, portanto, o conjunto de caminhos mais curtos para cada vértice do grafo, desde que seja alcançável a partir do vértice de origem.

A estratégia utilizada no algoritmo de Dijkstra pode ser resumida em duas componentes. Primeiro, o algoritmo envolve a manutenção de um conjunto de vértices do grafo em análise, sendo estes já definidos em seu caminho mais curto desde o vértice de origem. Para esses vértices é mantida a informação de seus predecessores, o que permite a montagem dos caminhos com base nas análises já realizadas. Aliado a este componente, o algoritmo utiliza uma técnica de reavaliação interativa dos valores para os caminhos envolvendo os vértices ainda não presentes no conjunto dos vértices já com o caminho mais curto definido. Esta técnica também é descrita como técnica de relaxamento dos vértices, em algumas descrições deste algoritmo. Resumidamente, ela consiste em um processo de atualização do cálculo do caminho mais curto de forma interativa, envolvendo os vértices ainda não definidos no conjunto fechado de vértices analisados. Isso garante que novos caminhos sejam sempre baseados nos caminhos mais curtos já definidos, o que permite a obtenção dos resultados corretos ao final do processamento.

A figura 40 descreve o pseudocódigo necessário para a implementação do procedimento. São consideradas as estruturas de dados auxiliares comentadas a seguir. Um conjunto de vértices identificado neste exemplo como vértices fechados (vertices_fechados), que consiste dos vértices já analisados e identificados com os valores de menor caminho adequados. Inicialmente, este conjunto é vazio e o primeiro vértice a ser inserido nele é o vértice de origem para a análise de menor caminho. Uma lista de valores de distância, que identifica a distância do vértice a seu predecessor.

Uma lista contendo valores que identificam o predecessor de cada vértice, de acordo com o menor caminho analisado. A cada passo do algoritmo a lista de valores de distância é atualizada, para adequar seus valores de acordo com os menores caminhos delimitados pela análise. Para uma implementação eficiente, uma escolha adequada para manutenção deste componente do algoritmo é o uso de um heap, mantendo com eficiência a ordenação e permitindo a flexibilidade necessária para os ajustes de valores. No exemplo da figura 49 esta questão é abstraída.

A execução do algoritmo passa por uma etapa de inicialização na qual os valores para distância entre os vértices são configurados com uma indicação de valor infinito e os valores de predecessor são configurados como nulos. Ainda na inicialização, a distância para o vértice de origem é inicializada com o valor zero e o conjunto de vértices fechados é inicializado como um conjunto vazio. Estes passos são descritos no pseudocódigo da figura 49, entre as linhas 2 e 7. Na linha 7 está considerada uma etapa de inicialização que depende da implementação específica a ser feita e poderia ser, por exemplo, a implementação de um heap. Durante o laço de execução do algoritmo, na linha 9, está previsto outro procedimento que ordena os valores e busca o valor mínimo, também dependente de implementação específica.

Durante o laço principal do algoritmo, repete-se uma sequência simples de procedimentos, que são a identificação do vértice de menor valor na lista de valores de distância (linha 9) e sua inserção no conjunto de valores fechados (linha 10). Para cada novo vértice inserido nesse conjunto, são avaliados os seus vizinhos e recalculado o valor da distância e o predecessor. Este procedimento permite a correta identificação dos menores caminhos a partir da análise sistemática realizada no grafo em análise.

```
1.  Dijkstra (Grafo *g, int origem)
2.          Para v entre 0 e g->MaxVertices()
3.                  distancia[v] = infinito
4.                  predecessor[v]=nulo
5.          distancia[origem] = 0
6.          vertices_fechados = vazio
7.          ordena_distancias()
8.          Enquanto ( analisar_distancias())
9.                  u = distancia_minima()
10.                 vertices_fechados = vertices_fechados + u
11.                 Enquanto existir v = vizinho(u)
12.                         Se distancia[v] > distancia[u] + peso(u,v)
13.                                 distancia[v] = distancia[u] + peso(u,v)
14.                                 predecessor [v] = u
```

Figura 49 – Pseudocódigo para algoritmo de Dijkstra.
Fonte: o autor.

Na figura 50 está detalhada a execução dos passos deste algoritmo para um grafo exemplo, contendo cinco vértices, arestas direcionadas e ponderadas. A cada etapa são atualizadas as informações das estruturas de dados auxiliares utilizadas para a manutenção do procedimento.

Na etapa 1, por exemplo, pode-se verificar o resultado da inicialização, com o conjunto de vértices fechados vazio, o valor de distância indicado apenas para o vértice de origem (vértice "A", com valor zero), enquanto os demais vértices são associados com uma distância infinita. Os valores de vértices predecessores também estão inicializados com valores nulos.

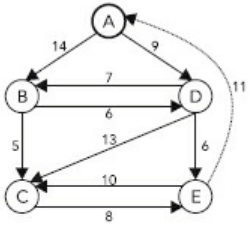
Observa-se na etapa 2 a execução do primeiro laço do algoritmo, compreendendo as linhas 8 a 14 do pseudocódigo descrito na figura 49. O vértice indicado como origem ("A") é o escolhido para ser inserido no conjunto de vértices fechados, pois apresenta a menor distância, já que possui valor zero enquanto todos os demais, inicialmente, possuem indicação de valor infinito. Com a sua escolha, são atualizadas as distâncias e os predecessores dos seus vértices vizinhos. No caso, os vértices são "B" e "D", que recebem valor igual a 14 e 9 para distância, respectivamente. O valor de antecessor para ambos é o do vértice "A".

Deve ser observado que a atualização da distância ocorre a partir do teste feito na

linha 12 do pseudocódigo. Por exemplo, no momento da simulação da execução com base no exemplo dado pela figura 50, um dos testes seria feito com o valor do vértice "B", comparando a distância armazenada com a distância do vértice "A" somada com o valor da aresta entre os dois ("AB"). Com a configuração da listagem de valores obtida, na etapa seguinte ocorrerá a inclusão do vértice "D" no conjunto de vértices fechados, com base na comparação dos valores disponíveis. A sua inserção no conjunto acarretará também a atualização dos valores dos seus vértices vizinhos: "B", "C" e "E".

Na etapa 4 será escolhido para inserção no conjunto de vértices fechados o vértice "B", sendo que deve ser destacado que apenas o seu vértice vizinho "C" será analisado quanto aos valores de distância e predecessor, pois o seu vértice vizinho "D" já se encontra no conjunto de vértices fechados.

Com base nesta dinâmica, o procedimento continuará até a etapa final, na qual o resultado é uma organização com os menores caminhos a partir do vértice de origem "A".



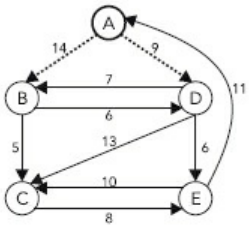
Etapa 1:

Vértice de origem : A

Inicialização de valores

		Distância				
Etapas	Vértices Fechados	A	B	C	D	E
1	{ }	0	8	8	8	8

Predecessor				
A	B	C	D	E
nulo	nulo	nulo	nulo	nulo



Etapa 2:

Busca vértice com distância mínima

Vértice A inserido no conjunto de vértices fechados.

Análise dos vizinhos de A: vértices B e D

Atualização das distâncias de B e D

Atualização do predecessor de B e D

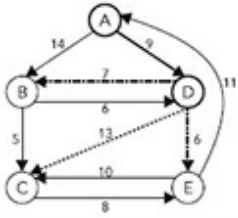
Testes na linhas 12:

se (distancia[B] > distancia[A] + peso(A, B)) -> (∞ > 0 + 14)

se (distancia[C] > distancia[A] + peso(A, C)) -> (∞ >0 + 9)

		Distância				
Etapas	Vértices Fechados	A	B	C	D	E
1	{ }	0	8	8	8	8
2	{A }	0	14	9	8	8

Predecessor				
A	B	C	D	E
nulo	nulo	nulo	nulo	nulo
nulo	A	A	nulo	nulo



Etapa 3:

Busca vértice com distância mínima

Vértice D inserido no conjunto de vértices fechados.

Análise dos vizinhos de D: vértices B, C e E

Atualização das distâncias de B, C e E

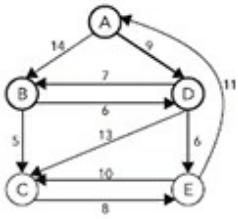
Atualização do predecessor de B, C e E

Distância

Etapas	Vértices Fechados	A	B	C	D	E
1	{ }	0	8	8	8	8
2	{A }	0	14	8	9	8
3	{A D }	0	14	22	9	15

Predecessor

A	B	C	D	E
nulo	nulo	nulo	nulo	nulo
nulo	A	nulo	A	nulo
nulo	A	D	A	D



Etapa 4:

Busca vértice com distância mínima

Vértice B inserido no conjunto de vértices fechados.

Análise dos vizinhos de B: vértices C

Atualização da distância de C

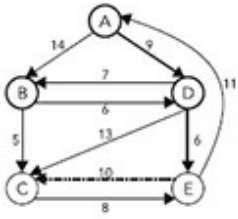
Atualização do predecessor de C

Distância

Etapas	Vértices Fechados	A	B	C	D	E
1	{ }	0	8	8	8	8
2	{A }	0	14	8	9	8
3	{A D }	0	14	22	9	15
4	{A D B }	0	14	19	9	15

Predecessor

A	B	C	D	E
nulo	nulo	nulo	nulo	nulo
nulo	A	nulo	A	nulo
nulo	A	D	A	D
nulo	A	B	A	D



Etapa 5:

Busca vértice com distância mínima

Vértice B inserido no conjunto de vértices fechados.

Análise dos vizinhos de B: vértices C

Atualização da distância de C

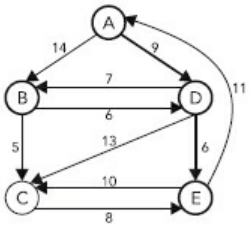
Atualização do predecessor de C

Distância

Etapas	Vértices Fechados	A	B	C	D	E
1	{ }	0	8	8	8	8
2	{A }	0	14	8	9	8
3	{A D }	0	14	22	9	15
4	{A D B }	0	14	19	9	15
5	{A D B E }	0	14	19	9	15

Predecessor

A	B	C	D	E
nulo	nulo	nulo	nulo	nulo
nulo	A	nulo	A	nulo
nulo	A	D	A	D
nulo	A	B	A	D
nulo	A	B	A	D



Etapa 6:

Busca vértice com distância mínima

Vértice C inserido no conjunto de vértices fechados.

Análise dos vizinhos de B: nenhum a analisar

		Distância					Predecessor				
Etapas	Vértices Fechados	A	B	C	D	E	A	B	C	D	E
1	{ }	0	8	8	8	8	nulo	nulo	nulo	nulo	nulo
2	{A }	0	14	8	9	8	nulo	A	nulo	A	nulo
3	{A D }	0	14	22	9	15	nulo	A	D	A	D
4	{A D B }	0	14	19	9	15	nulo	A	B	A	D
5	{A D B E }	0	14	19	9	15	nulo	A	B	A	D
6	{A D B E C }	0	14	19	9	15	nulo	A	B	A	D

Figura 50 – Exemplo da execução do algoritmo de Dijkstra.
Fonte: o autor.

O algoritmo de Dijkstra apresenta uma boa performance, com um tempo de execução na ordem de $O(n^3)$, para n vértices em um grafo. Detalhes da implementação, como o uso ou não de estruturas mais eficientes para os procedimentos de ordenação de distância necessários podem significar melhoras.

REFERÊNCIAS

AHO, A. V.; ULLMAN, J. D. *Data structures and algorithms*. Prentice-Hall, 1985.

AHO, A. V.; ULLMAN, J. *Foundations of computer science: principles of computer sciences series*. New York: W. H. Freeman e Co., 1995.

BRASSARD, G., BRATLEY, P. *Fundamentals of algorithmics*. New York: Prentice Hall, 1996.

CORMEN, T. H. *Algoritmos, teoria e prática*. São Paulo: Editora Campus, 2002.

DEITEL, H.M. DEITEL, P. J. *Java – como programar*. 6. ed. São Paulo: Pearson, 2007.

GOODRICH, M. T.; TAMASSIA, R. *Estruturas de dados e algoritmos em Java*. 4. ed. Porto Alegre: Bookman, 2007.

LAFORE, R. *Estruturas de dados e algoritmos em Java*. 2. ed. São José: Ciência Moderna, 2005.

SHAFFER, C. *A practical introduction do data structures and algorithm analysis*. New York: Prentice Hall, 2001.

VELOSO, P. et al. *Estruturas de dados*. São Paulo: Editora Campus, 1983.

UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS

Reitor
Pe. Marcelo Fernandes de Aquino, SJ

Vice-reitor
Pe. José Ivo Follmann, SJ

EDITORA UNISINOS

Diretor
Pe. Pedro Gilberto Gomes, SJ

Editora da Universidade do Vale do Rio dos Sinos
EDITORA UNISINOS
Av. Unisinos, 950
93022-000 São Leopoldo RS Brasil

Telef.: 51.3590 8239
Fax: 51.3590 8238
editora@unisinos.br

R572e Rigo, Sandro José.
Estruturas avançadas de dados: uma introdução / Sandro José Rigo. – São Leopoldo: Unisinos, 2011.

104 p. – (EaD)

ISBN 978-85-7431-438-9

1. Estruturas de dados (Computação). 2. Algoritmos. 3. Ensino a distância. I. Titulo. II. Série

CDD 005.73
CDU 004.6

Dados Internacionais de Catalogação na Publicação (CIP)
(Bibliotecário: Flávio Nunes – CRB 10/1298)

Esta obra segue as normas do Acordo Ortográfico da Língua Portuguesa vigente desde 2009.

COLEÇÃO EAD

Editor
Carlos Alberto Gianotti

Acompanhamento editorial
Mateus Colombo Mendes

Revisão	*Editoração*	*Capa*
Caroline Soares	Rafael Tarcísio Forneck	Isabel Carballo

Impressão, verão de 2011.

Sobre o autor

SANDRO JOSÉ RIGO é doutor em Ciência da Computação pela Universidade Federal do Rio Grande do Sul (UFRGS). mestre em Ciência da Computação pela UFRGS, bacharel em *Software* Básico pela Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS). Professor da Unisinos nos cursos de Ciência da Computação, Comunicação Digital e Desenvolvimento de Jogos Digitais, também leciona em cursos de especialização em Gestão Escolar e Administração de Tecnologia de Informação. É consultor em EaD na Unisinos Virtual, onde atua na produção de Objetos de Aprendizagem e desenvolvimento de cursos na modalidade Educação a Distância.

Edição digital: dezembro 2013

Arquivo ePub produzido pela **Simplíssimo Livros**

A coleção EaD, de que faz parte este livro, é uma produção da Universidade do Vale do Rio dos Sinos para apoiar os processos de ensino e aprendizagem dos seus cursos de graduação a distância. Entretanto, o uso dessa obra não fica restrito apenas a essa modalidade de ensino, uma vez que pode servir como orientador no estudo de qualquer acadêmico.
Os exemplares foram elaborados a partir da experiência de professores de reconhecido mérito acadêmico da Universidade, e traduzem a excelência dos cursos de graduação ofertados na modalidade presencial e a distância da Instituição.

MEMBRO DA
REDE DE EDITORAS UNIVERSITÁRIAS DA AUSJAL
www.ausjal.org

COLEÇÃO EAD
EDITORA UNISINOS

UNISINOS