2º Trabalho de Projeto e Analise de Algoritmos (PAA)
Algoritmos de Arvores
Busca em Profundidade(DFS)
Busca em Largura (BFS)

Gabriel Santos da Silva[1], **David Antonio Brocardo**[1]

[1]Centro de Ciências Exatas e Tecnológicas
Campus de Cascavel - UNIOESTE
Caixa Postal 801 – 85.814-110 – Cascavel – PR – Brazil

{gabirel.silva77,david.brocardo}@unioeste.br

Abstract. *This work aims to apply and validate the concepts learned about hierarchical data manipulation algorithms. To this end, a Red-Black (RB) tree was constructed, which served as the base structure for the searches to be evaluated. The observed performances are, therefore, directly related to the behavior of the obtained RB tree. The first solution consists of implementing Depth-First Search (DFS). Then, the implementation of the Breadth-First Search (BFS) strategy. The objective is to compare the two approaches in terms of execution time and number of comparisons between keys during search operations. The results compiled in this work conclude that, in general, depth-first search presents a better performance than breadth-first search, especially in relation to execution time, although both have the same asymptotic complexity.*

Resumo. *Este trabalho tem como objetivo aplicar e validar os conceitos aprendidos sobre algoritmos de manipulação de dados hierárquicos. Para isso, foi construída uma árvore Rubro-Negra (RB), que serviu como estrutura base para as buscas a serem avaliadas. Os desempenhos observados estão, portanto, diretamente relacionados ao comportamento da árvore RB obtida. A primeira solução consiste na implementação da Busca em Profundidade (DFS). Em seguida, a implementação da estratégia de Busca em Largura (BFS). O objetivo é comparar as duas abordagens em termos de tempo de execução e número de comparações entre chaves durante as operações de busca. Os resultados compilados nesse trabalho concluem que em geral, a busca em profundidade apresenta um desempenho superior à busca em largura, especialmente em relação ao tempo de execução, embora ambas possuam a mesma complexidade assintótica.*

1. Introdução

Segundo [for Geeks 2024c], Árvores de busca binárias são uma estrutura de dados fundamental, mas seu desempenho pode sofrer se a árvore ficar desbalanceada. Árvores Red-Black são um tipo de árvore de busca binária balanceada que usa um conjunto de regras para manter o equilíbrio, garantindo complexidade de tempo logarítmica para operações como inserção, exclusão e busca, independentemente do formato inicial da árvore.

Árvores Red-black são auto balanceadas, usando um esquema simples de codificação de cores para ajustar a árvore após cada modificação. [...] cada nó tem um atributo adicional: uma cor, que pode ser vermelha ou preta. O objetivo principal dessas árvores é manter o equilíbrio durante inserções e exclusões, garantindo recuperação e manipulação eficientes de dados.

Com base em [for Geeks 2024b], Busca em Largura (BFS) e Busca em Profundidade (DFS) são dois algoritmos fundamentais usados para percorrer ou pesquisar gráficos e árvores. [...] BFS é uma abordagem de travessia na qual primeiro percorremos todos os nós no mesmo nível antes de passar para o próximo nível. O DFS também é uma abordagem de travessia na qual a travessia começa no nó raiz e prossegue pelos nós o máximo possível até chegarmos ao nó sem nós próximos não visitados.

Neste trabalho, foi feita a implementação da Busca em Profundidade e a implementação da estratégia de Busca em Largura ambas aplicadas a uma árvore Red-Black, os dados utilizados para construir a árvore e fazer as consultas durante as buscas estão dispostos em arquivos de que vão desde os 100 números até o 250 mil números. Como resultados, encontramos que em geral a busca em profundidade apresenta um desempenho superior à busca em largura, especialmente em relação ao tempo de execução, embora ambas possuam a mesma complexidade assintótica.

2. Abordagens implementadas

Para atingir o objetivo deste trabalho, foi primeiramente construída a árvore RB com auxílio do modelo padrão presente no repositório da geek for geeks; em seguida as buscas BFS e DFS foram adicionadas ao algoritmo. As estruturas implementadas seguindo a ordem: Construção da árvore, busca BFS, busca DFS. A saída para cada uma das entradas são: ‘arquivoComparacoesBFS’ e ‘arquivoComparacoesDFS’ (presente o número de comparações necessarias); ‘arquivoTempo’ que armazena o tempo total de execução para BFS e DFS. Totas as implementações foram feitas em C++ utilizando a IDE Visual Studio Code e armazenadas em um repositório na plataforma GitHub. Foi escolhida a linguagem de programação C++, pois ela é uma excelente escolha se tratando de desempenho para aplicações que exigem a máxima velocidade e eficiência, além de dar um maior controle sobre o hardware e a memória e a flexibilidade em criar um software complexo e adaptável.

Para a coleta dos tempos de execução, foi utilizada a biblioteca “time”, que registra o tempo inicial no momento da execução do algoritmo e o tempo final quando ele é concluído, permitindo assim calcular o tempo total de execução e armazenar esse tempo devidamente identificado. Para esse trabalho, o tempo registrado pela biblioteca “time” retorna em segundos.

2.1. A Árvore RB

A Árvore Rubro-Negra (RB) foi criada através da inserção sequencial de valores numéricos lidos de arquivos de entrada, onde cada valor se torna um nó da árvore. Durante a inserção, primeiramente a cada novo nó ele é colorido de vermelho, e a árvore é rebalanceada para manter suas propriedades através de rotações e recolorações, garantindo assim que a árvore permaneça balanceada para as buscas.

2.2. Busca em Profundidade

A implementação da busca em profundidade percorre a árvore em sentido Pré-Ordem, ela é realizada por uma função presente no código que além de buscar o elemento desejado também conta o número de comparações. A busca em profundidade implementada verifica o lado esquerdo todo (até encontrar ou não) e depois verifica o lado direito todo (até encontrar ou não). Os possíveis retornos da função são: Se a raiz for null o retorno é null; Se a raiz for o número procurado o retorno é o número procurado; Se o número procurado estiver do lado esquerdo o retorno é o número procurado e o número de comparações; Se o número procurado estiver do lado direito então o retorno é o número procurado e o número de comparações.

Vale ressaltar que o custo assintótico da busca em profundidade depende do número de vértices tendo um custo de tempo em O(V+E), pois visita todos os vértices e percorre todas as arestas uma única vez. E custo assintótico de espaço em O(V) devido à profundidade da recursão.

2.3. Busca em Largura

A implementação da busca em largura é realizada por uma função presente no código que além de buscar o elemento desejado também conta o número de comparações. A busca em largura implementada usa uma estrutura de fila para encontrar o número desejado. Os possíveis retornos da função são: Enquanto a fila não está vazia procura o número desejado retornando o Nó Atual que está na frente da fila.

Vale ressaltar que o custo assintótico da busca em largura depende do número de vértices tendo um custo de tempo em O(V+E), pois também percorre todos os vértices e arestas uma única vez. E custo assintótico de espaço em O(V) pois armazena todos os nós do nível atual na fila. Ambas as buscas possuem o mesmo custo assintótico de tempo, mas o BFS pode consumir mais memória em grafos muito largos devido ao uso da fila.

3. Avaliação Prática

No presente algoritmo, foi implementado um trecho de código com o objetivo de avaliar, na prática, a busca em profundidade (DFS) e a busca em largura (BFS). Esse trecho foi responsável por abrir e ler os valores de entrada para a árvore rubro-negra e, após a inserção desses valores na estrutura, realizar seis execuções consecutivas, descartando os resultados da primeira, conforme especificado no trabalho. O tempo médio das cinco execuções restantes foi armazenado para análise futura em um arquivo CSV, utilizando a biblioteca *time* da linguagem C++. O professor da disciplina disponibilizou dois conjuntos de dados distintos:

- Conjunto ”Construir”: utilizado para a inserção de valores na árvore rubro-negra;
- Conjunto ”Consultar”: empregado pelos algoritmos DFS e BFS para a busca de elementos na árvore.

Cada conjunto possui 28 arquivos, com tamanhos variando de 50 até 250.000 números. Durante a execução das buscas em profundidade e largura, foram coletados os seguintes dados:

- Tempo médio de execução para cada arquivo do conjunto “Consultar” nos algoritmos DFS e BFS;

- Quantidade de comparações realizadas para encontrar cada número nos arquivos de consulta, em ambos os algoritmos.

Para a realização desses testes, os dados foram organizados em pastas de acordo com sua categoria, e o código *arvoreRB.cpp* foi compilado e executado via *prompt* de comando. Todos os experimentos foram realizados na mesma máquina, com as seguintes configurações: processador 12th Gen Intel(R) Core(TM) i5-12450H 2.00 GHz, 8,00 GB de memória RAM (7,71 GB utilizáveis) e sistema operacional Windows 11 de 64 bits.

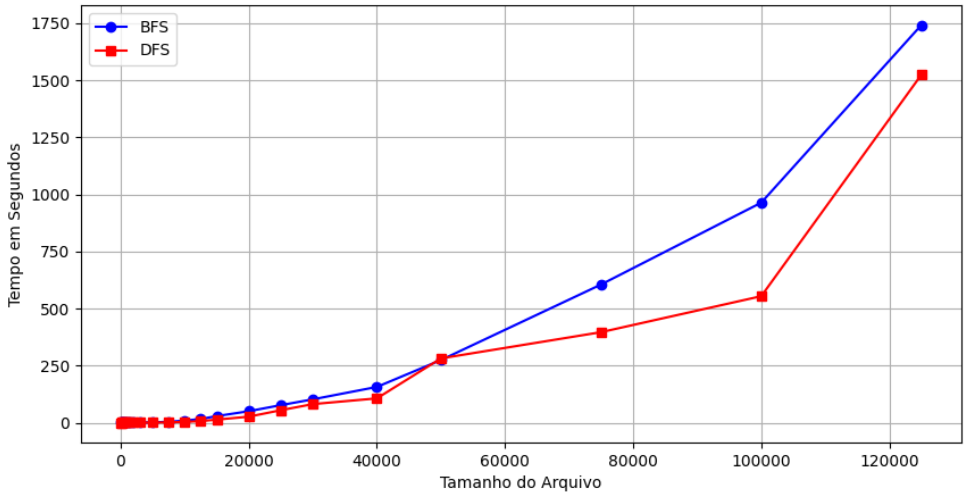
Importante destacar que foi realizado a DFS e BFS, somente até 125 mil valores, essa decisão de interromper a execução foi feita pelos autores devido a demora para finalizar a busca em arquivos maiores. Esses códigos permaneceram em execução por mais de 10 horas.

4. Resultados

A partir do experimento prático realizado, foram obtidos dados sobre o tempo de execução das duas abordagens de busca, bem como a quantidade de comparações realizadas por cada algoritmo na busca de elementos dentro da árvore rubro-negra. Para uma análise mais detalhada do desempenho dos algoritmos, primeiramente será avaliada a eficiência cronológica de cada abordagem e, em seguida, será analisada a quantidade de comparações realizadas para diferentes volumes de dados.

Ao examinar os tempos de execução dos algoritmos BFS e DFS, conforme apresentado na Figura 1, observa-se uma leve vantagem da busca em profundidade sobre a busca em largura. Essa diferença torna-se mais evidente nos arquivos com maior número de entradas, nos quais a busca em largura demanda um tempo significativamente maior para concluir a busca de todos os elementos. Esse comportamento é demonstrado por [for Geeks 2024a], que destaca que a busca em largura é mais eficiente para encontrar vértices próximos à fonte, enquanto a busca em profundidade se mostra mais adequada para cenários em que a solução está mais distante da origem.

Figura 1. Tempo de execução em segundo para buscar valores dentro de uma árvore Rubro negra, através de uma busca de profundidade(DFS) e largura(BFS), com quantidade de elementos a serem buscado variando de 100 a 125000 valores.



Fonte: Autoria própria

Ao analisar a Figura 1, observa-se que, para os elementos iniciais até o arquivo 20000.txt, o tempo de execução aparenta ser semelhante para ambos os algoritmos. No entanto, ao examinar a Tabela 1, percebe-se que, embora os tempos sejam próximos, a busca em profundidade ainda consegue ser ligeiramente mais rápida. Essa diferença, entretanto, é insignificante nos primeiros testes, uma vez que até a busca em 5000 números, ambos os algoritmos concluem o processo em menos de 1 segundo. A partir desse ponto, contudo, a diferença no tempo de execução entre os dois algoritmos torna-se mais evidente. Em muitos casos, a busca em profundidade executa todo o processo em aproximadamente metade do tempo necessário para a busca em largura.

Um aspecto relevante a ser destacado é a complexidade de tempo de ambos os algoritmos, que é $O(V + E)$. Essa relação pode ser facilmente observada nos dados coletados, tanto na Figura 1 quanto na Tabela 1. À medida que o número de vértices (números inseridos na árvore) e arestas aumenta, o tempo de execução também cresce.

Além disso, conforme mencionado anteriormente, não foram realizados testes para todos os arquivos disponibilizados devido ao longo tempo necessário para concluir a busca. Por exemplo, no caso dos arquivos contendo 125 mil valores, a busca levou um tempo total de $1742,134$ segundos. Considerando a necessidade de realizar seis execuções com os mesmos valores, o tempo total para concluir o experimento chega a quase três horas.

Tabela 1. Tempos médios de execução (segundos) para os algoritmos BFS e DFS, como diferentes tamanhos de busca

Arquivo	BFS (s)	DFS (s)
50	0,000400	0,000200
100	0,001200	0,000200
200	0,003600	0,000600
300	0,009000	0,001800
500	0,017200	0,003200
750	0,039800	0,012400
1000	0,094800	0,015000
1500	0,169600	0,044000
2000	0,336800	0,080400
3000	0,701400	0,213200
5000	0,915800	0,173600
7500	3,932000	1,178200
10000	8,822800	3,169400
12500	16,030400	6,714600
15000	28,683400	13,201400
20000	50,175400	26,212600
25000	76,693400	54,490600
30000	102,067400	81,190600
40000	156,542600	106,358600
50000	274,254800	281,508600
75000	605,336200	396,403200
100000	963,865200	554,061600
125000	1742,133800	1524,329000

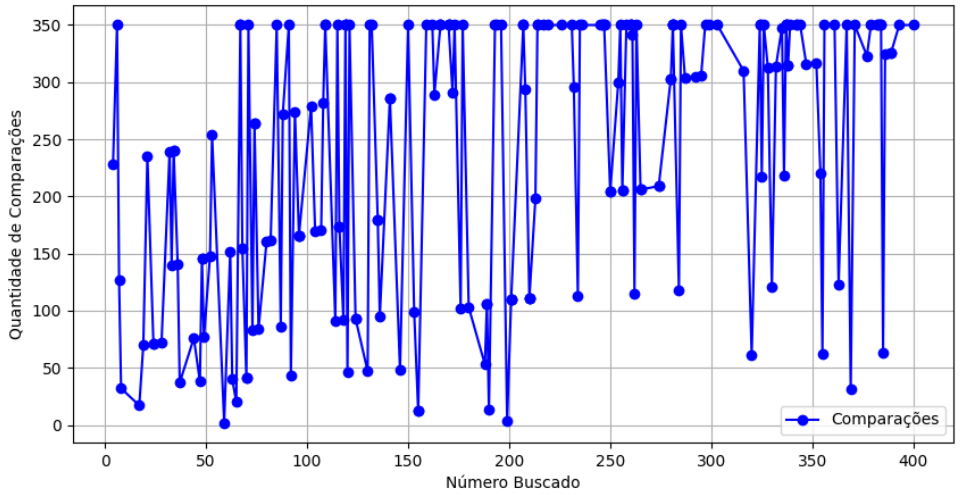
Embora ambos os algoritmos apresentem a mesma complexidade assintótica, o fato de a busca em largura apresentar um tempo de execução superior ao da busca em profundidade deve-se principalmente à forma como os valores são explorados. Enquanto a busca em profundidade percorre um caminho até o final antes de retroceder, tornando a estrutura do código mais simples e exigindo menos verificações, a busca em largura precisa manter todos os irmãos do nó atual armazenados em uma fila. Para isso, é necessário identificar cada irmão do nó atual e realizar verificações adicionais, o que torna o processo mais demorado à medida que o número de elementos na árvore aumenta.

Antes de analisar os gráficos que apresentam o número de comparações realizadas para encontrar cada valor dentro da árvore, é importante destacar que os dados foram ordenados do menor para o maior. Essa ordenação foi realizada após a execução dos algoritmos, com base nos dados armazenados nos arquivos $.csv$, para facilitar a visualização dos resultados.

Ao examinar o número de comparações realizadas por cada algoritmo de busca, observa-se um padrão presente em cada abordagem. A Figura 2 ilustra a quantidade de comparações necessárias para encontrar cada elemento em um arquivo contendo 200 valores, utilizando a busca em largura. Já a Figura 3 apresenta os resultados para o mesmo algoritmo, porém considerando um arquivo com 2000 valores. Em ambos os

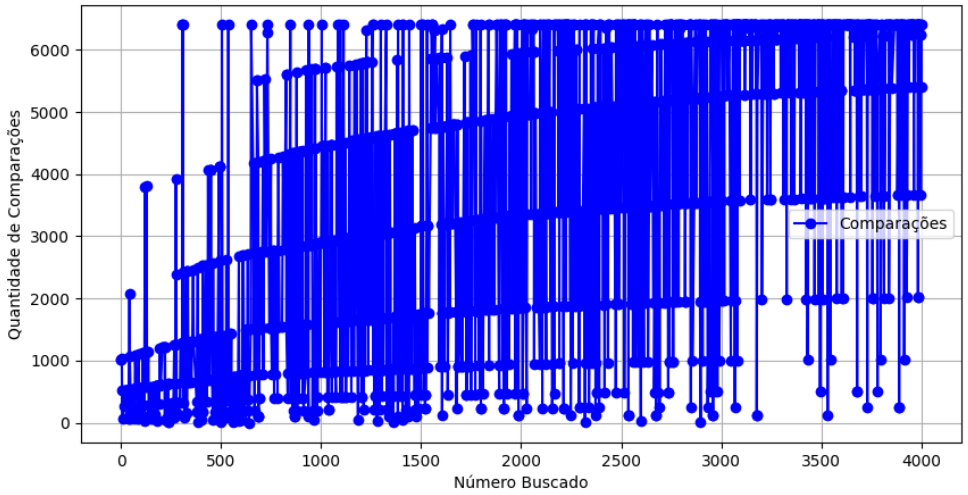
casos, percebe-se que o número de verificações depende diretamente da proximidade do valor buscado em relação à raiz da árvore, influenciando a quantidade de comparações realizadas. Esse padrão se mantém independentemente do tamanho do conjunto de dados.

Figura 2. Número de comparações realizadas pela busca em largura ao procurar os elementos do arquivo `200.txt` na árvore rubro-negra.



Fonte: Autoria própria

Figura 3. Número de comparações realizadas pela busca em largura ao procurar os elementos do arquivo `2000.txt` na árvore rubro-negra.

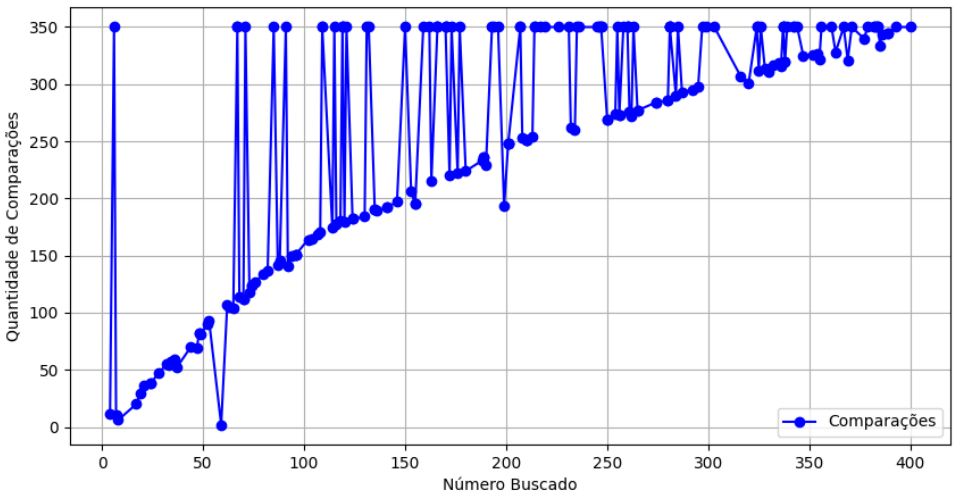


Fonte: Autoria própria

A análise das Figuras 4 e 5, que mostram respectivamente o número de

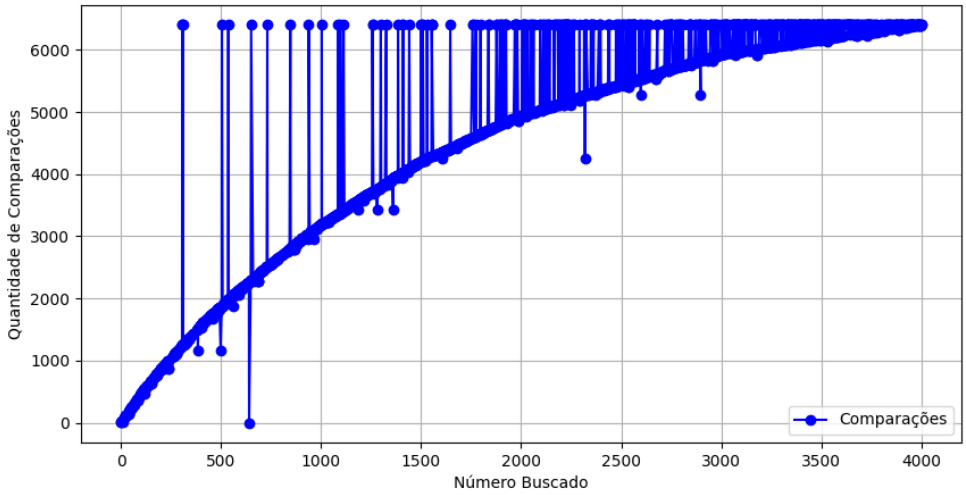
comparações realizadas pela busca em profundidade nos arquivos com 200 e 2000 valores, revela um comportamento distinto. Diferentemente da busca em largura, onde a distância até a raiz tem maior influência no número de comparações, na busca em profundidade, o principal padrão observado é que quanto maior o valor numérico, maior será o número de comparações necessárias para encontrá-lo. Da mesma forma, valores menores exigem menos comparações. Isso ocorre porque a busca em profundidade prioriza a exploração dos nós à esquerda antes de avançar para a direita. Em uma árvore ordenada, como a árvore Rubro-Negra, os valores menores que a raiz encontram-se no lado esquerdo, enquanto os maiores estão à direita. Dessa forma, a contagem de comparações tende a ser significativamente maior para números elevados na busca em profundidade do que na busca em largura.

Figura 4. Número de comparações realizadas pela busca em profundidade ao procurar os elementos do arquivo `200.txt` na árvore rubro-negra.



Fonte: Autoria própria

Figura 5. Número de comparações realizadas pela busca em profundidade ao procurar os elementos do arquivo `2000.txt` na árvore rubro-negra.



Fonte: Autoria própria

Um ponto interessante a ser destacado, na busca em profundidade é que é possível facilmente identificar qual é raiz da árvore, através do número de comparações feitas, isso pode ser observado tanto na Figura 4 e 5, ao qual apresenta no gráfico, um valor numérico que o número de comparações é igual 1, uma vez que é raiz e foi o primeiro a ser acessado. No caso da Figura 4 esse valor está entre as linhas 50 e 100 da coluna de valores buscados, já na Figura 5 o valor que corresponde a raiz está entre 500 e 1000

5. Considerações Finais

A partir dos testes apresentados, concluímos que o desempenho da busca em profundidade é superior ao da busca em largura . Esse resultado se deve ao fato de que a BFS, para avançar na busca, precisa sempre acessar todos os nós irmãos de cada nível, o que aumenta o número de operações e torna o algoritmo mais lento. Esse comportamento se torna ainda mais evidente em árvores maiores, onde a BFS pode levar praticamente o dobro do tempo em comparação à DFS.

Em relação ao número de comparações necessárias para encontrar um valor, observamos que a DFS tende a realizar mais comparações, especialmente para valores numéricos maiores do que a raiz, devido à sua estratégia de exploração. Ainda assim, mesmo com um número maior de comparações, a DFS se mostrou mais eficiente do que a BFS, que o número de comparações reflete a diferença da altura da árvore, do ponto do nó encontrado até a raiz, todavia necessita sempre que avança o nível, conhecer todos os seus vizinhos.

Como já destacado anteriormente, o site [for Geeks 2024a], menciona esse comportamento da BFS e DFS. No entanto, o site sugere que a BFS pode ser mais eficiente quando há poucos valores na árvore, algo que não foi confirmado empiricamente em nossos testes. Embora o tempo de execução da BFS tenha sido ligeiramente maior, na ordem

de milissegundos, nesse cenário específico com poucos valores, a BFS pode se mostrar um pouco mais interessante devido ao menor número de comparações realizadas em relação à DFS.

Com isso, podemos concluir que, no geral, a busca em profundidade apresenta um desempenho superior à busca em largura, especialmente em relação ao tempo de execução, apesar de ambas possuírem a mesma complexidade assintótica. No entanto, ambos os algoritmos se mostraram ineficientes para buscar grande quantidade de valores em árvores muito grandes, devido ao tempo significativo necessário para percorrê-las. Assim, para árvores binárias ordenadas, como as árvores rubro-negras, a abordagem que se mostra mais interessante é a busca binária simples, que possui um custo de $O(h)(h-altura)$ e evita percorrer caminhos irrelevantes para encontrar o valor desejado.

Referências

for Geeks, G. (2024a). Bfs vs dfs binary tree. Acessado em: 24 de março de 2025.

for Geeks, G. (2024b). Difference between bfs and dfs. Acessado em: 24 de março de 2025.

for Geeks, G. (2024c). Introduction to red-black tree. Acessado em: 24 de março de 2025.