UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE ENSINO SUPERIOR DO SERIDÓ
DEPARTAMENTO DE COMPUTAÇÃO E TECNOLOGIA
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Trabalho da Segunda Unidade: Análise de tempo de execução de algoritmos de busca

Luiz Miguel Santos Silva

Caicó - RN
Agosto de 2024

Luiz Miguel Santos Silva

Trabalho da Segunda Unidade: Análise de tempo de execução de algoritmos de busca

Relatório técnico apresentado ao curso de Bacharelado em Sistemas de Informação do Departamento de Computação e Tecnologia da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção de nota na disciplina de Estrutura de Dados.

Orientador(a): Prof.Dr João Paulo de Souza Medeiros.

Caicó - RN
Agosto de 2024

RESUMO

Este trabalho tem como objetivo apresentar uma análise dos tempos de execução dos algoritmos de busca: Árvore Binária, Árvore AVL e Tabela Hash.

Palavras-chave: Análises, Algoritmos, Tempos de execuções, Busca.

ABSTRACT

This work aims to present an analysis of the execution times of the sorting algorithms: Binary tree, AVL tree e Hash table.

Keywords: Analysis, Algorithms, Execution Times, Search.

SUMÁRIO

1 INTRODUÇÃO 7

2 ÁRVORE BINÁRIA 8
2.1 ALGORITMO DE BUSCA NA ÁRVORE BINÁRIA 9
2.2 GRÁFICOS DE TEMPO DE EXECUÇÃO DA BUSCA NA ÁRVORE BINÁRIA 10
2.2.1 Melhor Caso 10
2.2.2 Caso Médio 11
2.2.3 Pior Caso 12
2.2.4 Comparação do Melhor Caso e Caso Médio 13
2.2.5 Comparação de Todos os Casos 14
2.3 COMPLEXIDADE 14
2.4 ANÁLISE ASSINTÓTICA DA BUSCA NA ÁRVORE BINÁRIA . . . 15
2.5 ANÁLISE DE CUSTOS DA BUSCA NA ÁRVORE BINÁRIA 15

3 ÁRVORE AVL 16
3.1 ALGORITMO DE BUSCA NA ÁRVORE AVL 17
3.2 GRÁFICOS DE TEMPO DE EXECUÇÃO DA BUSCA NA ÁRVORE AVL 20
3.2.1 Melhor Caso 20
3.2.2 Caso Médio 21
3.2.3 Pior Caso 22
3.2.4 Comparação de Todos os Casos 23
3.3 COMPLEXIDADE 23
3.3.1 Melhor caso 23
3.3.2 Caso médio 23
3.3.3 Pior caso 24
3.4 ANÁLISE ASSINTÓTICA DA BUSCA NA ÁRVORE AVL 24
3.5 ANÁLISE DE CUSTOS DA BUSCA NA ÁRVORE AVL 24
3.5.1 Melhor caso 24
3.5.2 Caso médio 24
3.5.3 Pior caso 24

4 TABELA HASH 25
4.1 ALGORITMO DE BUSCA NA TABELA HASH 26

4.2 GRÁFICOS DE TEMPO DE EXECUÇÃO DA BUSCA NA TABELA HASH 29
4.2.1 Melhor Caso 29
4.2.2 Caso Médio 30
4.2.3 Pior Caso 31
4.2.4 Comparação do Melhor Caso e Caso Médio 32
4.2.5 Comparação de Todos os Casos 33
4.3 COMPLEXIDADE 33
4.3.1 Melhor caso 33
4.3.2 Caso médio 33
4.3.3 Pior caso 34
4.4 ANÁLISE ASSINTÓTICA DA BUSCA NA TABELA HASH 34
4.5 ANÁLISE DE CUSTOS DA BUSCA NA TABELA HASH 34
4.5.1 Melhor caso 34
4.5.2 Caso médio 34
4.5.3 Pior caso 34

5 CONCLUSÃO DA ANÁLISE 35

LISTA DE FIGURAS

Figura 1 – Algoritmo de busca na árvore binária implementado em C 9
Figura 2 – Tempo do melhor caso da busca na árvore binária 10
Figura 3 – Tempo de execução esperado da busca na árvore binária 11
Figura 4 – Tempo do pior caso da busca na árvore binária 12
Figura 5 – Comparação do tempo do melhor caso e do tempo de execução esperado da busca na árvore binária 13
Figura 6 – Comparação do tempo de execução de todos os casos da busca na árvore binária 14
Figura 7 – Algoritmo de busca na árvore avl implementado em C 17
Figura 8 – Algoritmo de busca na árvore avl implementado em C 18
Figura 9 – Algoritmo de busca na árvore avl implementado em C 19
Figura 10 – Tempo do melhor caso da busca na árvore avl 20
Figura 11 – Tempo de execução esperado da busca na árvore avl 21
Figura 12 – Tempo do pior caso da busca na árvore avl 22
Figura 13 – Comparação do tempo de execução de todos os casos da busca na árvore avl 23
Figura 14 – Algoritmo de busca na tabela hash implementado em C 26
Figura 15 – Algoritmo de busca na tabela hash implementado em C 27
Figura 16 – Algoritmo de busca na tabela hash implementado em C 28
Figura 17 – Tempo do melhor caso da busca na tabela hash 29
Figura 18 – Tempo de execução esperado da busca na tabela hash 30
Figura 19 – Tempo do pior caso da busca na tabela hash 31
Figura 20 – Comparação do tempo do melhor caso e do tempo de execução esperado da busca na tabela hash 32
Figura 21 – Comparação do tempo de execução de todos os casos da busca na tabela hash 33
Figura 22 – Gráfico que compara o tempo de execução esperado de todos os algoritmos analisados 36

1 INTRODUÇÃO

Trabalho da disciplina de Estrutura de Dados, referente à segunda unidade. Este trabalho se pauta no estudo dos temas e algoritmos discutidos em sala de aula, com objetivo de fornecer uma análise detalhada dos mesmos.

2 ÁRVORE BINÁRIA

O algoritmo de busca em uma árvore binária é uma abordagem eficiente para localizar um elemento em uma estrutura de dados hierárquica. Nesse algoritmo, a árvore é percorrida a partir do nó raiz, comparando o valor desejado com o valor do nó atual. Se o valor buscado for menor, a busca continua na subárvore à esquerda; se for maior, a busca continua na subárvore à direita. Esse processo é repetido recursivamente até que o valor seja encontrado ou até que a busca alcance um nó nulo, indicando que o elemento não está presente na árvore.

2.1 ALGORITMO DE BUSCA NA ÁRVORE BINÁRIA

```c
typedef struct node {
    int value;
    struct node* left;
    struct node* right;
} Node;

Node* create_node(int value) {
    Node* new_node = (Node*)malloc(sizeof(Node));
    new_node->value = value;
    new_node->left = NULL;
    new_node->right = NULL;
    return new_node;
}

void insert(Node **root, int value)
{
    if ((*root) == NULL)
    {
        (*root) = create_node(value);
    }

    else
    {
        if ((*root)->value > value){
            insert(&((*root)->left), value);
        }

        else{
            insert(&((*root)->right), value);
        }
    }
}

Node* search(Node* root, int number)
{
    if (root != NULL) {
        if (root->value == number){
            return root;
        }

        else if (number < root->value) {
            return search(root->left, number);

        }
        return search(root->right, number);
    }

    return NULL;
}
```

Figura 1 – Algoritmo de busca na árvore binária implementado em C

2.2 GRÁFICOS DE TEMPO DE EXECUÇÃO DA BUSCA NA ÁRVORE BINÁRIA

2.2.1 MELHOR CASO

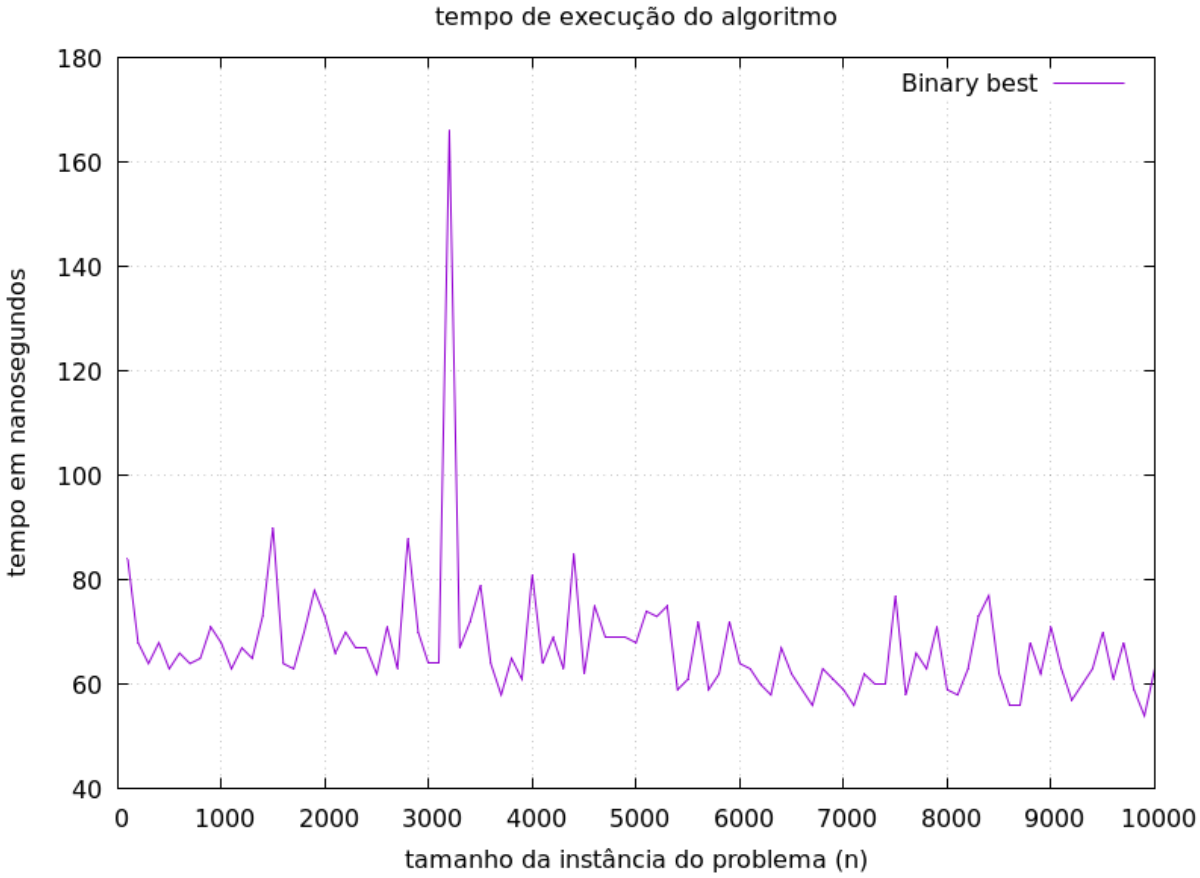

Figura 2 – Tempo do melhor caso da busca na árvore binária

2.2.2 CASO MÉDIO

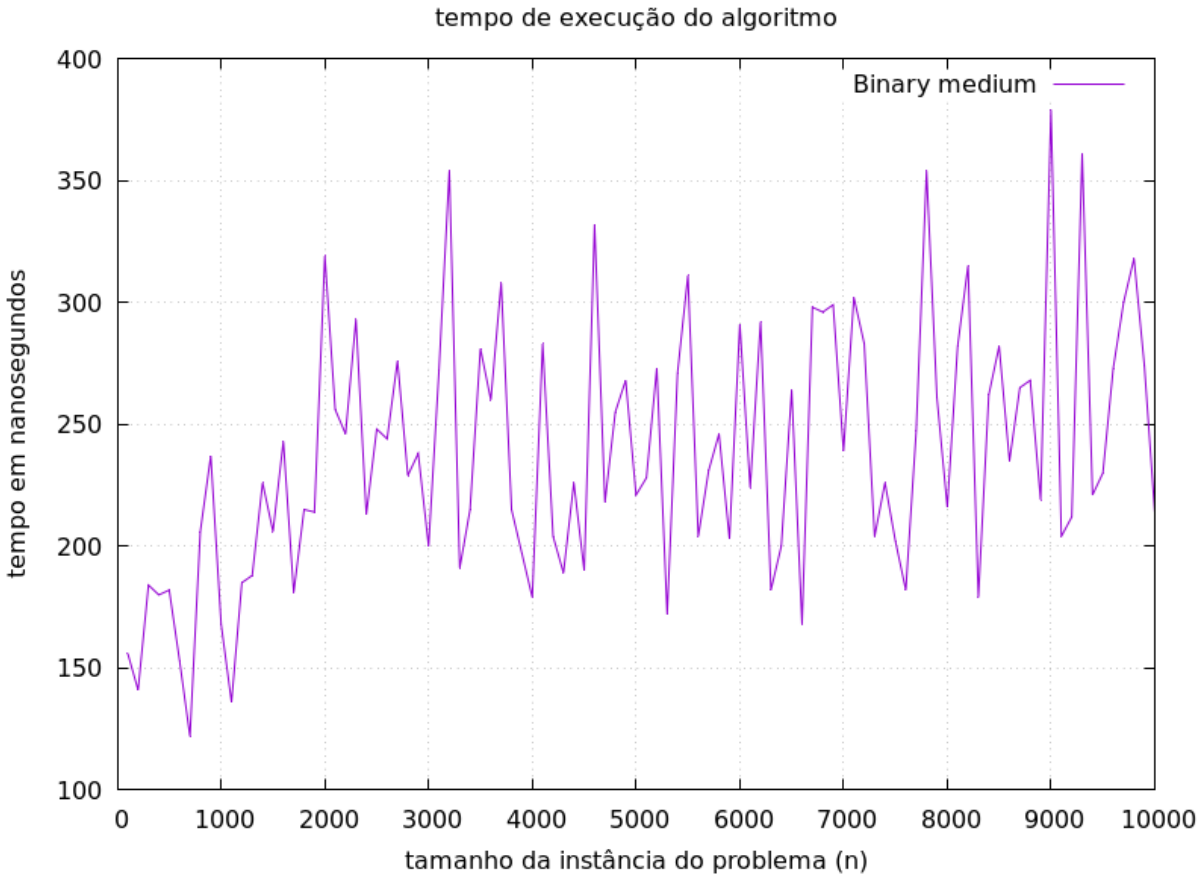

Figura 3 – Tempo de execução esperado da busca na árvore binária

2.2.3 PIOR CASO

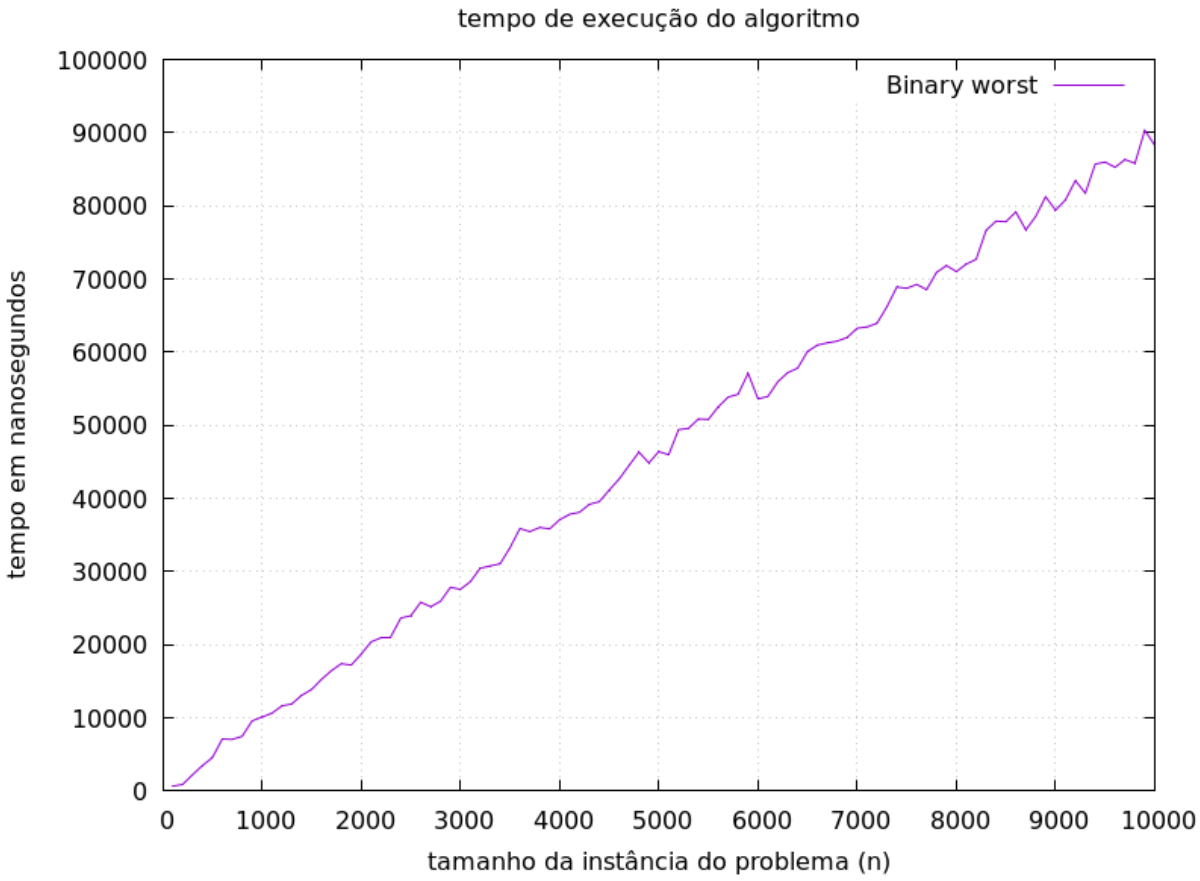

Figura 4 – Tempo do pior caso da busca na árvore binária

2.2.4 COMPARAÇÃO DO MELHOR CASO E CASO MÉDIO

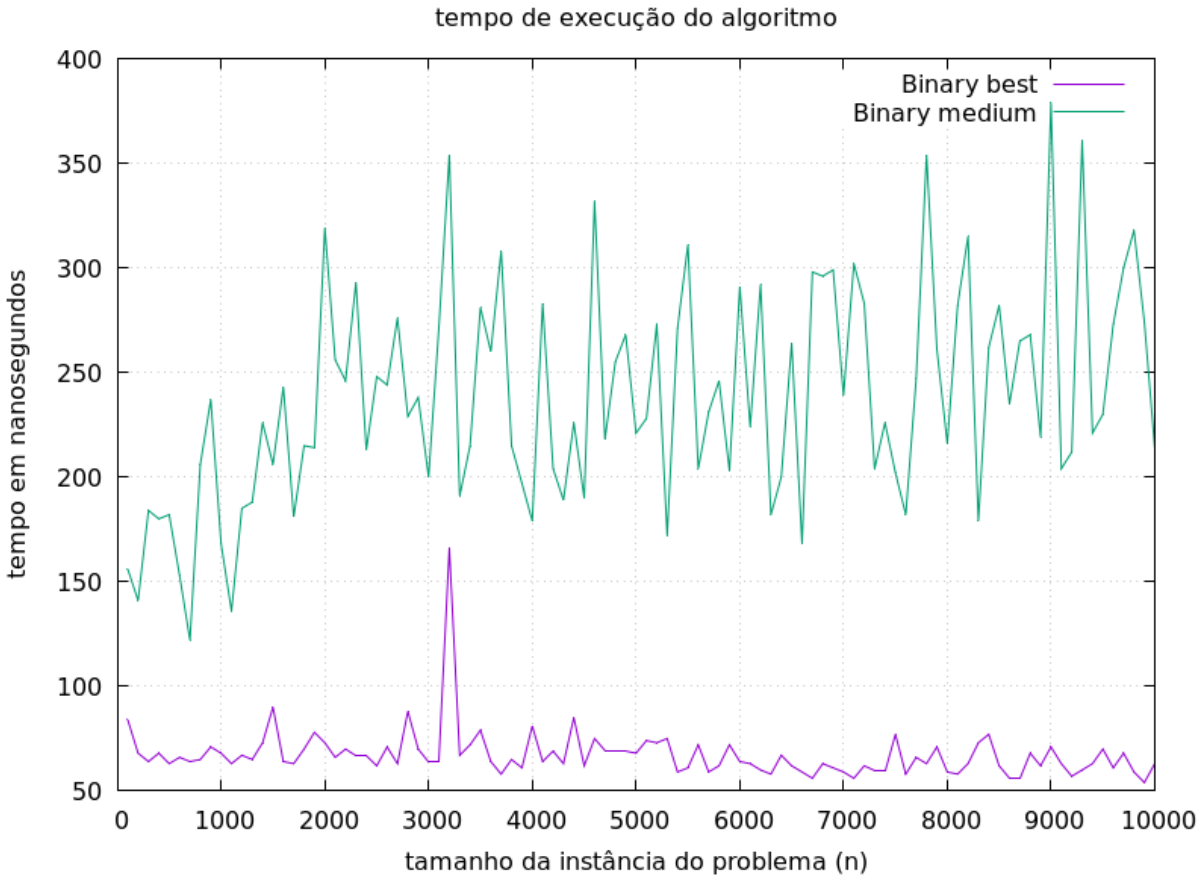

Figura 5 – Comparação do tempo do melhor caso e do tempo de execução esperado da busca na árvore binária

2.2.5 COMPARAÇÃO DE TODOS OS CASOS

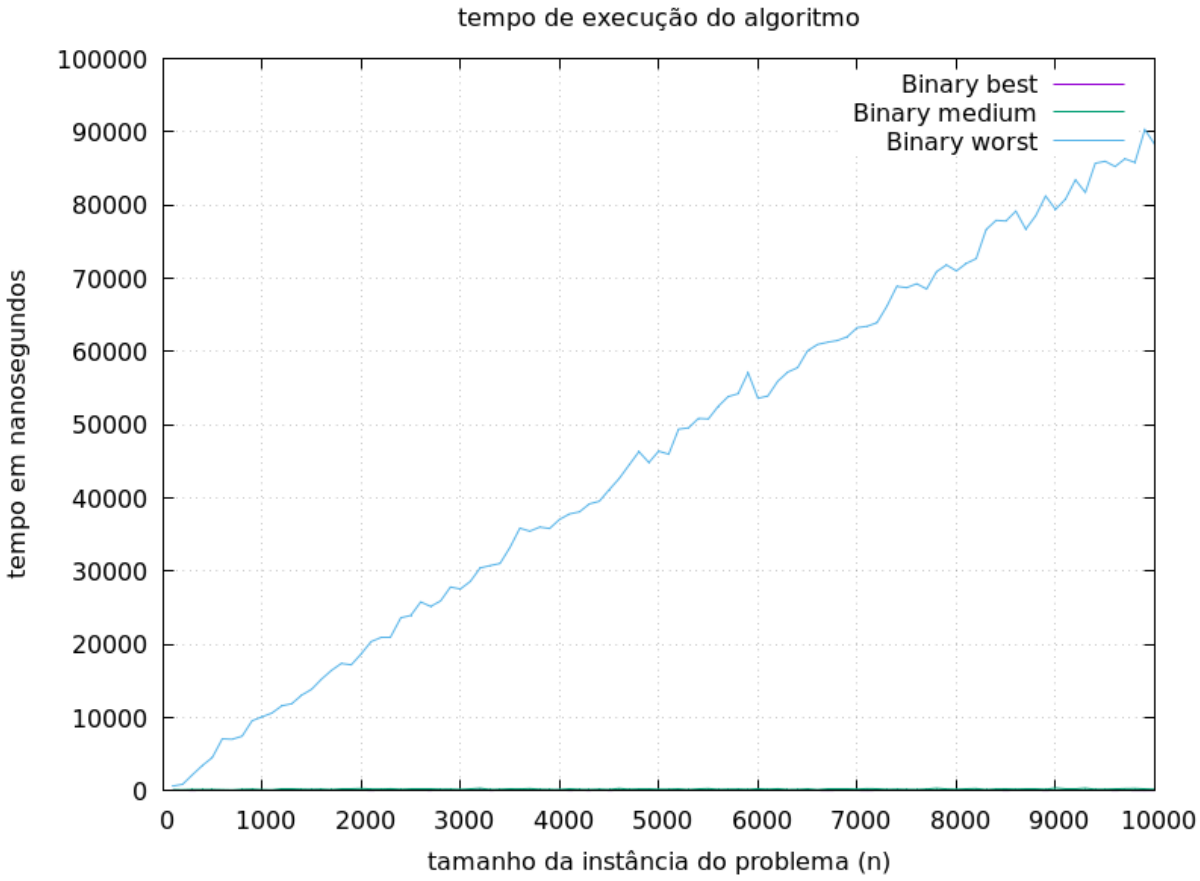

Figura 6 – Comparação do tempo de execução de todos os casos da busca na árvore binária

2.3 COMPLEXIDADE

A busca em uma árvore binária não possui um caso melhor ou pior claramente definido, pois seu desempenho depende diretamente da estrutura da árvore. Em uma árvore balanceada, a busca será eficiente, com complexidade logarítmica. Entretanto, em uma árvore desequilibrada, a busca pode se degradar para uma complexidade linear, semelhante à de uma lista encadeada. O tempo de execução do algoritmo será diretamente influenciado pela profundidade da árvore e pela distribuição dos nós. Quanto mais balanceada a árvore, mais eficiente será a busca; por outro lado, uma árvore altamente desequilibrada pode levar a tempos de busca significativamente maiores.

2.4 ANÁLISE ASSINTÓTICA DA BUSCA NA ÁRVORE BINÁRIA

Caso	**Complexidade**
Melhor Caso	$\Theta(1)$
Caso Médio	$\Theta(\log_2 n)$
Pior Caso	$\Theta(n)$

2.5 ANÁLISE DE CUSTOS DA BUSCA NA ÁRVORE BINÁRIA

Como mencionado anteriormente, o algoritmo de busca em uma árvore binária apresenta um comportamento que varia conforme a estrutura da árvore, mas não possui um caso estritamente melhor ou pior, uma vez que seu desempenho está atrelado à forma como os dados estão organizados na árvore. A busca consiste em percorrer a árvore a partir do nó raiz, comparando o valor desejado com os valores dos nós visitados, e decidindo se deve-se continuar a busca à esquerda ou à direita da árvore. Se a árvore estiver balanceada, a busca será eficiente, pois a profundidade da árvore é minimizada, o que resulta em um tempo de execução logarítmico. No entanto, em uma árvore desequilibrada, especialmente aquelas que se assemelham a uma lista encadeada, a busca pode ter seu tempo de execução aumentado, aproximando-se do tempo linear, semelhante à busca em uma estrutura linear.

3 ÁRVORE AVL

O algoritmo de busca em uma árvore AVL, inicia a partir do nó raiz e percorre a árvore de acordo com o valor a ser encontrado. A busca segue para a subárvore à esquerda se o valor desejado for menor que o valor do nó atual, ou para a subárvore à direita se for maior. A diferença crucial ao buscar em uma árvore AVL é que, devido ao balanceamento garantido após cada operação de inserção ou remoção, a árvore permanece balanceada, mantendo sua altura logarítmica em relação ao número de nós. O que torna sua execução mais otimizada.

3.1 ALGORITMO DE BUSCA NA ÁRVORE AVL

```c
typedef struct node {
    int value;
    unsigned int height;

    struct node* left;
    struct node* right;
    struct node* parent;
} Node;

int height(Node* node) {
    return (node == NULL) ? 0 : node->height;
}

int max(int a, int b) {
    return (a > b) ? a : b;
}

Node* create_node(int value) {
    Node* new_node = (Node*)malloc(sizeof(Node));

    new_node->value = value;
    new_node->height = 1;

    new_node->left = NULL;
    new_node->right = NULL;
    new_node->parent = NULL;

    return new_node;
}

int getBalance(Node* node) {
    return (node == NULL) ? 0 : height(node->left) - height(node->right);
}

Node* find_root(Node* root) {
    if (root->parent != NULL) {
        return find_root(root->parent);
    } else {
        return root;
    }
}

void calculate_height(Node* root) {
    if (root != NULL) {
        root->height = 1 + max(height(root->left), height(root->right));
        calculate_height(root->parent);
    }
}
```

Figura 7 – Algoritmo de busca na árvore avl implementado em C

```c
Node* rotate_left(Node* x) {
    Node* y = x->right;
    Node* C_TREE = y->left;

    y->left = x;
    x->right = C_TREE;

    y->parent = x->parent;
    x->parent = y;

    Node* parent = y->parent;
    if (parent != NULL) {
        if(parent->left == x) {
            parent->left = y;
        } else {
            parent->right = y;
        }
    }

    if(C_TREE != NULL) {
        C_TREE->parent = x;
    }

    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    return y;
}

Node* rotate_right(Node* x) {
    Node* y = x->left;
    Node* C_TREE = y->right;

    y->right = x;
    x->left = C_TREE;

    y->parent = x->parent;
    x->parent = y;

    Node* parent = y->parent;
    if (parent != NULL) {
        if(parent->left == x) {
            parent->left = y;
        } else {
            parent->right = y;
        }
    }

    if(C_TREE != NULL) {
        C_TREE->parent = x;
    }

    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    return y;
}
```

Figura 8 – Algoritmo de busca na árvore avl implementado em C

```c
void balance_tree(Node* root) {
    int balance = getBalance(root);

    // Caso Esquerda-Esquerda (LL) ou Caso 01
    if(balance > 1 && getBalance(root->left) >= 0) {
        root = rotate_right(root);
    }

    // Caso Direita-Direita (RR) ou Caso 02
    if(balance < -1 && getBalance(root->right) <= 0) {
        root = rotate_left(root);
    }

    // Caso Esquerda-Direita (LR) ou Caso 03
    if(balance > 1 && getBalance(root->left) < 0) {
        root->left = rotate_left(root->left);
        root = rotate_right(root);
    }

    // Caso Direita-Esquerda (RL) ou Caso 04
    if(balance < -1 && getBalance(root->right) > 0) {
        root->right = rotate_right(root->right);
        root = rotate_left(root);
    }
}

void insert_node(Node** root, int value) {
    if ((*root) != NULL) {
        if((*root)->value > value) {
            insert_node(&(*root)->left, value);
            (*root)->left->parent = (*root);
        } else {
            insert_node(&(*root)->right, value);
            (*root)->right->parent = (*root);
        }

        calculate_height((*root));

        balance_tree((*root));
    } else {
        (*root) = create_node(value);
    }
}

Node* binary_search(Node* root, int number) {
    if (root != NULL) {
        if(root->value == number) {
            return root;
        }

        if(root->value > number) {
            return binary_search(root->left, number);
        }

        return binary_search(root->right, number);
    }
    return NULL;
}
```

Figura 9 – Algoritmo de busca na árvore avl implementado em C

3.2 GRÁFICOS DE TEMPO DE EXECUÇÃO DA BUSCA NA ÁRVORE AVL

3.2.1 MELHOR CASO

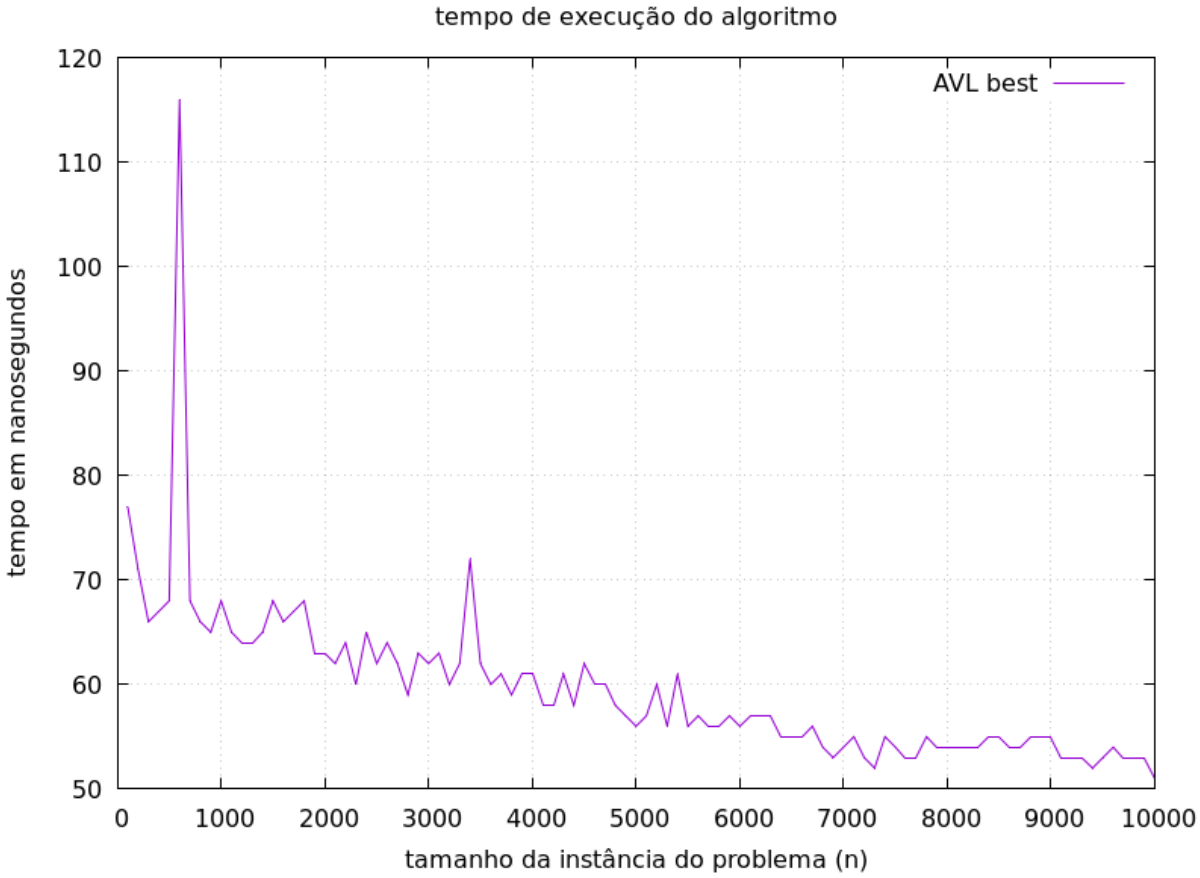

Figura 10 – Tempo do melhor caso da busca na árvore avl

3.2.2 CASO MÉDIO

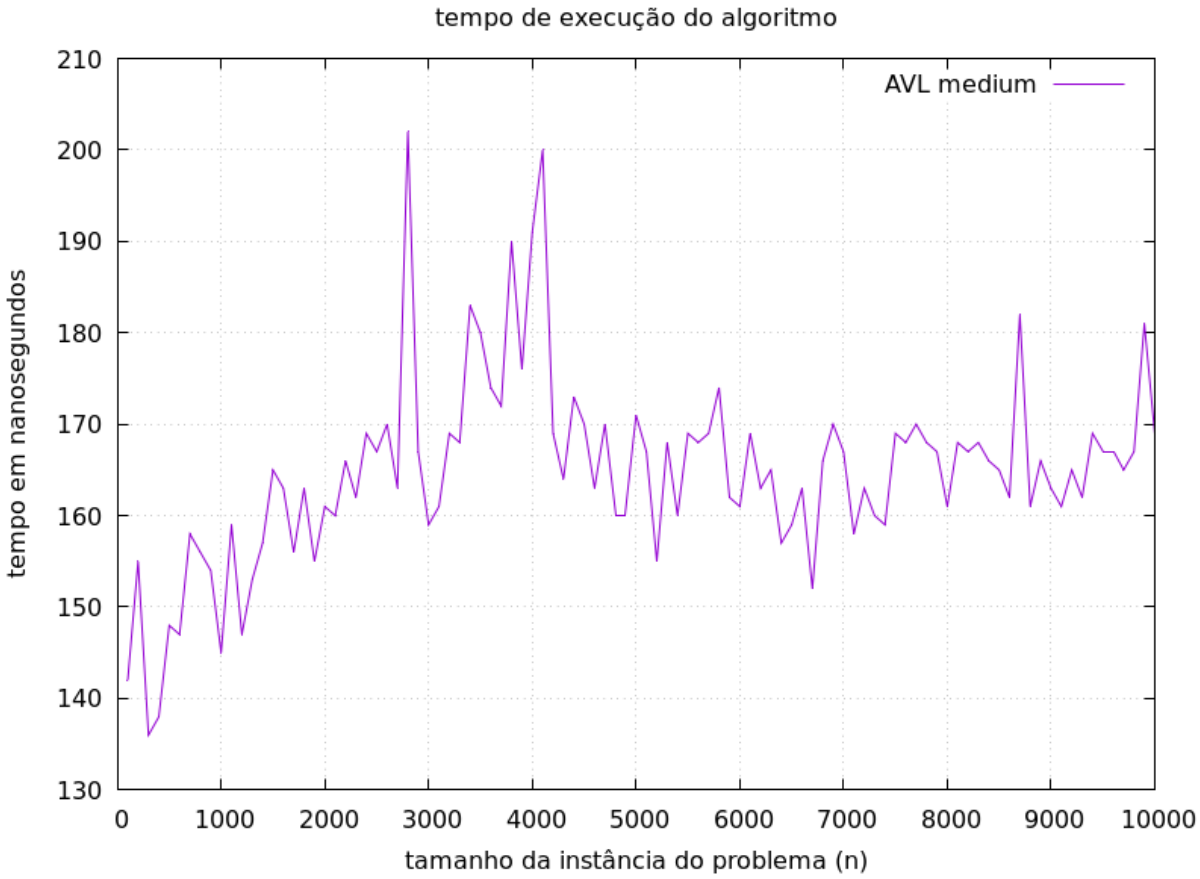

Figura 11 – Tempo de execução esperado da busca na árvore avl

3.2.3 PIOR CASO

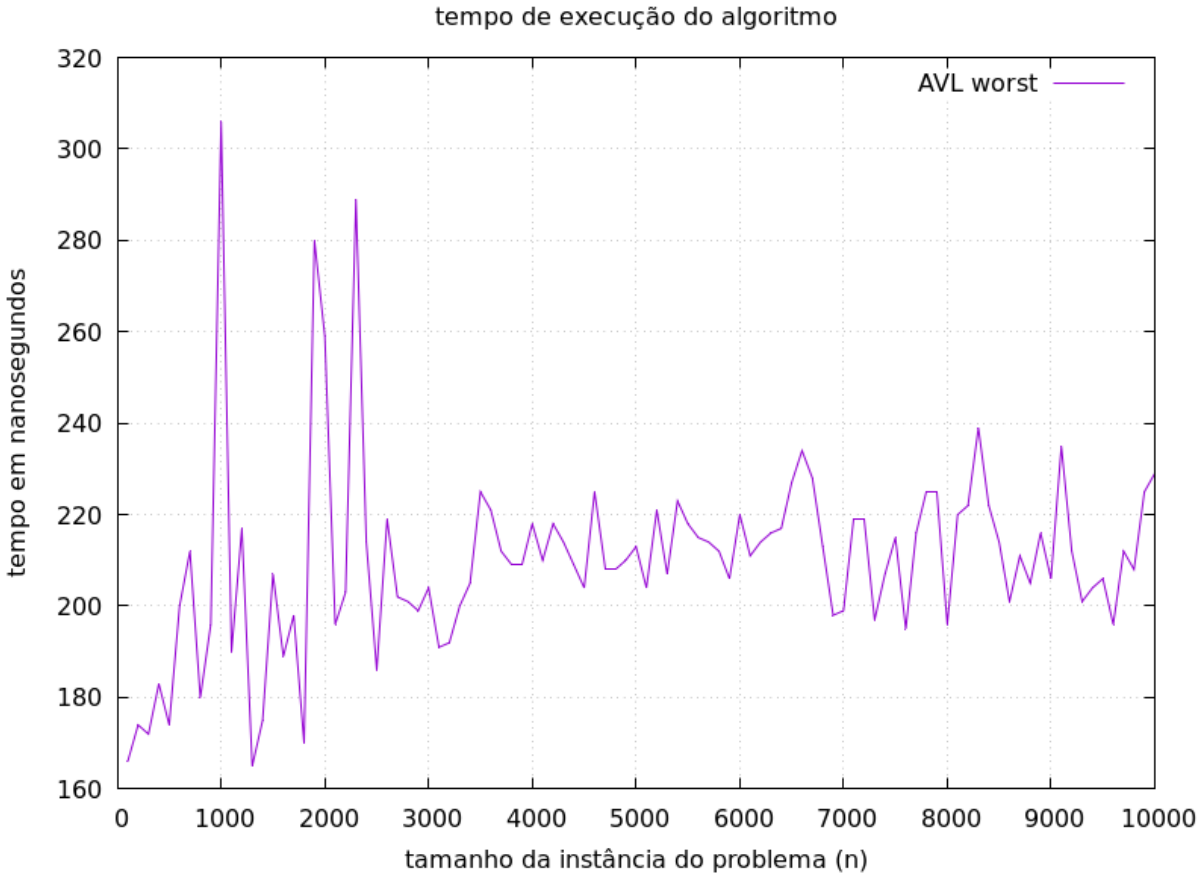

Figura 12 – Tempo do pior caso da busca na árvore avl

3.2.4 COMPARAÇÃO DE TODOS OS CASOS

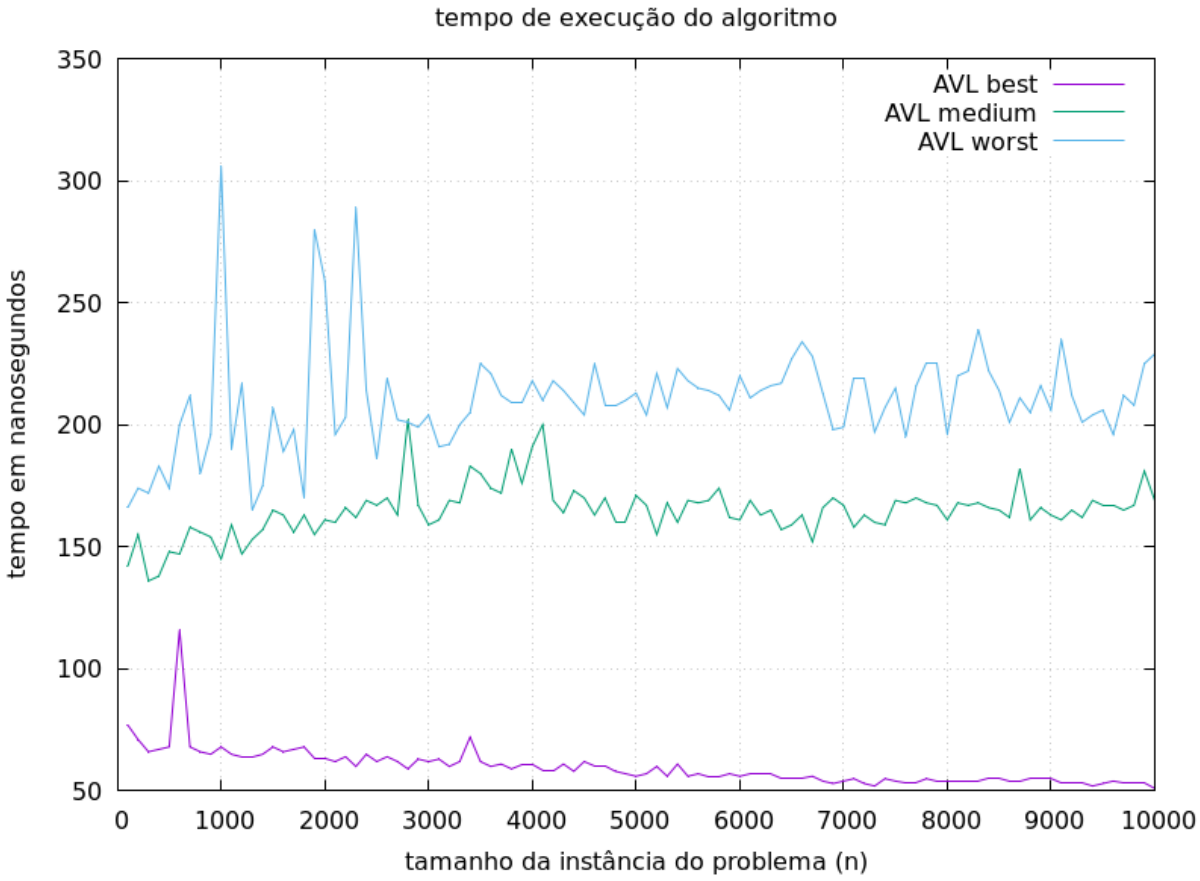

Figura 13 – Comparação do tempo de execução de todos os casos da busca na árvore avl

3.3 COMPLEXIDADE

3.3.1 MELHOR CASO

A configuração inicial para que ocorra o melhor caso da busca na árvore avl é quando o valor que você está procurando está na raiz. Nessa configuração, a busca percorre o mínimo de níveis possíveis, já que o valor é encontrado logo na primeira busca, resultando em um tempo de execução constante.

3.3.2 CASO MÉDIO

O caso médio para o algoritmo de busca em uma árvore AVL ocorre quando a árvore está perfeitamente balanceada, ou seja, para cada nó, as subárvores esquerda e direita têm alturas iguais ou diferença entre altura à esquerda e à direita $<= 1$. Nessa configuração, a busca percorre o mínimo de níveis possíveis, resultando em um tempo de execução logarítmico. Isso se dá porque a altura da árvore é minimizada, proporcionando a busca mais eficiente possível.

3.3.3 PIOR CASO

O pior caso para a busca em uma árvore AVL ocorre em uma situação extrema onde, após uma série de inserções e remoções, a árvore chega ao limite de balanceamento permitido pelas rotações AVL. Mesmo assim, devido ao mecanismo de balanceamento da árvore, a altura da árvore permanece logarítmica em relação ao número de nós, resultando em um tempo de execução também logarítmico.

3.4 ANÁLISE ASSINTÓTICA DA BUSCA NA ÁRVORE AVL

Caso	**Complexidade**
Melhor Caso	$\Theta(1)$
Caso Médio	$\Theta(\log_2 n)$
Pior Caso	$\Theta(\log_2 n)$

3.5 ANÁLISE DE CUSTOS DA BUSCA NA ÁRVORE AVL

3.5.1 MELHOR CASO

No melhor caso da busca em uma árvore AVL, o tempo de execução esperado é constante. Isso ocorre quando o valor busscado está na raiz da árvore, permitindo que o algoritmo encontre o elemento desejado na primeira recursão. Nesse caso, o custo é significativamente inferior ao dos outros casos.

3.5.2 CASO MÉDIO

No caso médio da busca em uma árvore AVL, o tempo de execução esperado é logarítmico. A árvore perfeitamente balanceada mantém uma estrutura equilibrada para garantir uma busca eficiente. O custo nesse caso é maior do que no melhor caso, mas ainda é um pouco mais eficiente do que o pior caso.

3.5.3 PIOR CASO

No pior caso da busca em uma árvore AVL, o tempo de execução esperado também é logarítmico. Mesmo que a árvore atinja o limite de seu balanceamento, as rotações garantem que a altura da árvore permaneça controlada, evitando a degeneração para um comportamento linear. Nessa situação, o custo é superior ao do melhor caso e ligeiramente superior ao do caso médio, mas ainda muito eficiente comparado a árvores não balanceadas.

4 TABELA HASH

No algoritmo de busca em uma tabela hash, o dado a ser encontrado é mapeado para um índice específico da tabela utilizando uma função hash. Essa função transforma a chave de busca em um número, que corresponde à posição na tabela onde o dado deve estar armazenado. Se a função hash for bem projetada e não houver colisões, o dado é encontrado diretamente na posição calculada, resultando em uma busca extremamente rápida. Caso ocorra uma colisão, ou seja, dois ou mais dados sejam mapeados para o mesmo índice, a tabela hash emprega técnicas de resolução de colisões, como encadeamento ou endereçamento aberto, para continuar a busca. A operação de busca se completa ao encontrar o dado ou ao determinar que ele não está presente na tabela. Essa lógica se aplica a todas as operações de busca, aproveitando a eficiência da tabela hash para acessar os dados rapidamente.

4.1 ALGORITMO DE BUSCA NA TABELA HASH

```c
struct node {
    int value;
    struct node* next;
};

struct hash_table {
    struct node** table;
    unsigned int size;
    unsigned int n;
};

struct hash_table* createHashTable(int size) {
    struct hash_table* hashTable = (struct hash_table*)malloc(sizeof(struct hash_table));

    hashTable->size = size;
    hashTable->n = 0;
    hashTable->table = (struct node**)malloc(sizeof(struct node*) * size);

    for (int i = 0; i < size; i++) {
        hashTable->table[i] = NULL;
    }

    return hashTable;
}

int hashFunction(int key, int size) {
    return key % size;
}

float loadFactor(struct hash_table* hashTable) {
    return (float)hashTable->n / hashTable->size;
}

struct node* create_node(int value) {
    struct node* node = (struct node*)malloc(sizeof(struct node));

    node->value = value;
    node->next = NULL;

    return node;
}
```

Figura 14 – Algoritmo de busca na tabela hash implementado em C

```c
void rehash(struct hash_table* hashTable) {
    int oldSize = hashTable->size;
    int newSize = oldSize * 2;
    struct node** oldTable = hashTable->table;

    hashTable->size = newSize;
    hashTable->table = (struct node**)malloc(sizeof(struct node*) * newSize);

    for (int i = 0; i < newSize; i++) {
        hashTable->table[i] = NULL;
    }

    for (int i = 0; i < oldSize; i++) {
        struct node* item = oldTable[i];
        while (item != NULL) {
            struct node* next = item->next;
            int index = hashFunction(item->value, newSize);
            item->next = hashTable->table[index];
            hashTable->table[index] = item;
            item = next;
        }
    }

    free(oldTable);
}

void insert(struct hash_table* hashTable, int value) {
    if (loadFactor(hashTable) >= 1.0) {
        rehash(hashTable);
    }

    int index = hashFunction(value, hashTable->size);

    struct node* newNode = (struct node*)malloc(sizeof(struct node));
    newNode->value = value;
    newNode->next = NULL;

    if (hashTable->table[index] == NULL) {
        hashTable->table[index] = newNode;
    } else {
        struct node* current = hashTable->table[index];
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
    }

    hashTable->n++;
}

unsigned int hash(int value, int m) {
    return value % m < 0 ? (value % m) + m : value % m;
}
```

Figura 15 – Algoritmo de busca na tabela hash implementado em C

```c
void insert_best(struct hash_table* hashTable, int value) {
    if (loadFactor(hashTable) >= 1.0) {
        rehash(hashTable);
    }

    int index = hashFunction(value, hashTable->size);

    // Inserir sempre no início da lista, para o melhor caso.
    struct node* newNode = create_node(value);
    newNode->next = hashTable->table[index];
    hashTable->table[index] = newNode;

    hashTable->n++;
}

void insert_worst(struct hash_table* hashTable, int value) {
    unsigned int key = hash(value, hashTable->size);

    // Como ser é o pior caso, então não há o rehash.
    if(1) {
        unsigned int index = hash(key, hashTable->size);

        struct node* new_node = create_node(value);

        if(hashTable->table[index] == NULL) {
            hashTable->table[index] = new_node;
        } else {
            struct node* index_node = hashTable->table[index];
            while(index_node->next != NULL) {
                index_node = index_node->next;
            }
            index_node->next = new_node;
        }

        hashTable->n++;

    } else {
        rehash(hashTable);
        insert_worst(hashTable, value);
    }
}

int search(struct hash_table* hashTable, int key) {
    int index = hashFunction(key, hashTable->size);
    struct node* item = hashTable->table[index];

    while (item != NULL) {
        if (item->value == key) {
            return 1;
        }
        item = item->next;
    }

    return 0;
}
```

Figura 16 – Algoritmo de busca na tabela hash implementado em C

4.2 GRÁFICOS DE TEMPO DE EXECUÇÃO DA BUSCA NA TABELA HASH

4.2.1 MELHOR CASO

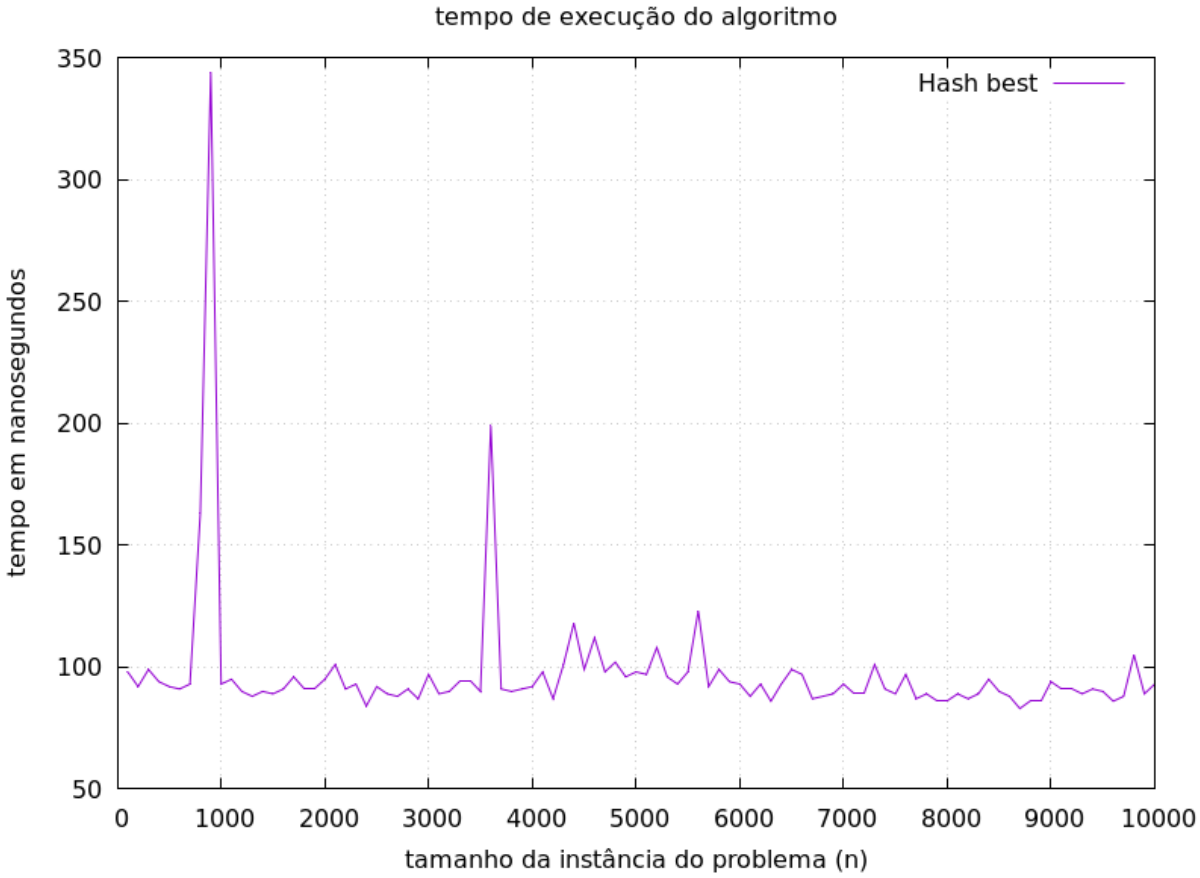

Figura 17 – Tempo do melhor caso da busca na tabela hash

4.2.2 CASO MÉDIO

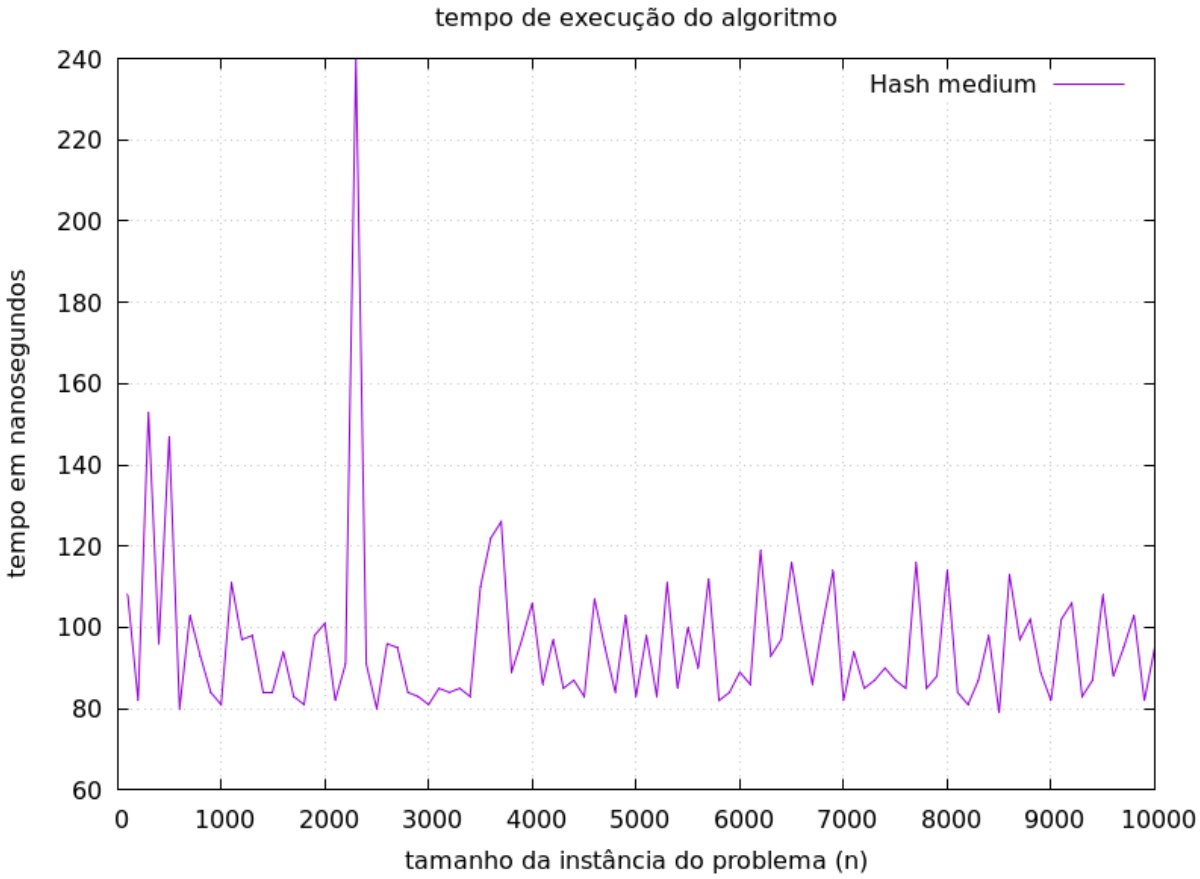

Figura 18 – Tempo de execução esperado da busca na tabela hash

4.2.3 PIOR CASO

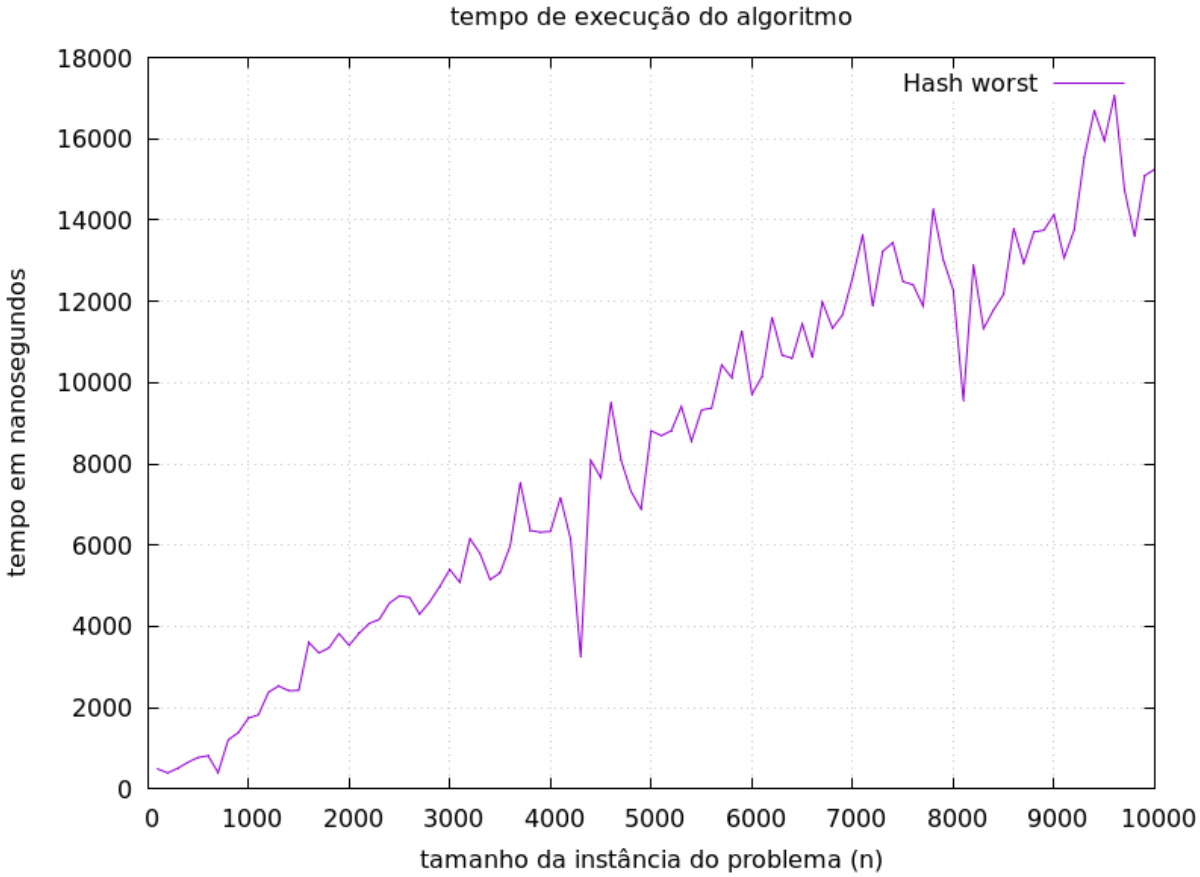

Figura 19 – Tempo do pior caso da busca na tabela hash

4.2.4 COMPARAÇÃO DO MELHOR CASO E CASO MÉDIO

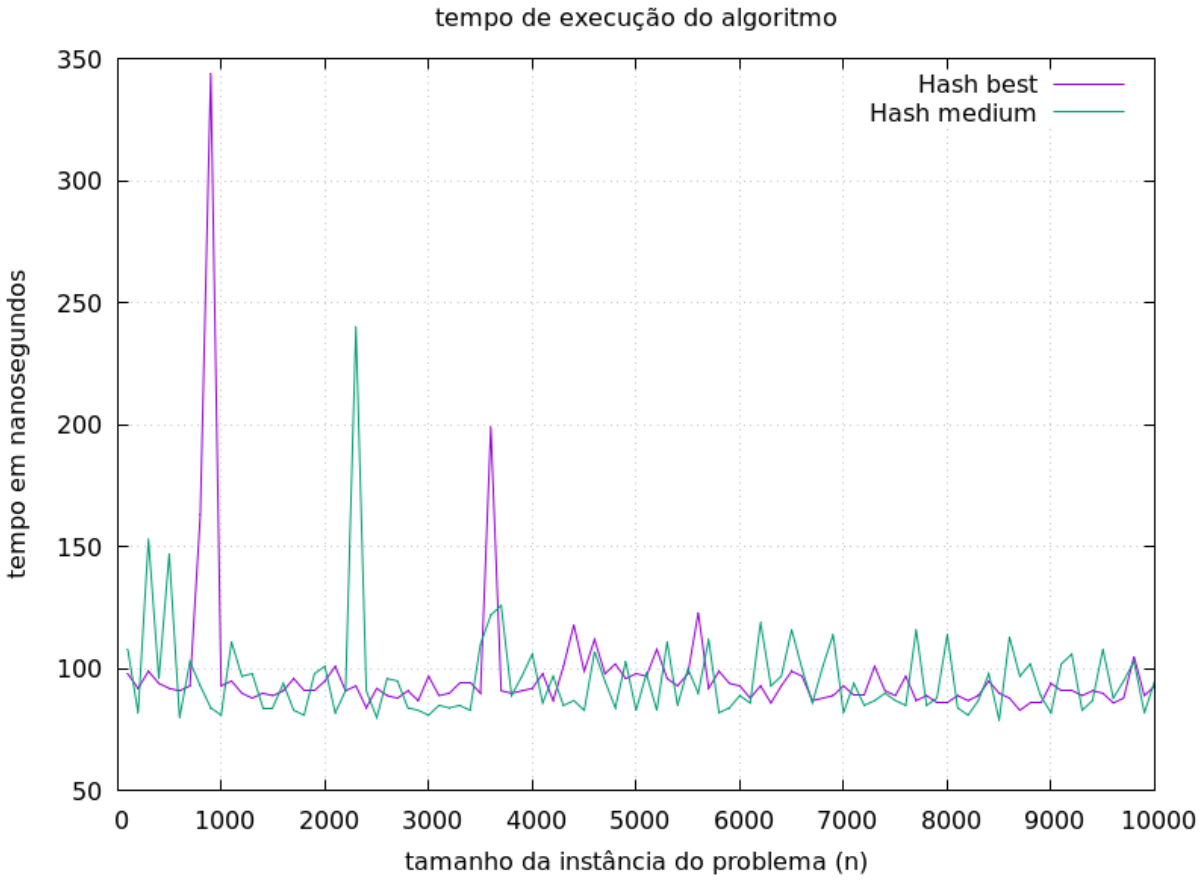

Figura 20 – Comparação do tempo do melhor caso e do tempo de execução esperado da busca na tabela hash

4.2.5 COMPARAÇÃO DE TODOS OS CASOS

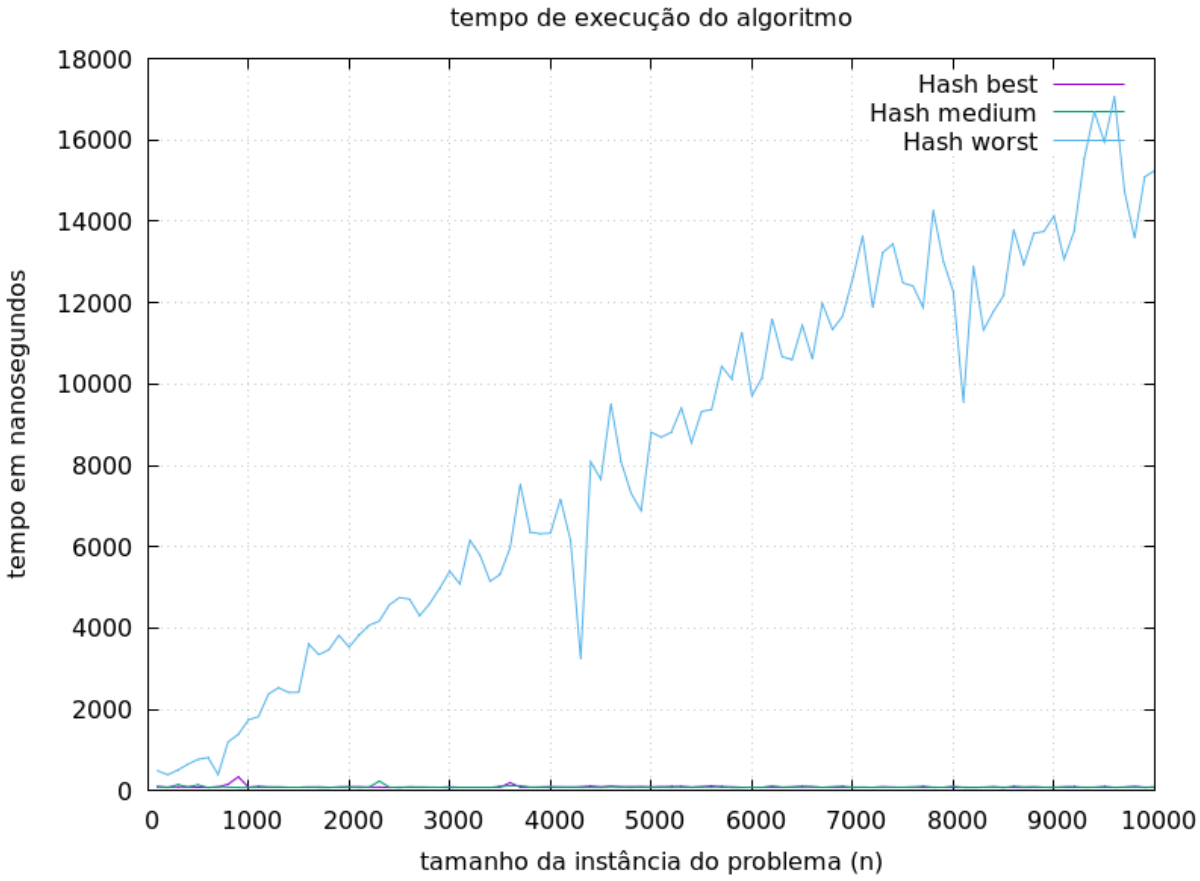

Figura 21 – Comparação do tempo de execução de todos os casos da busca na tabela hash

4.3 COMPLEXIDADE

4.3.1 MELHOR CASO

O melhor caso para a busca em uma tabela hash ocorre quando a função hash mapeia a chave diretamente para um índice único na tabela, sem colisões. Nesse cenário, o dado é encontrado imediatamente na primeira posição verificada, resultando em um tempo de execução constante.

4.3.2 CASO MÉDIO

O caso médio para a busca em uma tabela hash ocorre quando a função hash distribui as chaves uniformemente, mas ainda há uma pequena quantidade de colisões. Nessas situações, embora existam colisões, o tempo de execução esperado ainda é constante. Pois as técnicas de resolução de colisões são eficazes em manter o desempenho próximo do ideal.

4.3.3 PIOR CASO

O pior caso para a busca em uma tabela hash ocorre quando múltiplas chaves são mapeadas para o mesmo índice, resultando em várias colisões. Se muitas colisões ocorrerem, o desempenho pode degradar para tempo linear. Já que a busca pode se tornar semelhante à de percorrer uma lista encadeada ou uma sequência de verificações em endereçamento aberto.

4.4 ANÁLISE ASSINTÓTICA DA BUSCA NA TABELA HASH

Caso	**Complexidade**
Melhor Caso	$\Theta(1)$
Caso Médio	$\Theta(1)$
Pior Caso	$\Theta(n)$

4.5 ANÁLISE DE CUSTOS DA BUSCA NA TABELA HASH

4.5.1 MELHOR CASO

Como dito anteriormente, o melhor caso da busca em uma tabela hash ocorre quando a função hash mapeia a chave diretamente para um índice único na tabela, sem colisões, permitindo que o algoritmo encontre o elemento desejado na primeira tentativa. Nesse caso, o tempo de execução esperado é constante e o custo é inferior ao dos outros cenários.

4.5.2 CASO MÉDIO

No caso médio da busca em uma tabela hash, o tempo de execução esperado ainda é constante, mesmo com a presença de algumas colisões. A função hash distribui as chaves de forma relativamente uniforme, e as colisões são resolvidas eficientemente por técnicas como encadeamento ou endereçamento aberto. O custo nesse caso é ligeiramente maior do que no melhor caso, mas ainda é bastante eficiente.

4.5.3 PIOR CASO

No pior caso da busca em uma tabela hash, o tempo de execução pode degradar para linear. Isso ocorre quando muitas colisões acontecem e vários elementos são mapeados para o mesmo índice. Nessa situação, a busca pode se assemelhar à de percorrer uma lista encadeada ou uma sequência de verificações, tornando o custo superior ao do melhor e do caso médio.

5 CONCLUSÃO DA ANÁLISE

O gráfico abaixo exibe uma comparação dos tempos médios de execução dos algoritmos de busca citados no trabalho. A partir dele, podemos concluir que a busca em uma árvore binária apresentou o pior resultado. Isso se deve ao fato de que, em árvores binárias desbalanceadas, a busca pode degenerar para um tempo de execução linear, $\mathcal{O}(n)$, especialmente quando a árvore assume a forma de uma lista encadeada. Logo após, temos a busca em uma árvore AVL. Embora o algoritmo mantenha um tempo de execução logarítmico, $\mathcal{O}(\log n)$, mesmo no pior caso, devido ao balanceamento automático da árvore, o custo adicional de realizar rotações para manter o balanceamento pode torná-lo ligeiramente inferior ao desempenho da tabela hash em alguns cenários. A busca em uma tabela hash, por sua vez, ficou na primeira posição como a mais eficiente, apresentando um tempo de execução constante, $\mathcal{O}(1)$, no melhor e caso médio. No entanto, é importante observar que, no pior caso, quando ocorrem muitas colisões, o tempo de execução pode degradar para $\mathcal{O}(n)$. Mesmo assim, a tabela hash mostrou-se superior na maioria dos cenários, sendo a estrutura de dados mais eficiente para buscas rápidas, desde que seja bem projetada e dimensionada. Em resumo, enquanto a árvore binária pode sofrer degradação significativa em desempenho devido ao desbalanceamento, a árvore AVL melhora essa situação ao garantir balanceamento, mas ainda assim fica atrás da tabela hash, que se destacou como a solução mais eficiente, exceto em situações de colisões excessivas.

OBS: Vale ressaltar que cada um dos algoritmos foram executados 1000 vezes, tornando possível a criação de uma comparação gráfica confiável.

Comparação entre todos os algoritmos (casos médios)

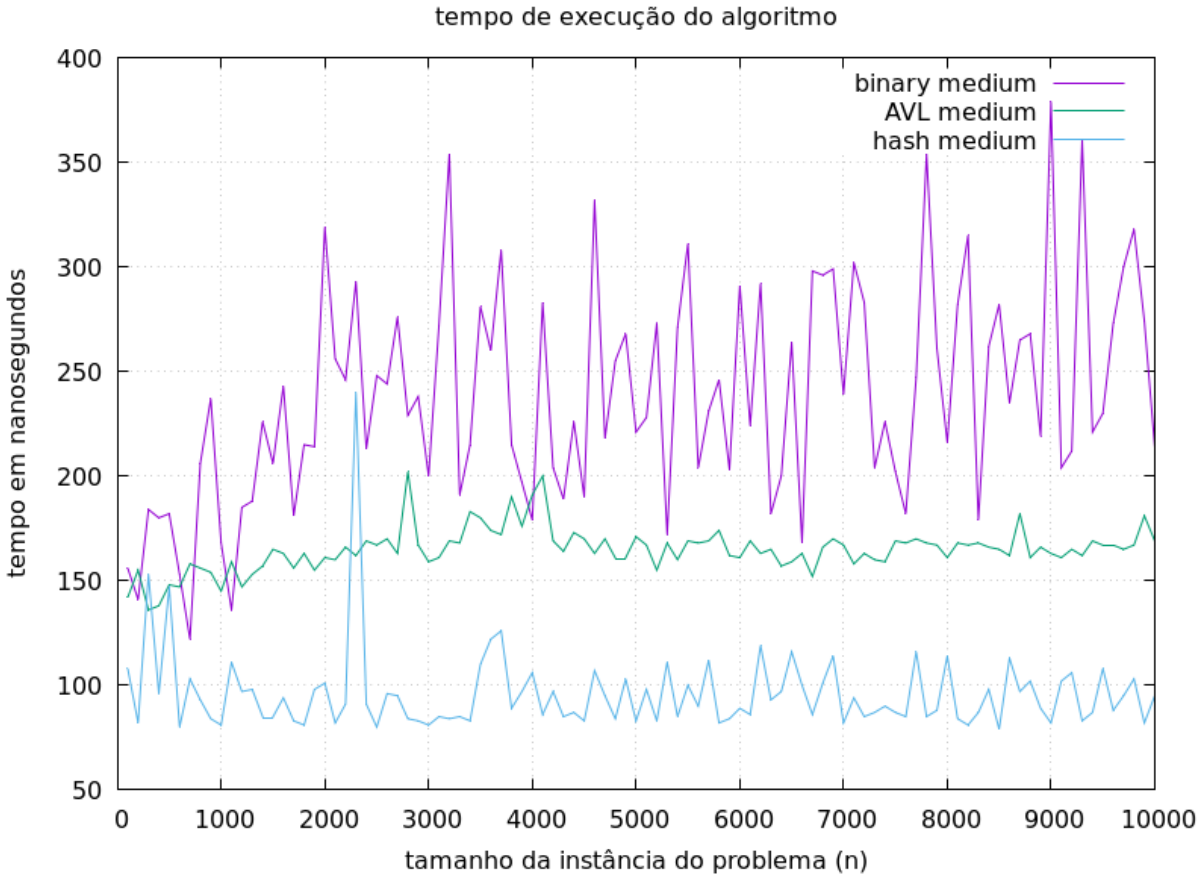

Figura 22 – Gráfico que compara o tempo de execução esperado de todos os algoritmos analisados

Classificação	Algoritmo	Tempo de Execução Esperado
1°	Hash Table	$\Theta(1)$
2°	AVL Tree	$\Theta(\log_2 n)$
3°	Binary Tree	$\Theta(\log_2 n)$