CADERNO DE CIÊNCIAS EXATAS E TECNOLOGIA

JOGO TORRES DE HANOI

TOWERS OF HANOI GAME

Aluno: Abel Isack da Silva Sá

Professora orientadora: Kadidja Valéria Reginaldo de Oliveira

Centro Universitário do Distrito Federal UDF, Coordenação de Tecnologia da Informação

RESUMO:

Introdução: Este trabalho apresenta o desenvolvimento de um jogo das Torres de Hanoi utilizando a linguagem C, com o objetivo de explorar conceitos fundamentais de lógica, algoritmos e programação estruturada. A implementação contempla uma interface interativa, permitindo ao usuário escolher níveis de dificuldade e realizar movimentos de acordo com as regras do jogo. O programa foi estruturado de forma modular, utilizando funções para imprimir as torres, gerenciar os movimentos e verificar condições de vitória. Durante o desenvolvimento, desafios como validação de entradas, controle lógico e tratamento de exceções foram abordados por meio de testes rigorosos e refinamento do código. O projeto destacou a importância de boas práticas de programação, organização do código e documentação detalhada, proporcionando uma experiência de aprendizado enriquecedora. A execução bem-sucedida do programa reforça sua aplicabilidade tanto no ensino de algoritmos quanto como base para a resolução de problemas mais complexos.

PALAVRAS-CHAVE: Torres de Hanoi, programação em C, algoritmos, alocação dinâmica, boas práticas de programação.

ABSTRACT:

Introduction: This study presents the development of a Tower of Hanoi game using the C programming language, aiming to explore fundamental concepts of logic, algorithms, and structured programming. The implementation includes an interactive interface, enabling users to select difficulty levels and perform moves according to the game's rules. The program was designed modularly, employing functions to print the towers, manage movements, and verify victory conditions. Challenges such as input validation, logical control, and exception handling were addressed through rigorous testing and code refinement. The project emphasized the importance of programming best practices, code organization, and thorough documentation, providing a comprehensive learning experience. The program's successful execution demonstrates its applicability both in algorithm teaching and as a foundation for solving more complex problems.

Tower of Hanoi, C programming, algorithms, dynamic memory allocation, programming best practices.

1. ANÁLISE

1.1. PROBLEMÁTICA

O problema abordado neste projeto é a implementação do clássico desafio das Torres de Hanoi em um ambiente interativo de programação, utilizando a linguagem C. Este problema é amplamente estudado em teoria da computação e programação, sendo conhecido por suas características recursivas e pelo desafio lógico que apresenta. A tarefa consiste em mover um conjunto de discos de tamanhos diferentes de uma torre inicial para outra, respeitando as seguintes restrições: apenas um disco pode ser movido por vez, e um disco maior nunca pode ser colocado sobre um disco menor. O objetivo principal é resolver o problema dentro de um número de movimentos máximo definido pelo jogador, com a opção de personalizar o nível de dificuldade. Assim, a definição do problema envolve tanto a fidelidade às regras clássicas quanto o desenvolvimento de uma interface de interação para os jogadores.

1.2. COMPREENSÃO E DESAFIOS DO PROBLEMA

O problema das Torres de Hanoi é um clássico da teoria computacional, conhecido por sua elegância matemática e pela lógica estruturada necessária para sua solução. Sua relevância acadêmica e prática reside no fato de ser um exemplo paradigmático de problemas recursivos e algoritmos de divisão e conquista. Neste projeto, a abordagem implementada não se baseia diretamente na recursividade clássica, mas utiliza conceitos fundamentais do problema, adaptados a uma interação iterativa com o jogador. Os desafios do problema podem ser divididos em três níveis principais:

1.2.1. Representação dos Estados das Torres

A escolha de vetores para representar os pinos (ou torres) reflete uma decisão prática e eficiente. Cada vetor armazena os discos como inteiros, ordenados do menor para o maior, simulando a ordem dos discos nos pinos. Essa estrutura permite manipulação direta e verificações rápidas das regras do jogo;

1.2.2. Validação das Regras

Uma parte fundamental da solução envolve garantir que os movimentos do jogador estejam em conformidade com as restrições do problema. Isso exige:

- Identificar corretamente o disco no topo da torre de origem;
- Determinar a posição correta para inserir o disco na torre de destino;
- Garantir que nenhum disco maior seja colocado sobre um menor, invalidando o movimento se necessário.

1.2.3. Gerenciamento das Interações com o Jogador

Uma interface interativa adiciona desafios únicos ao problema. É necessário lidar com entradas potencialmente inválidas e comunicar erros ao usuário de forma clara. Além disso, o programa deve fornecer um feedback constante sobre o progresso do jogador, como o estado atualizado das torres e o número de movimentos restantes. A implementação demonstra domínio desses aspectos, equilibrando fidelidade às regras clássicas do problema com a simplicidade de uma interface iterativa. Ao mesmo tempo, o código permite diferentes níveis de dificuldade e personalização, aumentando o valor educacional e recreativo da aplicação.

1.3. MODELO MATEMÁTICO

O modelo matemático subjacente ao problema das Torres de Hanoi é baseado na relação de recorrência $T(n) = 2T(n-1) + 1$, onde $T(n)$ representa o número mínimo de movimentos necessários para resolver o problema com n discos. Essa relação descreve uma solução recursiva: para mover uma pilha de n discos de uma torre para outra, é necessário primeiro mover os $n - 1$ discos superiores para uma torre intermediária, em seguida mover o maior disco diretamente para a torre de destino, e

finalmente mover os $n - 1$ discos da torre intermediária para a torre de destino. Embora essa implementação específica não utilize recursão, o modelo matemático guia toda a lógica do jogo:

- ✓ **Cálculo do Número Máximo de Movimentos:** A fórmula **(2^n)-1** é empregada para determinar o número máximo de jogadas, estabelecendo um limite e proporcionando ao jogador uma noção de eficiência.
- ✓ **Representação e Validação dos Estados:** Os vetores que representam as torres seguem o princípio de ordenação implícito do modelo matemático. Discos maiores não podem ser movidos para cima de discos menores, refletindo a restrição fundamental do problema.
- ✓ **Condição de Vitória:** A condição de vitória verifica se todos os discos estão corretamente organizados em uma torre de destino (B ou C). Essa verificação é realizada comparando os elementos do vetor da torre de destino com a ordem esperada dos discos, do menor para o maior.

Este modelo matemático é traduzido em lógica de programação por meio de operações iterativas e manipulação de vetores. A implementação também adiciona flexibilidade, permitindo ao jogador escolher o número de discos, o que altera a complexidade do problema em tempo de execução. Isso reforça o valor pedagógico do projeto, pois demonstra como conceitos abstratos de matemática e ciência da computação podem ser aplicados em um contexto prático e interativo.

2. PLANEJAMENTO

2.1. OBJETIVOS DO ALGORITIMO

O objetivo do algoritmo é desenvolver uma solução interativa para o problema das Torres de Hanoi, permitindo que o jogador mova discos entre as torres respeitando as regras clássicas do jogo. Além de garantir a funcionalidade básica do desafio, o algoritmo deve oferecer uma experiência intuitiva, incluindo:

- o Diferentes níveis de dificuldade, com base no número de discos;
- o Controle do número máximo de movimentos permitidos, com feedback constante ao jogador;
- o Validação de entradas e mensagens claras para movimentos inválidos.

2.2. MÉTRICAS ESTABELECIDAS

As métricas de avaliação consideram:

- ✓ Corretude: O algoritmo deve garantir que todos os movimentos respeitem as regras das Torres de Hanoi;
- ✓ Interatividade: O tempo de resposta para processar uma entrada do jogador deve ser curto, e o feedback deve ser imediato;
- ✓ Uso de memória: O uso de vetores dinâmicos para representar os pinos é avaliado quanto à eficiência na manipulação de dados;
- ✓ Facilidade de uso: A interface deve ser simples e funcional, reduzindo ao máximo a possibilidade de erros do jogador.

2.3. ESTRATÉGIA GERAL

A estratégia do algoritmo é baseada na modelagem do problema como um conjunto de torres representadas por vetores, onde:

- Cada vetor representa um pino e contém os discos ordenados.
- O jogador interage com o algoritmo fornecendo comandos para mover discos entre as torres.

- As funções do programa verificam a validade de cada movimento e atualizam o estado das torres.
- O jogo termina quando o jogador transfere todos os discos para uma torre de destino ou esgota o número de movimentos permitidos.

2.4. SUBPROBLEMAS IDENTIFICADOS

O problema foi decomposto nos seguintes subproblemas:

- Representação das torres: Escolher uma estrutura de dados que permita gerenciar os discos de maneira eficiente.
- Validação de movimentos: Implementar regras para verificar a validade dos movimentos solicitados pelo jogador.
- Controle do número de jogadas: Monitorar as jogadas restantes e informar o jogador.
- Condição de vitória: Determinar se o jogador completou o objetivo do jogo.
- Interatividade: Projetar uma interface que permita entrada de comandos e forneça feedback em tempo real.

2.5. ESTRUTURA GERAL DO ALGORITIMO

1) *Inicialização:*
    a) Solicitar o número de discos e calcular o número máximo de movimentos permitidos.
    b) Alocar dinamicamente os vetores representando os pinos.
    c) Configurar o pino inicial com os discos organizados.

2) *Laço principal:*
    a) Mostrar o estado atual das torres e o número de jogadas restantes.
    b) Solicitar ao jogador os pinos de origem e destino.
    c) Validar o movimento e atualizar as torres, se aplicável.
    d) Reduzir o número de jogadas restantes.
    e) Verificar a condição de vitória.

3) *Finalização:*
    a) Informar o jogador sobre o resultado (vitória ou derrota).
    b) Liberar a memória alocada dinamicamente.

2.6. CASOS LIMITE/SITUAÇÕES ESPECIAIS

- Entrada inválida (letras ou números fora do esperado) deve ser tratada para evitar comportamento inesperado.
- Tentativas de mover discos de uma torre vazia ou para uma torre onde o movimento viola as regras do jogo.
- Configuração personalizada com menos de 3 discos, o que tornaria o problema trivial ou inválido.

2.7. ANÁLISE TEÓRICA

A correção do algoritmo foi avaliada considerando:

Validação dos movimentos: O algoritmo impede movimentos inválidos, garantindo que um disco maior nunca seja colocado sobre um menor e que discos sejam movidos apenas de torres não vazias.

Progresso em direção ao objetivo: Cada movimento válido aproxima o jogador do estado final desejado, onde todos os discos estão na torre de destino.
Condição de vitória: A verificação final garante que a configuração das torres corresponda ao objetivo do jogo antes de declarar a vitória.
Terminação: O algoritmo termina corretamente ao atingir o número máximo de movimentos ou ao completar o desafio.

3. DESENHO

3.1. ANÁLISE DE COMPLEXIDADE E PONTOS CRÍTICOS DO ALGORITIMO

A análise de complexidade e identificação de pontos críticos de um algoritmo são etapas fundamentais para avaliar sua eficiência teórica e prática, além de identificar oportunidades de otimização. No caso do algoritmo para o jogo das Torres de Hanoi, essas análises envolvem tanto aspectos relacionados à eficiência computacional quanto à interação com o jogador, dada a natureza iterativa e interativa da solução.

Inicialmente, a complexidade temporal do algoritmo pode ser considerada a partir das operações fundamentais realizadas durante o jogo. A cada jogada, o programa realiza as seguintes tarefas: leitura das entradas do jogador, validação do movimento solicitado, atualização dos estados das torres, e exibição do estado atual. Para cada entrada, essas operações têm complexidade $O(n)$, onde n é o número de discos, devido à necessidade de percorrer os vetores que representam as torres para localizar os discos no topo e determinar as posições disponíveis. Assim, em um cenário onde m é o número máximo de movimentos permitido (m = 2^n − 1), a complexidade total do algoritmo pode ser aproximada como $O(m \cdot n)$, ou seja, $O(n \cdot 2\text{^}n)$.

A complexidade espacial é gerenciada eficientemente pela escolha de vetores para representar as torres. Cada vetor tem tamanho n, correspondendo ao número de discos, e há três vetores no total. Isso resulta em um uso de memória linear, $O(n)$, o que é adequado para o contexto de um jogo interativo. Além disso, a memória é alocada dinamicamente no início do programa e liberada ao final, minimizando desperdícios e possibilitando a escalabilidade para diferentes tamanhos de jogo.

Entretanto, alguns pontos críticos do algoritmo merecem atenção. O primeiro está relacionado à validação de movimentos. Embora a lógica atual assegure que nenhum disco maior seja colocado sobre um menor, essa operação envolve múltiplas verificações sequenciais no vetor, o que pode se tornar um gargalo conforme o número de discos aumenta. Para otimizar essa etapa, seria possível manter índices adicionais que apontam diretamente para os topos de cada torre, eliminando a necessidade de percorrer os vetores em busca do disco mais alto. Essa abordagem reduziria a complexidade das operações de validação para $O(1)$ no acesso direto, melhorando significativamente o desempenho para jogos maiores.

Outro ponto crítico é a interação com o jogador. O algoritmo depende de entradas válidas para funcionar corretamente, e a validação dessas entradas é realizada a cada jogada. Atualmente, o sistema realiza verificações básicas para garantir que as letras fornecidas (A, B, C) sejam válidas e converte caracteres para maiúsculas. No entanto, entradas inválidas ou repetidas podem prolongar artificialmente o tempo de execução, especialmente em um ambiente de jogo interativo. Um refinamento adicional poderia incluir mensagens de erro mais detalhadas e sugestões ao jogador, reduzindo o número de entradas malformadas.

Por fim, a análise teórica da eficiência do algoritmo destaca um limite intrínseco ao problema das Torres de Hanoi: o crescimento exponencial do número de movimentos mínimos necessários (2^n − 1) conforme o número de discos aumenta. Isso implica que, independentemente de otimizações no

código, o problema em si se torna computacionalmente mais desafiador à medida que o tamanho cresce. Portanto, o design do algoritmo deve equilibrar desempenho com a usabilidade em tamanhos razoáveis, como 3 a 6 discos, onde a experiência interativa permanece gerenciável tanto para o jogador quanto para o sistema.

Em síntese, enquanto a implementação atual é funcional e eficiente para o propósito proposto, melhorias na validação de movimentos e no gerenciamento de entradas poderiam otimizar ainda mais o desempenho. A análise de complexidade destaca que o uso de memória é linear, mas o tempo de execução cresce exponencialmente com o número de discos, refletindo as propriedades matemáticas do problema original. Essa compreensão é essencial para justificar decisões de design e propor aprimoramentos no futuro.

4. PROGRAMAÇÃO E TESTE

4.1. CÓDIGO FONTE DO JOGO

```
#include <stdio.h> // Biblioteca de funções principais (printf, scanf, puts, etc)
#include <stdlib.h> // Biblioteca para manipulação de memória
#include <locale.h> // Biblioteca para usar acentuação
#include <math.h> // Biblioteca para cálculo matemático
#include <ctype.h> // Biblioteca para usar a função topper

// Declarações de funções
void imprimirTorre(const char *nome, int tamanho, int *vetor);
void mudarDeTorre(int *vetorOrigem, int *vetorDestino, int tamanho);
void jogo(int tamanho, int jogadasMax);

int main() {
    setlocale(LC_ALL, "Portuguese");

    int option, tamanho, jogadasMax;

    printf("Defina o nível da sua Torre de Hanoi:\n");
    printf("1. Fácil - 3 pinos\n");
    printf("2. Médio - 4 pinos\n");
    printf("3. Difícil - 5 pinos\n");
    printf("4. Personalizado - escolha o número de pinos\n");
    scanf(" %d", &option);

    // Configura o tamanho da torre e as jogadas máximas
    switch (option) {
        case 1:
            tamanho = 3;
            break;
        case 2:
            tamanho = 4;
            break;
        case 3:
            tamanho = 5;
```

```
            break;
        case 4:
            printf("Digite o número de pinos desejado: ");
            scanf("%d", &tamanho);
            if (tamanho < 3) {
                printf("Número de pinos deve ser pelo menos 3!\n");
                return 1;
            }
            break;
        default:
            printf("Opção inválida!\n");
            return 1;
    }

    // Calcula o número máximo de jogadas permitidas: (2^n) - 1
    jogadasMax = (int)pow(2, tamanho) - 1;
    jogo(tamanho, jogadasMax);

    return 0;
}

// Função para imprimir o estado de uma torre
void imprimirTorre(const char *nome, int tamanho, int *vetor) {
    printf("Torre %s:\n", nome);
    for (int i = 0; i < tamanho; i++) {
        if (vetor[i] != 0)
            printf("  %d\n", vetor[i]);
        else
            printf("  -\n");
    }
}

// Função para mover um disco de uma torre para outra
void mudarDeTorre(int *vetorOrigem, int *vetorDestino, int tamanho) {
    int topoOrigem = -1, topoDestino = -1;

    // Encontrar o disco no topo da torre de origem
    for (int i = 0; i < tamanho; i++) {
        if (vetorOrigem[i] != 0) {
            topoOrigem = i;
            break;
        }
    }

    // Encontrar a posição disponível na base da torre de destino
    for (int i = tamanho - 1; i >= 0; i--) {
```

```
        if (vetorDestino[i] == 0) {
            topoDestino = i;
            break;
        }
    }

    if (topoOrigem == -1) {
        printf("Movimento inválido: Torre de origem está vazia\n");
        return;
    }

    if (topoDestino == tamanho - 1 || vetorOrigem[topoOrigem] < vetorDestino[topoDestino + 1]) {
        // Realizar o movimento
        vetorDestino[topoDestino] = vetorOrigem[topoOrigem];
        vetorOrigem[topoOrigem] = 0;
        printf("Movimento realizado: %d movido\n", vetorDestino[topoDestino]);
    } else {
        printf("Movimento inválido: Disco maior sobre disco menor\n");
    }
}

// Função principal do jogo
void jogo(int tamanho, int jogadasMax) {
    // Alocação dinâmica dos vetores que representam as torres
    int *pinoA = (int *)malloc(tamanho * sizeof(int));
    int *pinoB = (int *)calloc(tamanho, sizeof(int));
    int *pinoC = (int *)calloc(tamanho, sizeof(int));

    // Inicializa a torre A com os discos e as demais torres vazias
    for (int i = 0; i < tamanho; i++) {
        pinoA[i] = i + 1;

    }

    char partida, destino;
    printf("Desafio das Torres de Hanoi\n");

    // Imprime o estado inicial das torres
    imprimirTorre("A", tamanho, pinoA);
    imprimirTorre("B", tamanho, pinoB);
    imprimirTorre("C", tamanho, pinoC);

    while (jogadasMax > 0) {
        printf("Jogadas restantes: %d\n", jogadasMax);
        printf("Digite o pino de partida (A/B/C): ");
        scanf(" %c", &partida);
```

```
printf("Digite o pino de destino (A/B/C): ");
scanf(" %c", &destino);

// Normaliza a entrada para letras maiúsculas
partida = toupper(partida);
destino = toupper(destino);

// Realiza o movimento de acordo com a entrada do usuário
if (partida == 'A' && destino == 'B') {
    mudarDeTorre(pinoA, pinoB, tamanho);
} else if (partida == 'A' && destino == 'C') {
    mudarDeTorre(pinoA, pinoC, tamanho);
} else if (partida == 'B' && destino == 'A') {
    mudarDeTorre(pinoB, pinoA, tamanho);
} else if (partida == 'B' && destino == 'C') {
    mudarDeTorre(pinoB, pinoC, tamanho);
} else if (partida == 'C' && destino == 'A') {
    mudarDeTorre(pinoC, pinoA, tamanho);
} else if (partida == 'C' && destino == 'B') {
    mudarDeTorre(pinoC, pinoB, tamanho);
} else {
    printf("Movimento inválido!\n");
    continue;
}

// Imprime o estado atualizado das torres
printf("\nEstado das Torres:\n");
imprimirTorre("A", tamanho, pinoA);
imprimirTorre("B", tamanho, pinoB);
imprimirTorre("C", tamanho, pinoC);

jogadasMax--;

// Verifica a condição de vitória: se todos os discos estão na torre B ou C
int vitoria = 1;
for (int i = 0; i < tamanho; i++) {
    if (pinoB[i] != i + 1 && pinoC[i] != i + 1) {
        vitoria = 0;
        break;
    }
}

if (vitoria) {
    printf("Parabéns! Você resolveu o desafio das Torres de Hanoi!\n");
    break;
}
```

```
        if (jogadasMax == 0) {
            printf("Você atingiu o número máximo de jogadas!\n");
        }
    }

    // Libera a memória alocada dinamicamente
    free(pinoA);
    free(pinoB);
    free(pinoC);
}
```

4.2. ALGORITMO TRADUZIDO COM PRECISÃO PARA LINGUAGEM C

O algoritmo para o problema das Torres de Hanoi foi fielmente traduzido para código C, utilizando funções bem definidas e estruturas de controle adequadas. A estrutura geral do código reflete o modelo lógico do jogo, com as funções **mudarDeTorre**, **imprimirTorre** e *jogo* organizadas de forma modular.

O código começa com a configuração do nível do jogo através da variável **option**, onde o jogador escolhe o número de discos (3, 4 ou 5) ou insere um valor personalizado. A seguir, o número máximo de jogadas (**jogadasMax**) é calculado usando a fórmula $\mathbf{2\text{^}n - 1}$, conforme o enunciado do problema.

A função principal (**main**) trata da inicialização e da configuração do jogo, enquanto as funções auxiliares **mudarDeTorre** e **imprimirTorre** são responsáveis por gerenciar as operações específicas de movimentação de discos e exibição do estado das torres.

4.3. CLEAN CODE

O código foi escrito com clareza e organização, seguindo boas práticas de programação. A utilização de bibliotecas como <stdio.h>, <stdlib.h>, <locale.h>, <math.h>, e <ctype.h> é apropriada para garantir a funcionalidade do programa, como a manipulação de entradas e saídas, alocação de memória, e controle de acentuação e letras maiúsculas.

O código foi estruturado em funções independentes para cada tarefa: imprimirTorre para mostrar o estado de uma torre, mudarDeTorre para mover discos entre as torres e jogo para controlar o fluxo principal do jogo. A indentação e o uso de nomes claros para as variáveis, como pinoA, pinoB e pinoC (que representam as torres), ajudam na legibilidade e compreensão do código. Além disso, o código faz uso de comentários que facilitam o entendimento do que cada parte do código realiza.

4.4. TESTES

Foram realizados testes rigorosos para garantir que o jogo funcione corretamente em diferentes cenários. Testes com os níveis de dificuldade "Fácil" (3 discos), "Médio" (4 discos) e "Difícil" (5 discos) foram realizados para verificar se o código lida adequadamente com diferentes configurações de jogo.

O código foi validado tanto com jogadas válidas quanto inválidas, e o comportamento esperado foi verificado para todas as condições do jogo, incluindo a verificação de vitória e a condição de fim de jogo após o número máximo de jogadas. O código também foi testado com entradas personalizadas, onde o número de discos foi inserido manualmente, garantindo que o sistema lida corretamente com configurações não predefinidas.

4.4. ERROS E PROBLEMAS IDENTIFICADOS DURANTE OS TESTES

Durante os testes, alguns erros e problemas foram identificados e corrigidos. Inicialmente, o código apresentava problemas ao tentar mover um disco para uma torre quando não havia espaço suficiente, o que gerava um comportamento inesperado. Esse erro foi corrigido no código da função **mudarDeTorre**, onde foi adicionada uma verificação para garantir que o disco só fosse movido para a posição correta da torre de destino, respeitando a ordem dos discos.

Outro problema identificado foi o gerenciamento incorreto do número de jogadas restantes, que foi ajustado para garantir que a condição de fim de jogo fosse verificada corretamente. Além disso, ao lidar com a entrada de dados, a função **scanf** foi ajustada para garantir que as respostas do jogador fossem lidas corretamente, com especial atenção para a normalização das entradas de pinos, utilizando a função **toupper** para garantir que as letras fossem tratadas como maiúsculas. Todos esses ajustes foram testados novamente para garantir a correção do código.

5. CONCLUSÃO

O projeto de desenvolvimento do jogo das Torres de Hanoi em linguagem C proporcionou resultados significativos tanto em termos técnicos quanto acadêmicos. A implementação bem-sucedida de um algoritmo funcional, capaz de representar de maneira precisa as regras e a dinâmica do jogo, destaca a aplicação prática dos conceitos de programação estruturada, alocação dinâmica de memória e modularidade. Além disso, a interface com o usuário, ainda que simples, foi suficiente para permitir uma experiência interativa e intuitiva, garantindo a execução do jogo em diferentes níveis de dificuldade. Esses resultados demonstram a eficácia da abordagem escolhida para o desenvolvimento do programa. Durante o processo de desenvolvimento, foram enfrentados desafios relacionados ao controle lógico das operações e à validação das entradas do usuário, especialmente ao lidar com situações inesperadas ou erros. Esses obstáculos destacaram a importância de realizar testes abrangentes e de implementar verificações robustas para garantir a funcionalidade do programa em casos limite.

Ademais, o processo de depuração permitiu um aprendizado valioso sobre a importância de prever e tratar exceções no código, evitando comportamentos inesperados e garantindo a estabilidade do software. A experiência reforçou a relevância das boas práticas de programação, como a organização do código em funções reutilizáveis e a clareza na nomeação das variáveis.

Por fim, este projeto proporcionou um aprendizado abrangente sobre a tradução de problemas abstratos para soluções computacionais práticas. A Torre de Hanoi, embora um problema clássico, serviu como uma excelente base para explorar conceitos fundamentais de lógica, recursão e manipulação de dados. A execução do projeto evidenciou a relevância de documentar cada etapa do processo de desenvolvimento, tanto para facilitar a comunicação quanto para organizar o trabalho. Esse aprendizado será fundamental para enfrentar projetos futuros, onde habilidades similares poderão ser aplicadas para resolver problemas mais complexos e desafiadores.

6. REFERÊNCIAS

KERNIGHAN, Brian W.; RITCHIE, Dennis M. *The C Programming Language*. 2nd ed. Englewood Cliffs: Prentice Hall, 1988.

TANENBAUM, Andrew S.; BOS, Herbert. *Modern Operating Systems*. 4th ed. Pearson, 2014.

PRESSMAN, Roger S. *Engenharia de Software: Uma Abordagem Profissional*. 7ª ed. São Paulo: McGraw-Hill, 2011.

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Introduction to Algorithms*. 3rd ed. MIT Press, 2009.

SEDGEWICK, Robert; WAYNE, Kevin. *Algorithms*. 4th ed. Addison-Wesley, 2011.

DEITEL, Paul; DEITEL, Harvey. *C: Como Programar*. 8ª ed. São Paulo: Pearson, 2016.

KNUTH, Donald E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. 3rd ed. Addison-Wesley, 1997.

ZIVIANI, Nivio. *Projeto de Algoritmos: Com Implementações em Pascal e C*. 3ª ed. São Paulo: Thomson Learning, 2011.