Universidade Federal de Ouro Preto

Instituto de Ciências Exatas e Aplicadas

Departamento de Computação e Sistemas

ALGORITMOS E ESTRUTURAS DE DADOS
Primeiro Trabalho

Rafael Pereira da Silveira - 24.2.8062

Prof.: Bruno Hott

João Monlevade
2 de junho de 2025

Sumário

1 Introdução

Este trabalho tem como objetivo desenvolver um programa que resolve o jogo de "caça-palavras", bastante conhecido em revistas de passatempo e até usado em atividades educativas. A ideia é criar um sistema que receba uma matriz de letras e uma lista de palavras, e que consiga encontrar automaticamente onde cada palavra está dentro da matriz, informando a posição de início e fim de cada uma. Essas palavras podem estar dispostas em várias direções (horizontal, vertical e diagonal), o que torna o desafio mais interessante.

Para organizar melhor o código e facilitar a manutenção, o projeto foi construído usando Tipos Abstratos de Dados (TADs). Com isso, cada parte do programa ficou responsável por uma função específica. Foram criados TADs para lidar com a matriz principal, as palavras que precisam ser encontradas, as coordenadas das letras e também uma tabela que armazena as ocorrências das palavras encontradas.

Cada TAD foi implementado em seu próprio arquivo, deixando o código mais limpo e modular. O arquivo resolve.c funciona como um elo entre esses módulos: ele usa as funções dos TADs para montar a solução final do problema, integrando tudo de forma organizada.

Durante o desenvolvimento, também foram aplicados conceitos importantes da linguagem C, como o uso de ponteiros, manipulação de strings e estruturas, testes de consistência e alocação dinâmica de memória — essencial para permitir que o programa funcione com matrizes e listas de palavras de tamanhos variados.

Este relatório apresenta a lógica usada no projeto, os principais obstáculos enfrentados e como eles foram resolvidos. Além de propor uma solução funcional para o jogo, o trabalho busca reforçar a importância de um código bem organizado, modular e construído com boas práticas de programação.

2 Implementação

2.1 Organização do Código, Decisões de Implementação e Detalhes Técnicos

O código está organizado em onze arquivos principais: `Tmatriz.c`, `Tmatriz.h`, `resolve.c`, `resolve.h`, `Tcoordenada.c`, `Tcoordenada.h`, `Tocorrencias.c`, `Tocorrencias.h`, `Tpalavra.c` e `Tpalavra.h`, que implementam os Tipos Abstratos de Dados (TADs). O arquivo `main.c` contém o programa principal, responsável por controlar o fluxo geral da aplicação.

O arquivo `resolve.c` funciona como a camada central do programa, integrando as funcionalidades dos diferentes TADs. Ele é responsável por criar a estrutura principal do programa, carregar a matriz de letras e a lista de palavras, realizar a busca dessas palavras na matriz em todas as direções possíveis, imprimir os resultados encontrados e liberar a memória utilizada. Dessa forma, `resolve.c` orquestra a interação entre os módulos para garantir o funcionamento correto e eficiente do caça-palavras.

Para compilar o programa, foi utilizado o compilador GNU Compiler Collection (GCC) no sistema operacional Windows 10, com o Visual Studio Code versão 1.100 como ambiente de desenvolvimento integrado (IDE). A compilação pode ser realizada pela linha de comando com o seguinte comando:

```
gcc -o Main resolve.c Tmatriz.c Tpalavra.c Tocorrencias.c Tcoordenada.c
```

Em seguida, o programa pode ser executado com o comando:

```
./Main
```

Também é possível compilar e executar diretamente pelo ambiente do Visual Studio Code.

2.2 Estrutura de Dados

No desenvolvimento do jogo de caça-palavras, foram implementados quatro Tipos Abstratos de Dados (TADs) principais: Matriz, Coordenada, Ocorrências e Palavra. Cada um desses TADs desempenha um papel fundamental na organização e manipulação dos dados do jogo.

2.2.1 TAD Matriz

O TAD Matriz é responsável por representar a grade de letras onde as palavras serão buscadas. Ele encapsula uma matriz bidimensional de caracteres e oferece operações para criar, preencher, acessar e liberar a memória utilizada pela matriz. This abstract allows o código do jogo to interaja with a matriz of a efficient form and to express the implementaço.

Programa 1: TAD Tmatriz

```c
#ifndef TMATRIZ_H
#define TMATRIZ_H

// Estrutura que representa a matriz do caça-palavras
typedef struct Tmatriz {
    char **letras;
    int linhas;
    int colunas;
} Tmatriz;

// Cria e inicializa a matriz com as dimensões fornecidas
Tmatriz *criar_matriz(int linhas, int colunas);

// Preenche a matriz com caracteres fornecidos pelo usuário
int preencher_matriz(Tmatriz *matriz);

// Imprime a matriz na tela
void imprimir_matriz(Tmatriz *matriz);

// Libera a memória alocada pela matriz
int apagar_matriz(Tmatriz *matriz);

#endif
```

2.2.2 TAD Coordenada

O TAD Coordenada representa uma posição na matriz, armazenando os índices de linha e coluna. Ele fornece operações para criar uma coordenada com valores específicos, acessar os valores de linha e coluna e comparar se duas coordenadas são iguais. Esse TAD é essencial para registrar as posições de início e fim das palavras encontradas na matriz.

Programa 2: TAD Tcoordenada

```c
#ifndef TCOORDENADA_H
#define TCOORDENADA_H

#include "Tmatriz.h"

// Representa uma posição na matriz
typedef struct
{
    int linha;
    int coluna;
} Tcoordenada;


//Cria uma coordenada com linha e coluna fornecidas.

Tcoordenada criar_coordenada(int linha, int coluna);


//Verifica se a coordenada está dentro dos limites da matriz.

int verificar_coordenada(Tcoordenada coordenada, Tmatriz *matriz);

#endif
```

2.2.3 TAD Ocorrências

O TAD Ocorrências gerencia a lista de palavras encontradas no jogo. Utilizando uma lista ligada, ele permite operações como inserir uma nova ocorrência, exibir as palavras encontradas e liberar a memória utilizada pela lista. Esse TAD abstrai a complexidade da manipulação direta da lista, oferecendo uma interface simples para gerenciar as ocorrências durante o jogo.

Programa 3: TAD Tocorrencias

```
1 #ifndef TOCORRENCIAS_H
2 #define TOCORRENCIAS_H
3
4 #include "Tcoordenada.h"
5
6 // Representa uma ocorrência de uma palavra no caça-palavras
7 typedef struct
8 {
9     Tcoordenada inicio;
10    Tcoordenada fim;
11 } Tocorrencia;
12
13 // Estrutura que agrupa todas as ocorrências encontradas
14 typedef struct
15 {
16    Tocorrencia *ocorrencias; // Vetor de ocorrências
17    int qtd;                  // Quantidade de palavras/ocorrências
18 } Tocorrencias;
19
20 // Cria estrutura para armazenar ocorrências
21 Tocorrencias *criar_tocorrencias(int qtd_palavras);
22
23 // Define a ocorrência de uma palavra (posição inicial e final)
24 void adicionar_ocorrencia(Tocorrencias *toc, int indice, Tcoordenada ini,
       Tcoordenada fim);
25
26 // Libera memória alocada para as ocorrências
27 void apagar_tocorrencias(Tocorrencias *toc);
28
29 // Mostra uma ocorrência (útil para depuração)
30 void imprimir_ocorrencia(Tocorrencia ocorrencia);
31
32 #endif
```

2.2.4 TAD Palavra

O TAD Palavra representa uma palavra a ser buscada na matriz. Ele armazena a palavra e fornece operações para criar uma palavra com um valor específico, acessar o valor da palavra e comparar se a palavra corresponde a outra. Esse TAD é essencial para armazenar e manipular as palavras que o jogo deve encontrar na matriz.**??**.

Programa 4: TAD Tpalavra

```
1 #ifndef TPALAVRA_H
2 #define TPALAVRA_H
3
4 #include "Tcoordenada.h"
5
6 // Estrutura que representa uma palavra no caça-palavras
7 typedef struct
8 {
9     char *palavra;          // Texto da palavra
```

```
10      int achada;             // bool indicando se foi encontrada (1) ou nã
            o (0)
11      Tcoordenada inicio;     // Coordenada de início na matriz
12      Tcoordenada fim;        // Coordenada de fim na matriz
13  } Tpalavra;
14
15  // Cria uma nova palavra a partir de uma string
16  Tpalavra criar_palavra(char *palavra);
17
18  // Define a posição (início e fim) de uma palavra encontrada
19  void posicao_palavra(Tpalavra *palavra, Tcoordenada inicio, Tcoordenada
        fim);
20
21  // Marca a palavra como não encontrada (e reseta coordenadas)
22  void palavra_n_encontrada(Tpalavra *palavra);
23
24  // Libera a memória alocada da palavra
25  int apagar_palavra(Tpalavra *palavra);
26
27  // Adiciona uma nova palavra ao vetor (função auxiliar simples)
28  int palavras_add(Tpalavra *vetor, Tpalavra nova);
29
30  // Lê as palavras do usuário e preenche o vetor
31  int palavras_preencher(Tpalavra *vetor, int num_palavras);
32
33  // Mostra o resultado final das palavras (encontradas ou não)
34  void imprimir_solucao(Tpalavra *vetor, int num_palavras);
35
36  #endif
```

2.2.5 TAD Resolve

O arquivo resolve.c atua como a camada central do programa, integrando as funcionalidades dos diferentes TADs. Ele é responsável por criar a estrutura principal do programa, carregar a matriz de letras e a lista de palavras, realizar a busca das palavras na matriz em todas as direções possíveis, imprimir os resultados encontrados e liberar a memória utilizada. Dessa forma, resolve.c orquestra a interação entre os módulos para garantir o funcionamento correto e eficiente do caça-palavras.

Programa 5: TAD resolve

```
1  #ifndef RESOLVE_H
2  #define RESOLVE_H
3
4  #include "Tmatriz.h"
5  #include "Tpalavra.h"
6  #include "Tocorrencias.h"
7
8  // Estrutura principal que representa o resolvedor do caça-palavras
9  typedef struct
10 {
11      Tmatriz *matriz;        // Ponteiro para a matriz de letras
12      Tpalavra *palavras;     // Vetor de palavras a serem buscadas
13      int qtd_palavras;       // Quantidade de palavras
14 } Tresolve;
15
16  //Cria o resolvedor com base nas dimensões da matriz e quantidade de
        palavras.
17  //Aloca memória para matriz e vetor de palavras.
18
```

```
19 Tresolve *criar_resolve(int linhas, int colunas, int qtd_palavras);
20
21
22  //Lê a matriz e as palavras do usuário usando os TADs específicos.
23
24 void ler_resolve(Tresolve *resolve);
25
26
27  //Busca todas as palavras na matriz e retorna as ocorrências encontradas
        .
28  //Usa os TADs Tcoordenada e Tocorrencias para registrar os resultados.
29
30 Tocorrencias *buscar_palavras_resolve(Tresolve *resolve);
31
32
33 //Imprime as posições de início e fim de cada palavra encontrada na
       matriz.
34
35 void imprimir_resultado_resolve(Tocorrencias *ocorrencias, Tpalavra *
       palavras, int qtd_palavras);
36
37
38 //Libera toda a memória alocada na estrutura Tresolve.
39
40 void apagar_resolve(Tresolve *resolve);
41
42 #endif
```

O campo `int* elementos` é basicamente um `int[MAX]` que armazena os elementos do vetor enquanto o campo `n` guarda o número de elementos do vetor. Como estamos utilizando alocação estática de memória, foi criada uma constante `MAX` com o tamanho máximo do vetor.

2.3 Funções e Procedimentos

Os TAD's criados possui as seguintes funções:

2.3.1 Tmatriz

2.3.2 criar_matriz

A função criar_matriz é responsável por alocar dinamicamente a memória necessária para armazenar a matriz de letras. Ela recebe como parâmetros o número de linhas e colunas desejadas para a matriz. Primeiramente, aloca-se memória para a estrutura Tmatriz. Em seguida, aloca-se memória para as linhas da matriz e, por fim, para as colunas de cada linha. Caso ocorra algum erro durante a alocação, a função exibe uma mensagem de erro e encerra o programa.

```
1      Tmatriz *criar_matriz(int linhas, int colunas)
2 {
3      Tmatriz *m = (Tmatriz *)malloc(sizeof(Tmatriz));
4      if (m == NULL)
5      {
6          fprintf(stderr, "Erro ao alocar matriz.\n");
7          exit(1);
8      }
9
10     m->linhas = linhas;
11     m->colunas = colunas;
12
13     // Aloca linhas
14     m->letras = (char **)malloc(linhas * sizeof(char *));
```

```
15      if (m->letras == NULL)
16      {
17          free(m);
18          fprintf(stderr, "Erro ao alocar linhas da matriz.\n");
19          exit(1);
20      }
```

2.3.3 preencher_matriz

A função preencher_matriz permite que o usuário insira os caracteres que irão compor a matriz de letras. Ela percorre cada posição da matriz e solicita ao usuário que informe a letra correspondente. Essa função é fundamental para inicializar a matriz com os dados fornecidos pelo usuário, permitindo a personalização do jogo.

```
1       // Lê os caracteres da matriz fornecidos pelo usuário
2   int preencher_matriz(Tmatriz *matriz)
3   {
4       printf("Preencha a matriz do caca-palavras (%dx%d):\n", matriz->
            linhas, matriz->colunas);
5
6       for (int i = 0; i < matriz->linhas; i++)
7       {
8           for (int j = 0; j < matriz->colunas; j++)
9           {
10              printf("Digite a letra da posicao [%d][%d]: ", i + 1, j + 1);
11              scanf(" %c", &matriz->letras[i][j]);
12          }
13      }
14
15      return 0;
16  }
```

2.3.4 imprimir_matriz

A função imprimir_matriz exibe a matriz de letras no formato de uma tabela, facilitando a visualização das posições das palavras no jogo. Ela percorre cada linha e coluna da matriz, imprimindo os caracteres armazenados em cada posição. Essa função é útil para apresentar ao usuário a configuração atual da matriz.

```
1       // Exibe a matriz formatada
2   void imprimir_matriz(Tmatriz *matriz)
3   {
4       printf("\nMatriz do caça-palavras:\n");
5
6       for (int i = 0; i < matriz->linhas; i++)
7       {
8           for (int j = 0; j < matriz->colunas; j++)
9           {
10              printf("%c ", matriz->letras[i][j]);
11          }
12          printf("\n");
13      }
14  }
```

2.3.5 apagar_matriz

A função apagar_matriz é responsável por liberar toda a memória alocada para a matriz. Ela percorre cada linha da matriz, liberando a memória de cada coluna, e, por fim, libera a memória da estrutura

Tmatriz em si. Essa função é importante para evitar vazamentos de memória e garantir que os recursos sejam liberados adequadamente ao final do uso da matriz.

```
1    // Libera toda a memória associada à matriz
2 int apagar_matriz(Tmatriz *matriz)
3 {
4     if (matriz == NULL)
5         return -1;
6
7     for (int i = 0; i < matriz->linhas; i++)
8     {
9         free(matriz->letras[i]);
10    }
11
12    free(matriz->letras);
13    free(matriz);
14
15    return 0;
16 }
```

2.3.6 criar_matriz

A função criar_matriz é responsável por alocar dinamicamente a memória necessária para armazenar a matriz de letras. Ela recebe como parâmetros o número de linhas e colunas desejadas para a matriz. Primeiramente, aloca-se memória para a estrutura Tmatriz. Em seguida, aloca-se memória para as linhas da matriz e, por fim, para as colunas de cada linha. Caso ocorra algum erro durante a alocação, a função exibe uma mensagem de erro e encerra o programa.

```
1     Tmatriz *criar_matriz(int linhas, int colunas)
2 {
3     Tmatriz *m = (Tmatriz *)malloc(sizeof(Tmatriz));
4     if (m == NULL)
5     {
6         fprintf(stderr, "Erro ao alocar matriz.\n");
7         exit(1);
8     }
9
10    m->linhas = linhas;
11    m->colunas = colunas;
12
13    // Aloca linhas
14    m->letras = (char **)malloc(linhas * sizeof(char *));
15    if (m->letras == NULL)
16    {
17        free(m);
18        fprintf(stderr, "Erro ao alocar linhas da matriz.\n");
19        exit(1);
20    }
```

2.3.7 criar_matriz

A função criar_matriz é responsável por alocar dinamicamente a memória necessária para armazenar a matriz de letras. Ela recebe como parâmetros o número de linhas e colunas desejadas para a matriz. Primeiramente, aloca-se memória para a estrutura Tmatriz. Em seguida, aloca-se memória para as linhas da matriz e, por fim, para as colunas de cada linha. Caso ocorra algum erro durante a alocação, a função exibe uma mensagem de erro e encerra o programa.

```
1     Tmatriz *criar_matriz(int linhas, int colunas)
2 {
3     Tmatriz *m = (Tmatriz *)malloc(sizeof(Tmatriz));
```

```
4       if (m == NULL)
5       {
6           fprintf(stderr, "Erro ao alocar matriz.\n");
7           exit(1);
8       }
9
10      m->linhas = linhas;
11      m->colunas = colunas;
12
13      // Aloca linhas
14      m->letras = (char **)malloc(linhas * sizeof(char *));
15      if (m->letras == NULL)
16      {
17          free(m);
18          fprintf(stderr, "Erro ao alocar linhas da matriz.\n");
19          exit(1);
20      }
```

2.3.8 Tcoordenada

2.3.9 criar_coordenada

A função criar_coordenada é responsável por criar e retornar uma estrutura Tcoordenada com os valores de linha e coluna fornecidos como parâmetros. Essa função é útil para encapsular a criação de coordenadas, garantindo que todas as coordenadas utilizadas no programa sejam instanciadas de forma consistente.

```
1       //Cria e retorna uma coordenada com linha e coluna fornecidas.
2   Tcoordenada criar_coordenada(int linha, int coluna)
3   {
4       Tcoordenada coord;
5       coord.linha = linha;
6       coord.coluna = coluna;
7       return coord;
8   }
```

2.3.10 verificar_coordenada

A função verificar_coordenada verifica se uma determinada coordenada está dentro dos limites válidos da matriz. Ela recebe uma coordenada e um ponteiro para a matriz como parâmetros e retorna 1 se a coordenada estiver dentro dos limites da matriz ou 0 caso contrário. Essa verificação é essencial para evitar acessos inválidos à matriz, que poderiam causar erros de execução.

```
1   //Verifica se a coordenada está dentro dos limites da matriz.
2   //Retorna 1 se for válida, 0 caso contrário.
3
4   int verificar_coordenada(Tcoordenada coordenada, Tmatriz *matriz)
5   {
6       return coordenada.linha >= 0 && coordenada.linha < matriz->linhas &&
7              coordenada.coluna >= 0 && coordenada.coluna < matriz->colunas;
8   }turn coord;
9   }
```

2.3.11 Tocorrencias

2.3.12 criar_tocorrencias

A função `criar_tocorrencias` é responsável por alocar dinamicamente a estrutura `Tocorrencias`, que armazena o início e o fim das ocorrências das palavras encontradas na matriz. Ela recebe como parâmetro

a quantidade de palavras a serem buscadas e aloca um vetor de estruturas `Tocorrencia` correspondente. Essa estrutura centraliza o armazenamento das posições das palavras localizadas no caça-palavras.

```
1 // Cria estrutura de ocorrências com espaço para todas as palavras
2 Tocorrencias *criar_tocorrencias(int qtd_palavras)
3 {
4     Tocorrencias *toc = (Tocorrencias *)malloc(sizeof(Tocorrencias));
5     toc->qtd = qtd_palavras;
6     toc->ocorrencias = (Tocorrencia *)malloc(qtd_palavras * sizeof(
          Tocorrencia));
7     return toc;
8 }
```

2.3.13 adicionar_ocorrencia

A função `adicionar_ocorrencia` permite registrar uma ocorrência de palavra na estrutura `Tocorrencias`. Ela recebe como parâmetros a estrutura, o índice da palavra encontrada e duas coordenadas representando a posição inicial e final da palavra na matriz. A função armazena essas coordenadas no vetor interno, garantindo que todas as ocorrências fiquem organizadas segundo o índice da palavra.

```
1 // Armazena início e fim da ocorrência da palavra no índice indicado
2 void adicionar_ocorrencia(Tocorrencias *toc, int indice, Tcoordenada ini,
      Tcoordenada fim)
3 {
4     if (indice >= 0 && indice < toc->qtd)
5     {
6         toc->ocorrencias[indice].inicio = ini;
7         toc->ocorrencias[indice].fim = fim;
8     }
9 }
```

2.3.14 apagar_tocorrencias

A função `apagar_tocorrencias` é responsável por liberar toda a memória alocada pela estrutura `Tocorrencias`. Ela libera tanto o vetor de ocorrências quanto a estrutura principal, evitando vazamentos de memória ao final da execução do programa.

```
1 // Libera memória de estrutura de ocorrências
2 void apagar_tocorrencias(Tocorrencias *toc)
3 {
4     if (toc)
5     {
6         free(toc->ocorrencias);
7         free(toc);
8     }
9 }
```

2.3.15 imprimir_ocorrencia

A função `imprimir_ocorrencia` imprime na tela a coordenada de início e fim de uma ocorrência. Embora não seja essencial para o funcionamento do programa, ela pode ser utilizada para depuração durante o desenvolvimento.

```
1 // Mostra uma ocorrência individual na tela (opcional, para depuração)
2 void imprimir_ocorrencia(Tocorrencia ocorrencia)
3 {
4     printf("Início: (%d, %d), Fim: (%d, %d)\n",
5            ocorrencia.inicio.linha, ocorrencia.inicio.coluna,
6            ocorrencia.fim.linha, ocorrencia.fim.coluna);
7 }
```

2.3.16 resolve

2.3.17 criar_resolve

A função `criar_resolve` é responsável por inicializar a estrutura principal do resolvedor do caça-palavras. Ela aloca dinamicamente a estrutura `Tresolve`, que contém a matriz de letras e o vetor de palavras a serem buscadas. Essa função recebe como parâmetros o número de linhas e colunas da matriz, bem como a quantidade de palavras, e utiliza as funções dos TADs correspondentes para criar a matriz e alocar o vetor de palavras.

```
1 // Cria a estrutura principal do resolvedor
2 Tresolve *criar_resolve(int linhas, int colunas, int qtd_palavras)
3 {
4     Tresolve *resolve = (Tresolve *)malloc(sizeof(Tresolve));
5     resolve->matriz = criar_matriz(linhas, colunas);              // Cria
          matriz
6     resolve->palavras = (Tpalavra *)malloc(qtd_palavras * sizeof(Tpalavra
          )); // Aloca vetor de palavras
7     resolve->qtd_palavras = qtd_palavras;
8     return resolve;
9 }
```

2.3.18 ler_resolve

A função `ler_resolve` é responsável por preencher a matriz de letras e o vetor de palavras da estrutura `Tresolve`. Ela utiliza as funções dos TADs `Tmatriz` e `Tpalavra` para realizar a leitura dos dados fornecidos pelo usuário, garantindo que a matriz e as palavras estejam corretamente inicializadas para o processo de busca.

```
1 // Lê matriz e palavras, usando funções dos TADs
2 void ler_resolve(Tresolve *resolve)
3 {
4     preencher_matriz(resolve->matriz);                            // Lê
          matriz
5     palavras_preencher(resolve->palavras, resolve->qtd_palavras); // Lê
          palavras
6 }
```

2.3.19 buscar_palavras_resolve

A função `buscar_palavras_resolve` realiza a busca das palavras na matriz de letras em todas as oito direções possíveis (horizontal, vertical e diagonais). Para cada palavra, ela percorre a matriz tentando encontrar uma correspondência. Quando uma palavra é encontrada, suas coordenadas de início e fim são registradas utilizando as funções dos TADs `Tcoordenada` e `Tocorrencias`. Caso a palavra não seja encontrada, uma ocorrência vazia é registrada.

```
1 // Busca palavras na matriz usando função do TAD Tocorrencias para
      armazenar ocorrências
2 Tocorrencias *buscar_palavras_resolve(Tresolve *resolve)
3 {
4     Tocorrencias *toc = criar_tocorrencias(resolve->qtd_palavras);
5
6     // Percorre cada palavra
7     for (int p = 0; p < resolve->qtd_palavras; p++)
8     {
9         Tpalavra *palavra = &resolve->palavras[p];
10        int encontrada = 0;
11
12        // Percorre a matriz para tentar encontrar a palavra
13        for (int i = 0; i < resolve->matriz->linhas && !encontrada; i++)
14        {
```

```
15              for (int j = 0; j < resolve->matriz->colunas && !encontrada;
                    j++)
16              {
17                  // Todas as 8 direções possíveis
18                  int dirs[8][2] = {
19                      {-1, -1},{-1, 0},{-1, 1},{0, -1},{0, 1},{1, -1},{1,
                            0},{1, 1}
20                  };
21
22                  for (int d = 0; d < 8 && !encontrada; d++)
23                  {
24                      int di = dirs[d][0], dj = dirs[d][1];
25                      int x = i, y = j, k;
26
27                      // Verifica se a palavra bate nas posições seguintes
28                      for (k = 0; k < (int)strlen(palavra->palavra); k++)
29                      {
30                          // Se sair dos limites ou letras não baterem, sai
31                          if (x < 0 || y < 0 || x >= resolve->matriz->
                                linhas || y >= resolve->matriz->colunas)
32                              break;
33                          if (resolve->matriz->letras[x][y] != palavra->
                                palavra[k])
34                              break;
35
36                          x += di;
37                          y += dj;
38                      }
39
40                      // Se percorreu toda a palavra, encontrou!
41                      if (k == (int)strlen(palavra->palavra))
42                      {
43                          Tcoordenada ini = criar_coordenada(i, j);
44                          Tcoordenada fim = criar_coordenada(x - di, y - dj
                                );
45                          posicao_palavra(palavra, ini, fim);          //
                                Atualiza posição na palavra
46                          adicionar_ocorrencia(toc, p, ini, fim);      //
                                Salva ocorrência
47                          encontrada = 1;
48                      }
49                  }
50              }
51          }
52
53          // Se não encontrou, marca como não encontrada e adiciona ocorrê
                ncia vazia
54          if (!encontrada)
55          {
56              Tcoordenada coord_zero = criar_coordenada(0, 0);
57              palavra_n_encontrada(palavra);
58              adicionar_ocorrencia(toc, p, coord_zero, coord_zero);
59          }
60      }
61
62      return toc;
63  }
```

2.3.20 imprimir_resultado_resolve

A função `imprimir_resultado_resolve` é responsável por exibir os resultados da busca das palavras na matriz. Ela percorre a estrutura `Tocorrencias` e imprime, para cada palavra, as coordenadas de início e fim, bem como a palavra correspondente. Essa função facilita a visualização dos resultados pelo usuário.

```
1 // Imprime resultados com base nas ocorrências
2 void imprimir_resultado_resolve(Tocorrencias *toc, Tpalavra *palavras,
      int qtd_palavras)
3 {
4     for (int i = 0; i < qtd_palavras; i++)
5     {
6         Tcoordenada ini = toc->ocorrencias[i].inicio;
7         Tcoordenada fim = toc->ocorrencias[i].fim;
8         printf("%d %d %d %d %s\n", ini.linha, ini.coluna, fim.linha, fim.
              coluna, palavras[i].palavra);
9     }
10 }
```

2.3.21 apagar_resolve

A função `apagar_resolve` é responsável por liberar toda a memória alocada pela estrutura `Tresolve`. Ela chama as funções apropriadas para liberar a memória da matriz e das palavras, garantindo que não haja vazamentos de memória ao final da execução do programa.

```
1 // Libera memória alocada
2 void apagar_resolve(Tresolve *resolve)
3 {
4     for (int i = 0; i < resolve->qtd_palavras; i++)
5     {
6         apagar_palavra(&resolve->palavras[i]); // Libera memória de cada
              palavra
7     }
8     apagar_matriz(resolve->matriz); // Libera matriz
9     free(resolve->palavras);
10    free(resolve);
11 }
```

2.3.22 Função `main`

A função `main` é o ponto de entrada de qualquer programa em linguagem C. É nela que a execução do programa se inicia, sendo responsável por coordenar as chamadas às demais funções e controlar o fluxo geral da aplicação. No contexto do programa de caça-palavras apresentado, a função `main` desempenha um papel central na orquestração das etapas de inicialização, entrada de dados, processamento e finalização do programa.

Inicialmente, a função solicita ao usuário que insira as dimensões da matriz (número de linhas e colunas) e a quantidade de palavras que serão buscadas. Esses valores são armazenados nas variáveis `linhas`, `colunas` e `qtd_palavras`, respectivamente. Em seguida, a função `criar_resolve` é chamada para alocar e inicializar a estrutura principal do resolvedor, que inclui a matriz de letras e o vetor de palavras.

Após a criação da estrutura, a função `ler_resolve` é invocada para preencher a matriz e ler as palavras que o usuário deseja encontrar. Essa função utiliza internamente outras funções dos TADs envolvidos para realizar a leitura e o armazenamento adequados dos dados.

Com os dados carregados, a função `buscar_palavras_resolve` é chamada para realizar a busca das palavras na matriz. Essa função percorre a matriz em todas as direções possíveis, procurando correspondências com as palavras fornecidas. As ocorrências encontradas são armazenadas em uma estrutura de ocorrências, que registra as coordenadas de início e fim de cada palavra encontrada.

Em seguida, a função `imprimir_resultado_resolve` é utilizada para exibir os resultados das buscas. Para cada palavra, são impressas as coordenadas de início e fim, seguidas pela própria palavra, permitindo ao usuário visualizar onde cada palavra foi localizada na matriz.

Por fim, as funções `apagar_tocorrencias` e `apagar_resolve` são chamadas para liberar a memória alocada durante a execução do programa, garantindo que não haja vazamentos de memória. A função `main` retorna 0, indicando que o programa foi executado com sucesso.

```c
#include <stdio.h>
#include <stdlib.h>

#include "Tmatriz.h"
#include "Tpalavra.h"
#include "Tocorrencias.h"
#include "resolve.h"

int main()
{
int linhas, colunas, qtd_palavras;

perl
Copiar
Editar
// Leitura das dimensões da matriz e da quantidade de palavras
printf("Digite o numero de linhas\n");
scanf("%d", &linhas);

printf("Digite o numero de colunas \n");
scanf("%d", &colunas);

printf("Digite a quantidade de palavras:\n");
scanf("%d", &qtd_palavras);

// Criação da estrutura para resolver o caça-palavras
printf("Criando estrutura do jogo...\n");
Tresolve *resolve = criar_resolve(linhas, colunas, qtd_palavras);

// Leitura da matriz e das palavras (usa funções internas do resolve)
printf("Digite a matriz de letras (%d linhas de %d letras):\n", linhas,
    colunas);
printf("E depois as %d palavras, uma por linha:\n\n", qtd_palavras);
ler_resolve(resolve);

// Busca as palavras na matriz
printf("Buscando palavras na matriz...\n");
Tocorrencias *ocorrencias = buscar_palavras_resolve(resolve);

// Imprime os resultados
printf("\nResultados encontrados:\n");
imprimir_resultado_resolve(ocorrencias, resolve->palavras, qtd_palavras);

// Libera memória alocada
apagar_tocorrencias(ocorrencias);
apagar_resolve(resolve);

return 0;
}
```

Essa estrutura modular e organizada permite que o programa seja facilmente compreendido, mantido e expandido, facilitando a implementação de novas funcionalidades ou ajustes conforme necessário.

3 Análise de Complexidade dos Algoritmos

A análise de complexidade a seguir considera as variáveis L (número de linhas da matriz), C (número de colunas da matriz) e P (quantidade de palavras a serem buscadas). A complexidade é expressa em termos da notação Big-O, focando no pior caso para cada função.

3.1 Funções de Manipulação de Matrizes

criar_matriz: Esta função aloca memória para uma matriz de tamanho $L \times C$. A alocação envolve um loop para cada linha, onde cada linha aloca um vetor de C caracteres. Portanto, a complexidade é $O(L \times C)$.

preencher_matriz: Percorre cada posição da matriz para ler um caractere, resultando em $L \times C$ operações de leitura. Complexidade: $O(L \times C)$.

imprimir_matriz: Similar à função de preenchimento, percorre toda a matriz para imprimir seus elementos. Complexidade: $O(L \times C)$.

apagar_matriz: Libera a memória alocada para cada linha e, em seguida, para o vetor de ponteiros de linhas. Como cada linha é liberada individualmente, a complexidade é $O(L)$.

3.2 Funções de Manipulação de Coordenadas

criar_coordenada: Cria uma estrutura com dois inteiros. Operações de atribuição simples resultam em complexidade constante: $O(1)$.

verificar_coordenada: Verifica se uma coordenada está dentro dos limites da matriz. Envolve comparações simples, resultando em $O(1)$.

3.3 Funções de Manipulação de Ocorrências

criar_tocorrencias: Aloca memória para um vetor de P ocorrências. Complexidade: $O(P)$.

adicionar_ocorrencia: Atribui valores de início e fim para uma ocorrência específica. Operações de atribuição simples resultam em $O(1)$.

apagar_tocorrencias: Libera a memória alocada para o vetor de ocorrências. Complexidade: $O(1)$.

imprimir_ocorrencia: Imprime os valores de início e fim de uma ocorrência. Operações de impressão simples resultam em $O(1)$.

3.4 Funções de Resolução do Caça-Palavras

criar_resolve: Aloca memória para a estrutura principal, incluindo a matriz e o vetor de palavras. A complexidade depende das funções chamadas internamente:

criar_matriz: $O(L \times C)$

Alocação do vetor de palavras: $O(P)$

Portanto, a complexidade total é $O(L \times C + P)$.

ler_resolve: Chama funções para preencher a matriz e ler as palavras. Supondo que a leitura de cada palavra seja $O(W)$, onde W é o comprimento médio das palavras, a complexidade é:

preencher_matriz: $O(L \times C)$

Leitura das palavras: $O(P \times W)$

Complexidade total: $O(L \times C + P \times W)$.

buscar_palavras_resolve: Esta função é a mais complexa, pois realiza a busca de cada palavra na matriz em todas as direções possíveis. Para cada palavra de comprimento W, percorre cada posição da matriz e verifica em até 8 direções:

Total de posições na matriz: $L \times C$

Direções por posição: 8

Comparações por direção: até W

Portanto, para cada palavra, a complexidade é $O(L \times C \times 8 \times W) = O(L \times C \times W)$. Para P palavras, a complexidade total é $O(P \times L \times C \times W)$.

imprimir_resultado_resolve: Imprime as ocorrências encontradas para cada palavra. Como são P palavras, a complexidade é $O(P)$.

apagar_resolve: Libera a memória alocada para as palavras e a matriz. Supondo que a liberação de cada palavra seja $O(1)$, a complexidade é:

Liberação das palavras: $O(P)$

apagar_matriz: $O(L)$
Complexidade total: $O(P + L)$.

3.5 Função Principal (main)

A função principal realiza as seguintes operações:
Leitura de L, C e P: $O(1)$
criar_resolve: $O(L \times C + P)$
ler_resolve: $O(L \times C + P \times W)$
buscar_palavras_resolve: $O(P \times L \times C \times W)$
imprimir_resultado_resolve: $O(P)$
apagar_tocorrencias: $O(1)$
apagar_resolve: $O(P + L)$
A operação dominante é a busca de palavras na matriz, com complexidade $O(P \times L \times C \times W)$. Portanto, a complexidade total da função principal é $O(P \times L \times C \times W)$.

3.6 Resumo das Complexidades

- criar_matriz: $O(L \times C)$
- preencher_matriz: $O(L \times C)$
- imprimir_matriz: $O(L \times C)$
- apagar_matriz: $O(L)$
- criar_coordenada: $O(1)$
- verificar_coordenada: $O(1)$
- criar_tocorrencias: $O(P)$
- adicionar_ocorrencia: $O(1)$
- apagar_tocorrencias: $O(1)$
- imprimir_ocorrencia: $O(1)$
- criar_resolve: $O(L \times C + P)$
- ler_resolve: $O(L \times C + P \times W)$
- buscar_palavras_resolve: $O(P \times L \times C \times W)$
- imprimir_resultado_resolve: $O(P)$
- apagar_resolve: $O(P + L)$
- main: $O(P \times L \times C \times W)$

A função mais custosa em termos de tempo é `buscar_palavras_resolve`, devido à necessidade de verificar cada posição da matriz em todas as direções para cada palavra. A complexidade total do programa é dominada por esta função, resultando em $O(P \times L \times C \times W)$

4 Apresentação e discussão dos resultados

Com base nos testes realizados, foi possível validar o funcionamento correto do programa em um cenário típico de uso, no qual o usuário insere uma matriz 5x6 de letras e uma lista com 6 palavras a serem buscadas. A execução, exibida na Figura 1, demonstra a capacidade do programa de ler e armazenar corretamente os dados de entrada, realizar a busca pelas palavras na matriz em múltiplas direções e exibir os resultados de forma clara e organizada.

Durante essa execução, o usuário forneceu manualmente os caracteres da matriz, bem como a lista de palavras. O programa processou as informações, identificou corretamente a posição inicial e final de

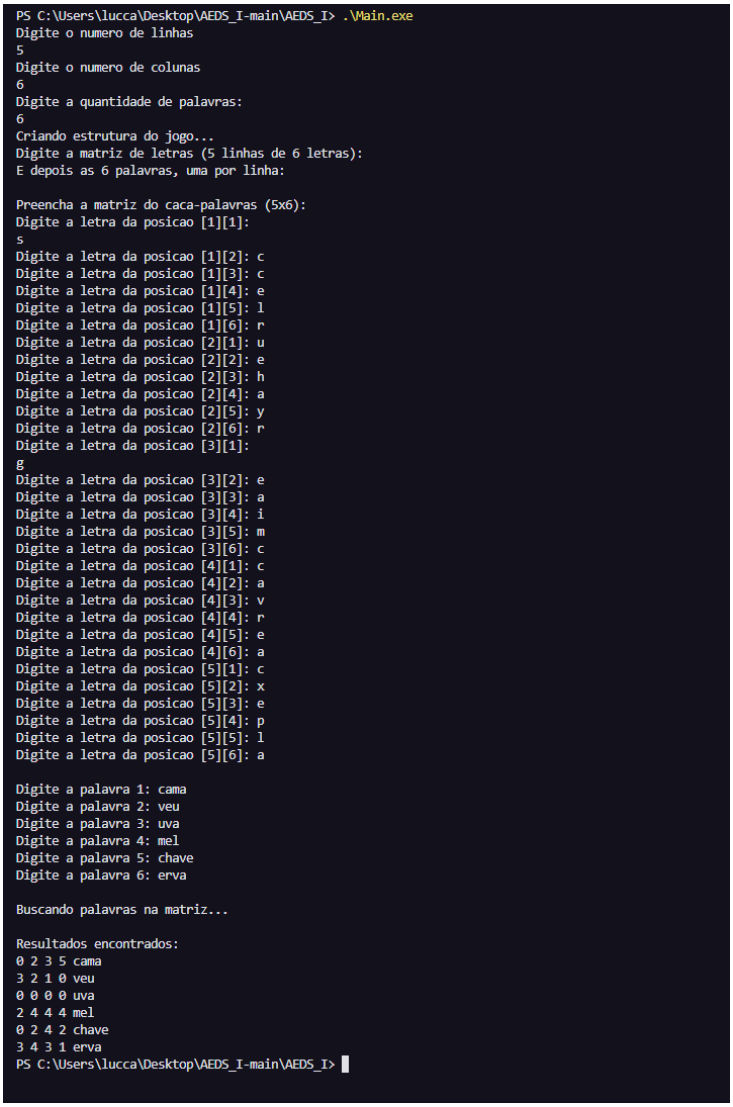

Figura 1: Saída de uma execução típica do trabalho prático

cada palavra dentro da matriz e exibiu os resultados em formato de coordenadas seguidas da respectiva palavra.

Essa execução ilustra não apenas o correto funcionamento dos algoritmos de leitura, armazenamento e busca, mas também a eficiência da estrutura de dados utilizada. O uso de vetores dinâmicos, combinados com um algoritmo de varredura em múltiplas direções, demonstrou ser eficiente para o porte do problema proposto, com tempo de resposta aceitável mesmo em ambientes com recursos limitados, como um processador Pentium 4 com 1 GB de memória RAM.

No entanto, assim como observado em outros testes, a limitação de inserção de dados exclusivamente via entrada padrão torna o processo manual e mais suscetível a erros humanos. Para futuras melhorias, seria interessante implementar uma interface gráfica ou suporte à leitura de arquivos de entrada.

Por fim, os resultados obtidos reforçam que o programa cumpre seu objetivo de maneira eficaz, sendo capaz de identificar corretamente as palavras na matriz, mesmo com variações em direção e posição. As estruturas e algoritmos adotados provaram-se adequados para o problema proposto.

5 Conclusão

A implementação do trabalho de caça-palavras foi uma experiência desafiadora e enriquecedora, especialmente por exigir o uso de técnicas e conceitos que ainda não haviam sido formalmente abordados em sala de aula, como a alocação dinâmica de memória e o uso de ponteiros. Apesar disso, o desenvolvimento do projeto foi bem-sucedido, e os resultados obtidos confirmam a validade da abordagem adotada com a

utilização de TADs (Tipos Abstratos de Dados) e modularização.

Entre as principais dificuldades enfrentadas, destaca-se o uso de alocação dinâmica com malloc, free e ponteiros. Por não terem sido ensinados no primeiro período, foi necessário buscar referências externas e realizar diversos testes para garantir que a memória fosse corretamente alocada e liberada, evitando vazamentos e comportamentos indefinidos. Além disso, a manipulação de ponteiros para structs encadeadas e vetores alocados dinamicamente exigiu bastante atenção e compreensão detalhada do funcionamento interno da linguagem C.

Outro ponto desafiador foi a implementação da função de busca pelas palavras na matriz. A lógica envolvida no escaneamento em todas as oito direções possíveis exigiu raciocínio espacial e cuidados com os limites da matriz, garantindo que não ocorressem acessos fora da área válida de memória. Além disso, foi necessário garantir que, ao encontrar uma palavra, a posição inicial e final fossem corretamente registradas e armazenadas, mantendo a integridade dos dados.

Apesar das dificuldades, o trabalho proporcionou uma oportunidade importante de aplicar conceitos fundamentais de programação estruturada, modularização, leitura de dados e manipulação de estruturas compostas. A utilização de structs organizadas em TADs contribuiu significativamente para a clareza e manutenibilidade do código, separando responsabilidades e facilitando testes e depuração.

Como resultado, o programa final foi capaz de cumprir os requisitos propostos, realizando corretamente a leitura da matriz, o armazenamento das palavras e a busca eficiente de cada uma delas, com exibição clara dos resultados. A experiência, portanto, foi essencial para consolidar o aprendizado prático e desenvolver uma base sólida para os próximos desafios da graduação.

Referências

[1] N. Ziviani, *Projeto de Algoritmos com Implementações em Pascal e C*, 3ª ed., Cengage Learning, 2007.

[2] Overleaf, "Code listing", disponível em: https://pt.overleaf.com/learn/latex/Code_listing, acesso em: 1 de junho de 2025.

[3] Learn LaTeX, "Citações e referências", disponível em: https://www.learnlatex.org/pt/lesson-12, acesso em: 1 de junho de 2025.

[4] PET Computação UFSC, "Inserindo código-fonte - LaTeX", disponível em: https://pet-comp-ufsc.github.io/tutorials/langs/latex/source-code/basics.html, acesso em: 1 de junho de 2025.

[5] Wikipedia, "BibTeX", disponível em: https://pt.wikipedia.org/wiki/BibTeX, acesso em: 1 de junho de 2025.

[6] Wikipedia, "JabRef", disponível em: https://pt.wikipedia.org/wiki/JabRef, acesso em: 1 de junho de 2025.

[7] ZIVIANI, Nivio. *Algoritmos e Estruturas de Dados II*. Departamento de Ciência da Computação, Universidade Federal de Minas Gerais. Disponível em: https://www.dcc.ufmg.br/ nivio/cursos/aed2. Acesso em: 1 jun. 2025.

[8] UNIVERSIDADE FEDERAL DE MINAS GERAIS. *Curso de Linguagem C*. Departamento de Ciência da Computação. Disponível em: https://homepages.dcc.ufmg.br/ nivio/cursos/aed2/roteiro/. Acesso em: 1 jun. 2025.