Universidade Federal de Ouro Preto

Instituto de Ciências Exatas e Aplicadas

Departamento de Computação e Sistemas

ALGORITMOS E ESTRUTURAS DE DADOS

Segundo Trabalho

Caio Mendonça Justino

Prof.: Bruno Hott

João Monlevade
26 de julho de 2025

Sumário

1 Introdução

1.1 Descrição do Problema a Ser Resolvido

O problema para ser resolvido no trabalho foi criar um programa em C que consiga resolver o jogo de caça-palavras. O objetivo é encontrar a posição exata de cada palavra em uma matriz, dada uma matriz de letras de um certo tamanho e uma lista de palavras fornecida pelo usuário. As palavras podem ser organizadas em oito direções diferentes: horizontalmente (da esquerda para a direita e vice-versa), verticalmente (de cima para baixo e vice-versa) e ao longo das quatro diagonais. O programa deve ser capaz de processar a entrada do usuário, realizar a busca com eficiência e, por fim, produzir um relatório indicando as coordenadas inicial e final de cada palavra encontrada ou indicando se ela está faltando (nao está presente no caça-palavras).

1.2 Visão Geral sobre o Funcionamento do Programa

Para resolver o problema proposto, o programa foi estruturado em "modulos", utilizando Tipos Abstratos de Dados (TADs) para encapsular as principais entidades: a Matriz, a Palavra, a Coordenada das palavras e a tabela de Ocorrências (estrutura que será explicada melhor mais pra frente, na qual é principal para a otimização do programa). O funcionamento do programa funciona de a cordo com esses seguintes passos:

- `Entrada de Dados:` Inicialmente, o programa interage com o usuário com o objetivo de obter as dimensões da matriz, as letras que a compõem e a lista de palavras que vai ser procuradas.
- `Armazenamento da Matriz:` A estrutura de dados principal para armazenar o caça-palavras é uma matriz duplamente encadeada (metodo utilizado no trabalho, para que otimizasse a parte do codigo que mais era mal otimizado, que é a busca das palvras dentro da matriz). Essa estrutura, composta por células interligadas por ponteiros (acima, abaixo, direita, esquerda), por ter sua flexibilidade e eficiência na navegação entre os elementos, mesmo o programa tratando de uma matriz "densa".
- `Otimização da Busca:` Antes que inicie a busca das palavras que o usuário forneceu, o programa adota uma estratégia de pré-processamento para otimizar o desempenho. Ele cria uma "tabela de ocorrências", que mapeia cada letra do alfabeto para uma lista de todas as coordenadas onde essa letra aparece na matriz. Dessa forma, ao procurar por uma palavra, em vez de varrer a matriz inteira, a busca é direcionada apenas para as posições onde a primeira letra da palavra ocorre, reduzindo assim o tempo de busca drasticamente.
- `Processo de Solução:` Agora com a matriz preenchida e a tabela de ocorrências pronta, a função principal de solução do programa itera sobre cada palavra da lista. Para cada palavra, ela consulta a tabela de ocorrências para obter os possíveis pontos de partida. A partir de cada um desses pontos, o programa verifica sistematicamente as oito direções possíveis, comparando caractere por caractere até encontrar a palavra completa ou não ter mais possibilidades.
- `Apresentação do Resultado:` Finalizando o processo, o programa exibe a solução, na qual ele lista cada palavra fornecida e suas respectivas coordenadas de início e fim. Se uma palavra não for encontrada, suas coordenadas são apresentadas como zeradas (ex: 0 0 0 0 uva), indicando o fracasso na busca. Por fim, toda a memória alocada dinamicamente durante a execução é liberada.

2 Implementação

O código foi organizado nos seguintes arquivos principais: `matriz.c`, `matriz.h`, `resolver.c`, `resolver.h`, `coordenada.c`, `coordenada.h`, `ocorrencias.c`, `ocorrencias.h`, `palavra.c` e `palavra.h`, que implementam os Tipos Abstratos de Dados (TADs). O arquivo `main.c` contém o programa principal, na qual ele controla a ordem em que o programa executa suas tarefas.

Já o arquivo `resolver.c` funciona como a camada central do programa, integrando as funcionalidades de cada TAD. 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, o `resolver.c` administra a interação entre os módulos para que haja o funcionamento correto do caça-palavras.

Para compilar o programa, foi utilizado o compilador GCC (GNU Compiler Collection) no sistema operacional Windows 11, com o Visual Studio Code como ambiente de desenvolvimento integrado (IDE). A compilação foi realizada pela linha de comando:

```
gcc coordenada.c main.c matriz.c ocorrencias.c palavra.c resolver.c -o jogo.exe
```

Em seguida, executa este comando, para que rode o programa:

```
./jogo.exe
```

2.1 Estruturas de Dados Utilizadas

Para a implementação do caça-palavras, foram definidas as seguintes estruturas de dados principais:

TAD coordenada.h

Programa 1: TAD coordenada.h

```c
#ifndef COORDENADA_H
#define COORDENADA_H
#include "matriz.h"

// indica onde começa ou termina uma palavra
typedef struct {
    int linha;
    int coluna;
} TCoordenada;

// Criar uma nova coordenada
TCoordenada coordenada_criar(int i, int j);

// Verificar se coordenada está dentro dos limites da matriz, retornando 1 em caso positivo e 0 caso contrário
int coordenada_verificar(TCoordenada coordenada, TMatriz* matriz);

#endif
```

TAD matriz.h

Programa 2: TAD matriz.h

```c
#ifndef MATRIZ_H
#define MATRIZ_H

/* Estrutura da célula:
 * Cada célula representa uma letra e está conectada nas direções:
 *  direita/esquerda: elementos que pertencem a mesma linha
 *  acima/abaixo: elementos que pertencem a mesma coluna
 * As células cabeça possuem linha e coluna = (-1,-1)
 */
typedef struct TCelula {
    struct TCelula* direita;
    struct TCelula* esquerda;
    struct TCelula* abaixo;
    struct TCelula* acima;
    int linha, coluna; // índice da linha e coluna
    char letra; // letra armazenada (apenas células de dados)
} TCelula;

```

```c
/* Estrutura da matriz do caça palavras
 * possui  um ponteiro para a célula cabeça principal
 */
typedef struct TMatriz {
    TCelula* cabeca_principal; // ponteiro para a célula cabeça principal (-1,-1)
    int linhas;                // numero de linhas da matriz
    int colunas;               // numero de colunas da matriz
} TMatriz;


// Funções do TAD matriz:

// Inicializa um caça-palavras vazio de dimensões L x C, criando todas as células cabeça (L=linhas C=colunas)
TMatriz* matriz_criar(int linhas, int colunas);

// Função que preenche o caça-palavras a partir da entrada, inserindo letras nas células corretas
int matriz_preencher(TMatriz* matriz);

// Imprime o caça palavras na tela
void matriz_imprimir(TMatriz* matriz);

// Desaloca toda a memória da matriz caça palavras, liberando todas as células
int matriz_apagar(TMatriz* matriz);


// Funções auxiliares para manipulação da matriz:
// Insere uma letra na posição (linha, coluna) da matriz, criando uma célula de dados
int matriz_inserir_letra(TMatriz* matriz, int linha, int coluna, char letra);

// Obtém a letra armazenada na posição (linha, coluna) da matriz
char matriz_obter_letra(TMatriz* matriz, int linha, int coluna);

#endif
```

TAD ocorrencias.h

Programa 3: TAD ocorrencias.h

```c
#ifndef OCORRENCIAS_H
#define OCORRENCIAS_H

#include "coordenada.h"
#include "matriz.h"
#include "palavra.h"

// usamos o "#Define" para Definir o tamanho do alfabeto e o máximo de ocorrências por letra
#define ALF 26
#define MAX_OCORRENCIAS 500

/*
 * 'TOcorrenciaLetra' é uma estrutura que armazena todas as coordenadas onde uma letra aparece na matriz.
```

```
14  * O campo 'quantidade' indica quantas vezes a letra aparece na matriz.
15  */
16 typedef struct {
17     TCoordenada coords[MAX_OCORRENCIAS]; // Vetor de coordenadas da letra
18     int quantidade;                      // Quantidade de ocorrências
           daquela letra
19 } TOcorrenciaLetra;
20
21 /*
22  * 'TOcorrencias_s' é a estrutura principal da tabela de ocorrências.
23  * Para cada letra do alfabeto, guarda um 'TOcorrenciaLetra'
24  */
25 typedef struct TOcorrencias_s {
26     TOcorrenciaLetra letras[ALF]; // Vetor para cada letra do alfabeto
27 } TOcorrencias;
28
29 // Serve para inicializar a tabela de ocorrências, zerando os contadores
30 void ocorrencias_inicializar(TOcorrencias* o);
31
32 // A função: 'ocorrencias_adicionar' adiciona uma nova coordenada no
       vetor de ocorrências daquela letra que corresponde
33 int ocorrencias_adicionar(TOcorrencias* o, TCoordenada c, TMatriz* m);
34
35 // 'ocorrencias_calcular' calcula todas as ocorrências de letras na
       matriz e preenche a tabela
36 int ocorrencias_calcular(TOcorrencias* o, TMatriz* m);
37
38 // 'TOcorrenciaLetra ocorrencias_buscar_palavra' irá retornar o vetor de
       ocorrências da primeira letra de uma palavra
39 TOcorrenciaLetra ocorrencias_buscar_palavra(TOcorrencias* o, TPalavra
       palavra);
40
41 // 'ocorrencias_apagar' serve para liberar recursos da tabela de ocorrê
       ncias
42 int ocorrencias_apagar(TOcorrencias* ocorrencias);
43
44 // 'ocorrencias_buscar_pos' é para buscar a posição final de uma palavra,
        a partir de uma coordenada inicial ('TCoordenada inicio')
45 TCoordenada ocorrencias_buscar_pos(TMatriz* matriz, TPalavra palavra,
       TCoordenada inicio);
46
47 #endif
```

TAD palvra.h

Programa 4: TAD palavra.h

```
1 #ifndef PALAVRA_H
2 #define PALAVRA_H
3
4 #include "coordenada.h"
5 #include "matriz.h"
6
7 // declaração para uso cruzado
8 struct TOcorrencias_s;
9 typedef struct TOcorrencias_s TOcorrencias;
10
11 //    essa Estrutura que representa uma palavra a ser buscada na matriz
```

```c
typedef struct TPalavra_s {
    char* texto; // (ex: "uva")
    TCoordenada inicio; //coordenada de início caso encontrada
    TCoordenada fim; // coordenada de fim caso encontrada
    int foi_encontrada; // 1 se encontrada, 0 caso contrário
} TPalavra;

/* 'palavra_criar' :
 * Cria uma nova estrutura de palavra, alocando memória para o texto
 * Par metro usado é 'texto_original' - (uma string da palavra)
 * Retorno: A estrutura 'TPalavra' inicializada
 */
TPalavra palavra_criar(const char* texto_original);

/* 'palavra_marcar_encontrada' :
 * Atualiza uma palavra encontrada, marcando ela como encontrada e guardando suas coordenadas
 * Par metros: um ponteiro 'TPalavra* p', 'TCoordenada inicio, TCoordenada fim' - coordenadas
 */
void palavra_marcar_encontrada(TPalavra* p, TCoordenada inicio, TCoordenada fim);

/* 'palavra_buscar_pos' :
 * Serve para buscar a palavra em uma coordenada específica do caça palavras
 * Par metros: 'TPalavra palavra' , 'TMatriz* matriz' - um ponteiro, 'TCoordenada pos' - coordenada inicial
 * Retorno: irá retornar a coordenada final da palavra ou (-1,-1) se não encontrada
 */
TCoordenada palavra_buscar_pos(TPalavra palavra, TMatriz* matriz, TCoordenada pos);

/* 'palavra_buscar' :
 * Essa função irá buscar a palavra no caça palavras usando a "tabela de ocorrências"
 * Retorno: coordenada final da palavra ou (-1,-1) se não encontrada
 */
TCoordenada palavra_buscar(TPalavra palavra, TMatriz* matriz, TOcorrencias* ocorr);

/* 'palavras_add' :
 * adiciona uma palavra ao vetor de 'palavras'
 * Par metros: palavras_vet - vetor de TPalavra, contador - ponteiro para quantidade atual,
 * capacidade - tamanho máximo, palavra_nova - palavra a adicionar
 * Retorno: novo tamanho do vetor ou -1 em caso de erro
 */
int palavras_add(TPalavra* palavras_vet, int* contador, int capacidade, TPalavra palavra_nova);

/* 'palavras_preencher' :
 * Preenche o vetor de palavras a partir da entrada
 * Retorno: 1 se deu sucesso, 0 se deu erro
 */
int palavras_preencher(TPalavra* palavras, int num_palavras);

```

```
59 /*
60  * Imprime somente os dados de uma palavra
61  */
62 void palavra_imprimir(TPalavra p);
63
64 /* 'palavras_imprimir_solucao' :
65  * Serve para imprimir a solução do caça palavras, mostrando todas as
        palavras e suas posições
66  */
67 void palavras_imprimir_solucao(TPalavra* palavras, int num_palavras);
68
69 /*
70  * Libera a memória alocada para o texto da palavra.
71  */
72 void palavra_apagar(TPalavra* p);
73
74 /* 'palavras_apagar_todas' :
75  * Libera a memória alocada para o texto de todas as palavras do vetor
76  * Retorno: 1 se deu sucesso, 0 se deu erro
77  */
78 int palavras_apagar_todas(TPalavra* palavras, int num_palavras);
79
80 #endif
```

TAD resolver.h

Programa 5: TAD resolver.h

```
1 #ifndef RESOLVER_H
2 #define RESOLVER_H
3 #include "matriz.h"
4
5 // Declaração antecipada das estruturas usadas
6 struct TPalavra_s;
7 typedef struct TPalavra_s TPalavra;
8 struct TOcorrencias_s;
9 typedef struct TOcorrencias_s TOcorrencias;
10
11 /* 'matriz_solucionar' :
12  * Função que "resolve" o caça-palavras por completo
13  * Ele procura todas as palavras do vetor na matriz e logo ap s marca as
        que forem encontradas
14  * Par metros :
15  *
16  *   - matriz: ponteiro para a matriz duplamente encadeada (TMatriz*
        matriz)
17  *
18  *   - ocorr: tabela de ocorrências já preenchida (TOcorrencias* ocorr)
19  *
20  *   - palavras: vetor de palavras a serem buscadas (TPalavra* palavras)
21  *
22  *   - num_palavras: Cria uma variavel em "INT" paraguardar a quantidade
        de palavras no vetor (int num_palavras)
23  *
24  * Retorno da função :
25  *   - '1' se todas as buscas foram realizadas (mesmo encontrando todas
        ou não)
26  *
```

```
27  *   - '0' se algum ponteiro for inválido
28  */
29 int matriz_solucionar(TMatriz* matriz, TOcorrencias* ocorr, TPalavra*
       palavras, int num_palavras);
30
31 #endif
```

2.2 Funções e Procedimentos

TAD coordenada.c

Programa 6: TAD coordenada.c

```
1 #include <stdio.h>
2 #include "coordenada.h"
3 #include "matriz.h"
4
5 // Criar uma nova coordenada
6 TCoordenada coordenada_criar(int i, int j) {
7     TCoordenada c; // cria uma variavel do "tipo" 'TCoordenada'
8     c.linha = i; // o campo 'linha' recebe valor do parametro 'i'
9     c.coluna = j; // o campo 'coluna' recebe valor do parametro 'j'
10    return c;  // retorna a estrutura 'c' preenchida
11 }
12
13 // Verificar se coordenada está dentro dos limites da matriz, retornando
       1 em caso positivo e 0 caso contrario.
14 int coordenada_verificar(TCoordenada coordenada, TMatriz* matriz) {
15    if (!matriz) return 0; // verifica se a matriz é nula sendo 0 = False
          | 1 = True
16    return (coordenada.linha >= 0 && coordenada.linha < matriz->linhas &&
17            coordenada.coluna >= 0 && coordenada.coluna < matriz->colunas
                );
18 }
```

TAD matriz.c

Programa 7: TAD matriz.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "matriz.h"
4
5 /*
6  * Essa funçao cria uma matriz duplamente encadeada de dimensões 'linhas'
       x 'colunas' que:
7  * Aloca a célula cabeça principal, as cabeças de linha e coluna, e
       conecta todas.
8  */
9 TMatriz* matriz_criar(int linhas, int colunas) {
10    TMatriz* matriz = (TMatriz*)malloc(sizeof(TMatriz));
11    if (!matriz) return NULL;
12
13    matriz->linhas = linhas;
14    matriz->colunas = colunas;
15
16    // aloca a célula cabeça principal (-1, -1)
17    TCelula* cabeca_principal = (TCelula*)malloc(sizeof(TCelula));
```

```
18      cabeca_principal->linha = -1;
19      cabeca_principal->coluna = -1;
20      cabeca_principal->direita = cabeca_principal;
21      cabeca_principal->esquerda = cabeca_principal;
22      cabeca_principal->abaixo = cabeca_principal;
23      cabeca_principal->acima = cabeca_principal;
24
25      matriz->cabeca_principal = cabeca_principal;
26
27      // Cria cabeças das colunas e conecta elas horizontalmente
28      TCelula* ultimo_col = cabeca_principal;
29      for (int j = 0; j < colunas; j++) {
30          TCelula* nova_cabeca = (TCelula*)malloc(sizeof(TCelula));
31          nova_cabeca->linha = -1;
32          nova_cabeca->coluna = j;
33          nova_cabeca->abaixo = nova_cabeca; // Inicialmente ela irá aponta
                para si mesma
34          nova_cabeca->acima = nova_cabeca;
35          // Conecta horizontalmente
36          ultimo_col->direita = nova_cabeca;
37          nova_cabeca->esquerda = ultimo_col;
38          ultimo_col = nova_cabeca;
39      }
40      // Assim fecha o ciclo horizontal
41      ultimo_col->direita = cabeca_principal;
42      cabeca_principal->esquerda = ultimo_col;
43
44      // Cria as cabeças das linhas e conecta elas verticalmente
45      TCelula* ultimo_lin = cabeca_principal;
46      for (int i = 0; i < linhas; i++) {
47          TCelula* nova_cabeca = (TCelula*)malloc(sizeof(TCelula));
48          nova_cabeca->linha = i;
49          nova_cabeca->coluna = -1;
50          nova_cabeca->direita = nova_cabeca; // Inicialmente ela irá
                aponta para si mesma
51          nova_cabeca->esquerda = nova_cabeca;
52          // Conecta verticalmente
53          ultimo_lin->abaixo = nova_cabeca;
54          nova_cabeca->acima = ultimo_lin;
55          ultimo_lin = nova_cabeca;
56      }
57      // Assim fecha o ciclo vertical
58      ultimo_lin->abaixo = cabeca_principal;
59      cabeca_principal->acima = ultimo_lin;
60
61      return matriz;
62  }
63
64  /*
65   * A função 'matriz_inserir_letra: Insere uma letra na posição (linha,
        coluna) da matriz e:
66   * Cria uma célula de dados. E depois conecta nas listas da linha e da
        coluna.
67   */
68  int matriz_inserir_letra(TMatriz* matriz, int linha, int coluna, char
        letra) {
69      if (!matriz || linha < 0 || linha >= matriz->linhas || coluna < 0 ||
            coluna >= matriz->colunas) {
```

```
70          return 0;
71      }
72
73      // TCelula* pLinha é um ponteiro que navega até a cabeça da linha '
            linha'
74      TCelula* pLinha = matriz->cabeca_principal->abaixo;
75      for (int i = 0; i < linha; i++) pLinha = pLinha->abaixo;
76
77      // Com o laço While irá Navegar na linha até a posição de inserção (
            na ordem crescente de coluna)
78      while (pLinha->direita != pLinha && pLinha->direita->coluna != -1 &&
            pLinha->direita->coluna < coluna) {
79          pLinha = pLinha->direita;
80      }
81
82      // O pomteiro pColuna irá navegar até a cabeça da coluna 'coluna'
83      TCelula* pColuna = matriz->cabeca_principal->direita;
84      for (int j = 0; j < coluna; j++) pColuna = pColuna->direita;
85
86      // Com o laço While irá navegar na coluna até a posição de inserção (
            na ordem crescente de linha)
87      while (pColuna->abaixo != pColuna && pColuna->abaixo->linha != -1 &&
            pColuna->abaixo->linha < linha) {
88          pColuna = pColuna->abaixo;
89      }
90
91      // O ponteiro 'nova' aloca e preenche nova célula de dados
92      TCelula* nova = (TCelula*) malloc(sizeof(TCelula));
93      nova->linha = linha;
94      nova->coluna = coluna;
95      nova->letra = letra;
96
97      // irá Conectar horizontalmente na linha
98      nova->direita = pLinha->direita;
99      nova->esquerda = pLinha;
100     pLinha->direita->esquerda = nova;
101     pLinha->direita = nova;
102
103     // e agora Conecta verticalmente na coluna
104     nova->abaixo = pColuna->abaixo;
105     nova->acima = pColuna;
106     pColuna->abaixo->acima = nova;
107     pColuna->abaixo = nova;
108
109     return 1;
110 }
111
112 /* 'matriz_preencher' :
113  * Serve para Preencher a matriz lendo as letras da entrada.
114  * e para cada linha, ela insere as letras nas posições corretas.
115  */
116 int matriz_preencher(TMatriz* matriz) {
117     if (!matriz) return 0;
118     char* buffer = (char*)malloc((matriz->colunas + 2) * sizeof(char));
            // Buffer para leitura
119
120     for (int i = 0; i < matriz->linhas; i++) {
121         printf("Linha %d: ", i + 1);
```

```
122         if (scanf("%s", buffer) != 1) {
123             free(buffer);
124             return 0;
125         }
126         for (int j = 0; j < matriz->colunas; j++) {
127             if (buffer[j] == '\0') break;
128             matriz_inserir_letra(matriz, i, j, buffer[j]);
129         }
130     }
131     free(buffer);
132     return 1;
133 }
134
135 /* 'matriz_imprimir' :
136  * Imprime a matriz navegando o ponteiro pelas listas encadeadas.
137  * Para cada celula de dados encontrada, imprime a letra, se caso contrá
       rio, irá imprimir: '.'.
138  */
139 void matriz_imprimir(TMatriz* matriz) {
140     if (!matriz) return;
141     TCelula* pLinha = matriz->cabeca_principal->abaixo;
142
143     for (int i = 0; i < matriz->linhas; i++) {
144         TCelula* pCelula = pLinha->direita;
145         for (int j = 0; j < matriz->colunas; j++) {
146             if (pCelula != pLinha && pCelula->coluna == j) {
147                 printf("%c ", pCelula->letra) ;
148                 pCelula = pCelula->direita ;
149             } else {
150                 printf(". ") ;
151             }
152         }
153         printf("\n");
154         pLinha = pLinha->abaixo;
155     }
156 }
157
158 /* Matriz_apagar':
159  * Serve para liberar toda a memória alocada para a matriz, incluindo cé
       lulas de dados e cabeças.
160  */
161 int matriz_apagar(TMatriz* matriz) {
162     if (matriz == NULL) return 1;
163
164     // Apaga todas as linhas (células de dados e cabeças de linha:
165     TCelula* pLinha = matriz->cabeca_principal->abaixo;
166     while (pLinha != matriz->cabeca_principal) {
167         TCelula* pCelula = pLinha->direita;
168         while (pCelula != pLinha) {
169             TCelula* proxima = pCelula->direita;
170             free(pCelula);
171             pCelula = proxima;
172         }
173         TCelula* proximaLinha = pLinha->abaixo;
174         free(pLinha);
175         pLinha = proximaLinha;
176     }
177
```

```
178     // Apaga todas as cabeças de coluna:
179     TCelula* pColuna = matriz->cabeca_principal->direita;
180     while (pColuna != matriz->cabeca_principal) {
181         TCelula* proximo = pColuna->direita;
182         free(pColuna);
183         pColuna = proximo;
184     }
185
186     // Apaga a cabeça principal e a estrutura da matriz:
187     free(matriz->cabeca_principal);
188     free(matriz);
189     return 1;
190 }
191
192 /* 'matriz_obter_letra' :
193  * a função Retorna a letra armazenada na posição (linha, coluna).
194  *  e navega pela lista da linha até encontrar a célula de dados
        correspondente.
195  */
196 char matriz_obter_letra(TMatriz* matriz, int linha, int coluna) {
197     if (!matriz || linha < 0 || linha >= matriz->linhas || coluna < 0 ||
        coluna >= matriz->colunas) {
198         return '\0';
199     }
200     TCelula* pLinha = matriz->cabeca_principal->abaixo;
201     for (int i = 0; i < linha; i++) pLinha = pLinha->abaixo;
202
203     TCelula* pCelula = pLinha->direita;
204     while (pCelula != pLinha) {
205         if (pCelula->coluna == coluna) {
206             return pCelula->letra;
207         }
208         if (pCelula->coluna > coluna) break;
209         pCelula = pCelula->direita;
210     }
211     return '\0';
212 }
```

TAD ocorrencias.c

Programa 8: TAD ocorrencias.c

```
1 #include <stdio.h>
2 #include <ctype.h>
3 #include <string.h>
4 #include <stdlib.h>
5 #include "ocorrencias.h"
6 #include "matriz.h"
7
8
9  //Função para inicializar a tabela de ocorrências
10  //Zera o contador de cada letra do alfabeto
11 void ocorrencias_inicializar(TOcorrencias* o) {
12     if (!o) return;
13     for (int i = 0; i < ALF; i++) {
14         o->letras[i].quantidade = 0;
15     }
16 }
```

```
17
18 /* 'ocorrencias_adicionar' :
19  * Serve para adicionar uma coordenada ao vetor de ocorrências da letra
        correspondente
20  * Usa matriz_obter_letra para acessar a letra na matriz.
21  */
22 int ocorrencias_adicionar(TOcorrencias* o, TCoordenada c, TMatriz* m) {
23     if (!o || !m) return 0;
24
25     // o 'IF' aqui serve para verificar se a coordenada está nos limites
           da matriz
26     if (c.linha < 0 || c.linha >= m->linhas || c.coluna < 0 || c.coluna
          >= m->colunas) {
27         return 0;
28     }
29
30     // Com 'letra_na_matriz' é para Obter a letra na posição (linha,
           coluna)
31     char letra_na_matriz = matriz_obter_letra(m, c.linha, c.coluna);
32     if (letra_na_matriz == '\0') {
33         // nesse 'IF' é como se: Se letra_na_matriz Não há letra nessa
               posição
34         return 1; // retorna 1
35     }
36
37     letra_na_matriz = tolower(letra_na_matriz);
38     int idx = letra_na_matriz - 'a'; // a variavel 'idx' srve para
           calcular o indice da letra no vetor de ocorrencias, transformando
           o caracter da letra em um numero de 0 a 25
39
40     // Nesse 'IF' será para verificar se é uma letra válida
41     if (idx < 0 || idx >= ALF) {
42         return 0;
43     }
44     // Aqui verifica se ainda há espaço para adicionar a ocorrência
45     if (o->letras[idx].quantidade >= MAX_OCORRENCIAS) {
46         return 0;
47     }
48
49     // Adiciona a coordenada ao vetor
50     o->letras[idx].coords[o->letras[idx].quantidade] = c;
51     o->letras[idx].quantidade++;
52     return 1;
53 }
54
55 /* 'ocorrencias_calcular' :
56  * Serve para  calcular todas as ocorrências de letras na matriz
57  * e para cada posição, irá chamar ocorrencias_adicionar
58  */
59 int ocorrencias_calcular(TOcorrencias* o, TMatriz* m) {
60     if (!o || !m) return 0;
61     ocorrencias_inicializar(o); // Zerar antes de começar
62     for (int i = 0; i < m->linhas; i++) {
63         for (int j = 0; j < m->colunas; j++) {
64             TCoordenada c = coordenada_criar(i, j);
65             ocorrencias_adicionar(o, c, m);
66         }
67     }
```

```
68      return 1;
69  }
70
71  /* 'ocorrencias_buscar_palavra' :
72   * Serve para buscar rapidamente todas as posições onde a primeira letra
         da palavra aparece
73   */
74  TOcorrenciaLetra ocorrencias_buscar_palavra(TOcorrencias* o, TPalavra
        palavra) {
75      TOcorrenciaLetra vazia;
76      vazia.quantidade = 0;
77
78      if (!o || !palavra.texto || strlen(palavra.texto) == 0) {
79          return vazia;
80      }
81
82      char primeira_letra = tolower(palavra.texto[0]);
83      int idx = primeira_letra - 'a';
84
85      if (idx < 0 || idx >= ALF) {
86          return vazia;
87      }
88      return o->letras[idx];
89  }
90
91  /* 'ocorrencias_buscar_pos' : é uma função do tipo 'TCoordenada' que
        busca a posição final
92  de uma palavra a partir de uma coordenada inicial, testando todas as dire
        ções possíveis */
93  TCoordenada ocorrencias_buscar_pos(TMatriz* matriz, TPalavra palavra,
        TCoordenada inicio) {
94      int len = strlen(palavra.texto);
95      int direcoes[8][2] = {
96          {0, 1}, {1, 0}, {0, -1}, {-1, 0},
97          {1, 1}, {1, -1}, {-1, 1}, {-1, -1}
98      };
99      for (int d = 0; d < 8; d++) {
100         int dr = direcoes[d][0], dc = direcoes[d][1];
101         int l = inicio.linha, c = inicio.coluna, k;
102         for (k = 0; k < len; k++) {
103             int nl = l + k * dr, nc = c + k * dc;
104             if (nl < 0 || nl >= matriz->linhas || nc < 0 || nc >= matriz
                    ->colunas) break;
105             if (matriz_obter_letra(matriz, nl, nc) != palavra.texto[k])
                    break;
106         }
107         if (k == len) {
108             TCoordenada fim = {inicio.linha + (len-1)*dr, inicio.coluna +
                    (len-1)*dc};
109             return fim;
110         }
111     }
112     TCoordenada nao_encontrado = {-1, -1};
113     return nao_encontrado;
114 }
115
116 /* 'ocorrencias_apagar' :
117  * "Apaga" a tabela de ocorrências, zerando todos os dados.
```

```
118  */
119 int ocorrencias_apagar(TOcorrencias* ocorrencias) {
120     if (!ocorrencias) return 0;
121     for (int i = 0; i < ALF; i++) {
122         ocorrencias->letras[i].quantidade = 0;
123         for (int j = 0; j < MAX_OCORRENCIAS; j++) {
124             ocorrencias->letras[i].coords[j].linha = -1;
125             ocorrencias->letras[i].coords[j].coluna = -1;
126         }
127     }
128     return 1;
129 }
```

TAD palavra.c

Programa 9: TAD palvra.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 #include <ctype.h>
 5 #include "palavra.h"
 6 #include "coordenada.h"
 7 #include "matriz.h"
 8 #include "ocorrencias.h"
 9
10 //    uma "declaração antecipada" para que funções como '
       palavra_buscar_pos' possam chamá-la antes mesmo de sua definição
       completa aparecer no código
11 int verificar_palavra_em_direcao(TMatriz* matriz, const char*
       texto_palavra, TCoordenada inicio, int dr, int dc, TCoordenada* p_fim)
       ;
12
13 /* 'palavra_criar' :
14  * Objetivo:     Funciona como um "construtor" para a estrutura 'TPalavra'
15
16  * Ela inicializa todos os campos de uma nova palavra
17
18  * Lógica:       Define valores padrão para os campos (como -1 para
        coordenadas e 0 para 'foi_encontrada'). E aloca memória dinamicamente
         para o texto da palavra e o converte para minúsculas, facilitando as
         comparações futuras
19
20  * Par metros:
21   - texto_original: A string (char*) da palavra a ser criada
22
23  * Retorno:
24   - Uma estrutura 'TPalavra'
25  */
26 TPalavra palavra_criar(const char* texto_original) {
27     TPalavra p;
28     // Inicializa os campos com valores padrões
29     p.texto = NULL;
30     p.foi_encontrada = 0;
31     p.inicio = coordenada_criar(-1, -1);
32     p.fim = coordenada_criar(-1, -1);
33
34     if (texto_original == NULL) return p;
```

```
35
36      // Aloca memória para a string da palavra
37      int len = strlen(texto_original);
38      p.texto = (char*)malloc(len + 1); // +1 para o caractere nulo '\0'
39
40      if (p.texto != NULL) { //Copia a palavra, convertendo a letra para
            minúsculo
41          for (int i = 0; i < len; i++) p.texto[i] = tolower(texto_original
                [i]);
42          p.texto[len] = '\0'; // Adiciona o terminador nulo ao final
43      } else {
44          fprintf(stderr, "Erro: Falha ao alocar memoria para o texto da
                palavra.\n");
45      }
46      return p;
47  }
48
49  /* 'palavra_marcar_encontrada' :
50   * Objetivo: Atualizar o estado de uma palavra quando é encontrada na
         matriz
51
52   * Lógica: Recebe um ponteiro (p) para uma 'TPalavra' e modifica seus
         campos internos para mostrar que ela foi encotrada, salvando sua
         coordenada de inicio e fim
53
54   * Par metros:
55   * - p: Ponteiro para a 'TPalavra' que será modificada
56   * - inicio: A coordenada onde começa
57   * - fim: A coordenada onde termina
58   */
59  void palavra_marcar_encontrada(TPalavra* p, TCoordenada inicio,
        TCoordenada fim) {
60      // Usamos um ponteiro para modificar a palavra original
61      if (p != NULL) {
62          p->inicio = inicio;
63          p->fim = fim;
64          p->foi_encontrada = 1; // 1 = busca realizada com sucesso
65      }
66  }
67
68  /* 'verificar_palavra_em_direcao' : uma "Função Auxiliar"
69   * Com o Objetivo de verificar se uma palavra existe na matriz a partir
         de um ponto inicial e em uma única direção específica
70
71   * a Logica é Usar um loop para percorrer os caracteres da palavra. A
         cada passo, calcula a próxima coordenada na matriz com base na direçã
         o (dr, dc) e compara a letra
72   * Par metros:
73   * - matriz é o pponteiro para a nossa matriz
74   * - texto_palavra: A string da palavra que estamos procurando.
75   * - inicio: A coordenada (l,c) de onde a verificação deve começar.
76   * - dr, dc: O vetor de direção ("delta linha, delta coluna"). Ex: (1,0)
         = para baixo
77   * - p_fim: Ponteiro para uma 'TCoordenada' onde o fim da palavra será
         salvo
78   * Retorno:
79    - 1 se a palavra foi encontrada ou 0 caso não
80   */
```

```
81 int verificar_palavra_em_direcao(TMatriz* matriz, const char*
       texto_palavra, TCoordenada inicio, int dr, int dc, TCoordenada* p_fim)
       {
82     if (!matriz || !texto_palavra || !p_fim) return 0;
83     int len = strlen(texto_palavra);
84     if (len == 0) return 0;
85     // Loop que "caminha" pela matriz na direção especificada
86     for (int i = 0; i < len; i++) {
87         // Agora Calcula a próxima coordenada a ser verificada
88         int l = inicio.linha + i * dr;
89         int c = inicio.coluna + i * dc;
90         TCoordenada atual = coordenada_criar(l, c);
91
92         // Verificar se a coordenada calculada ainda está dentro da
               matriz
93         if (!coordenada_verificar(atual, matriz)) return 0;
94
95         //Buscar a letra na estrutura da matriz encadeada
96         char letra_na_matriz = matriz_obter_letra(matriz, l, c);
97
98         // Comparar a letra da matriz com a letra esperada da palavra
99         if (tolower(letra_na_matriz) != texto_palavra[i]) return 0;
100    }
101    // quando o loop terminar, a palavra inteira foi encontrada
102    // e então salva a coordenada final.
103    p_fim->linha = inicio.linha + (len - 1) * dr;
104    p_fim->coluna = inicio.coluna + (len - 1) * dc;
105    return 1; // 1 = executou correto
106 }
107
108 TCoordenada palavra_buscar_pos(TPalavra palavra, TMatriz* matriz,
       TCoordenada pos) {
109    TCoordenada coord_final_encontrada = coordenada_criar(-1, -1);
110    if (!matriz || !palavra.texto) return coord_final_encontrada;
111    // Vetores que representam as 8 direções: N, NE, L, SE, S, SO, O, NO
112    int dr[] = {-1, -1, 0, 1, 1, 1, 0, -1};
113    int dc[] = {0, 1, 1, 1, 0, -1, -1, -1};
114    // Aqui testa cada uma das 8 direções
115    for (int i = 0; i < 8; i++) {
116        TCoordenada fim_temp;
117        if (verificar_palavra_em_direcao(matriz, palavra.texto, pos, dr[i
               ], dc[i], &fim_temp)) {
118            return fim_temp; // Encontrou, então retorna a coordenada
                   final e para de executar
119        }
120    }
121    // Se o loop terminar sem achar, retorna a coordenada inválida
122    return coord_final_encontrada;
123 }
124
125 TCoordenada palavra_buscar(TPalavra palavra, TMatriz* matriz,
       TOcorrencias* ocorr) {
126    TCoordenada coord_final_encontrada = coordenada_criar(-1, -1);
127
128    // Aqui ele pega a lista de possíveis pontos de partida da "tabela de
           ocorrências"
129    if (!matriz || !palavra.texto || strlen(palavra.texto) == 0 || !ocorr
           ) return coord_final_encontrada;
```

```
130     TOcorrenciaLetra ocorrs_letra = ocorrencias_buscar_palavra(ocorr,
            palavra);
131
132     // Itera apenas sobre os pontos de partida promissores
133     for (int i = 0; i < ocorrs_letra.quantidade; i++) {
134         TCoordenada pos_inicial = ocorrs_letra.coords[i];
135         // Tenta encontrar a palavra completa a partir deste ponto
136         TCoordenada fim = palavra_buscar_pos(palavra, matriz, pos_inicial
                );
137         if (fim.linha != -1) return fim; // se 'return fim', quer dizer
                que achou
138     }
139     return coord_final_encontrada;
140 }
141
142 //Serve para adicionar uma nova palavra a um vetor de palavras
143 int palavras_add(TPalavra* palavras_vet, int* contador, int capacidade,
        TPalavra palavra_nova) {
144     if (*contador < capacidade) {
145         palavras_vet[*contador] = palavra_nova;
146         (*contador)++; // Incrementa o contador de palavras no vetor
147         return *contador;
148     }
149     fprintf(stderr, "Erro: Capacidade do vetor de palavras excedida.\n");
150     return -1;
151 }
152
153 // 'palavras_preencher' : Preencher um vetor de palavras lendo da entrada
154 int palavras_preencher(TPalavra* palavras_vet, int num_palavras) {
155     if (!palavras_vet || num_palavras <= 0) return 0;
156     printf("Digite as %d palavras (uma por linha):\n", num_palavras);
157     for (int i = 0; i < num_palavras; i++) {
158         char buffer[100];
159         if (scanf("%99s", buffer) != 1) {
160             fprintf(stderr, "Erro ao ler palavra %d.\n", i + 1);
161             for(int k=0; k<i; ++k) palavra_apagar(&palavras_vet[k]);
162             return 0;
163         }
164         palavras_vet[i] = palavra_criar(buffer);
165         if(palavras_vet[i].texto == NULL){
166             fprintf(stderr, "Erro ao criar palavra %d ('%s') com
                    palavra_criar.\n", i + 1, buffer);
167             for(int k=0; k<i; ++k) palavra_apagar(&palavras_vet[k]);
168             return 0;
169         }
170     }
171     return 1;
172 }
173
174 //função de imprimir os dados de uma única palavra no formato da solução
175 void palavra_imprimir(TPalavra p) {
176     if (p.foi_encontrada) {
177         printf("%d %d %d %d %s\n", p.inicio.linha, p.inicio.coluna, p.fim
                .linha, p.fim.coluna, p.texto ? p.texto : "(null)");
178     } else {
179         printf("0 0 0 0 %s\n", p.texto ? p.texto : "(null)");
180     }
181 }
```

```
182
183 // Imprime a solução final, iterando sobre todas as palavras
184 void palavras_imprimir_solucao(TPalavra* palavras, int num_palavras) {
185     if (!palavras) return;
186     for (int i = 0; i < num_palavras; i++) palavra_imprimir(palavras[i]);
187 }
188 // Libera a memória alocada para uma palavra (o 'texto')
189 void palavra_apagar(TPalavra* p) {
190     if (p != NULL && p->texto != NULL) {
191         free(p->texto);
192         p->texto = NULL;
193     }
194 }
195
196 // Função que libera a memória de todas as palavras em um vetor
197 int palavras_apagar_todas(TPalavra* palavras_vet, int num_palavras) {
198     if (!palavras_vet) return 0;
199     for (int i = 0; i < num_palavras; i++) palavra_apagar(&palavras_vet[i
        ]);
200     return 1;
201 }
```

TAD resolver.c

Programa 10: TAD resolver.c

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <ctype.h>
4 #include <stdlib.h>
5 #include "resolver.h"
6 #include "palavra.h"
7 #include "ocorrencias.h"
8 #include "coordenada.h"
9 #include "matriz.h"
10
11 /* 'Função: matriz_solucionar' :
12  * Objetivo: buscar todas as palavras na matriz.
13  * Lógica:
14  * Para cada palavra na lista de palavras a serem buscadas :
15  * 1. Pede pra 'ocorrencias' a lista de todas as posições da "primeira
      letra".
16  * 2. Para cada uma dessas posiçoes, chama 'palavra_buscar_pos' para
      verificar se a palavra inteira começa em alguma das 8 direçoes
17  * 3. Se 'palavra_buscar_pos' retorna uma coordenada válida, significa
      que
18  * a palavra foi encontrada (return 1)
19  */
20 int matriz_solucionar(TMatriz* matriz, TOcorrencias* ocorr, TPalavra*
    palavras, int num_palavras) {
21     if (!matriz || !palavras || !ocorr) return 0;
22
23     for (int i = 0; i < num_palavras; i++) {
24         TOcorrenciaLetra ocorrs_letra = ocorrencias_buscar_palavra(ocorr,
               palavras[i]);
25
26         for (int j = 0; j < ocorrs_letra.quantidade; j++) {
27             TCoordenada inicio = ocorrs_letra.coords[j];
```

```
28              TCoordenada fim;
29
30              // aqui Tenta buscar a palavra completa a partir deste ponto
                    de início
31              fim = palavra_buscar_pos(palavras[i], matriz, inicio);
32
33              // O if significa que, se a coordenada de 'fim' for válida (
                    diferente de -1)
34              if (fim.linha != -1 && fim.coluna != -1) {
35                  palavra_marcar_encontrada(&palavras[i], inicio, fim);
36                  break; //Chama 'palavra_marcar_encontrada' para guardar o
                        resultado e interrompe a busca daquela palavra (pelo
                        'break')
37              }
38          }
39      }
40      return 1;
41  }
```

2.3 Função Principal

2.3.1 main.c

A função main atua como o "maestro"do programa, sendo responsável por coordenar a interação entre todos os Tipos Abstratos de Dados. Ela gerencia o ciclo de vida completo da aplicação: inicia com a coleta e validação dos dados de entrada do usuário (matriz e palavras), prepara as estruturas para uma busca eficiente e, em seguida, delega a tarefa de resolução ao módulo resolver. Após a exibição dos resultados, a main garante que toda a memória alocada dinamicamente seja liberada de forma segura e ordenada.

Programa 11: TAD main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "matriz.h"
4  #include "palavra.h"
5  #include "ocorrencias.h"
6  #include "resolver.h"
7  #include "coordenada.h"
8
9  int main() {
10      int linhas, colunas;
11
12 //ENTRADA DE DADOS
13      printf("********** CACA PALAVRAS **********\n");
14      printf("Digite as dimensoes da matriz (ex: 5 6): "); // e pedido e
            lido as dimensoes da matriz
15      if (scanf("%d %d", &linhas, &colunas) != 2 || linhas <= 0 || colunas
            <= 0) {
16          fprintf(stderr, "Dimensoes invalidas.\n"); // caso as dimensões
                colocadas seja diferente de 2 ou menor que 0
17          return 1;
18      }
19
20      // cria a matriz dinamicamente
21      TMatriz* matriz = matriz_criar(linhas, colunas);
22      if (!matriz) {
23          fprintf(stderr, "Erro ao criar matriz.\n");
24          return 1;
25      }
```

```
26
27 // Pede ao usuario para preencher a matriz
28     printf("Digite as letras da matriz (%d linhas, %d caracteres por
           linha):\n", linhas, colunas);
29     if (!matriz_preencher(matriz)) {
30         fprintf(stderr, "Erro ao preencher matriz.\n");
31         matriz_apagar(matriz);
32         return 1;
33     }
34
35 // Imprime a matriz digitada (funcao extra colocada para ajudar a
       visualizacao da matriz que o usuario está digitando)
36     printf("\n--- Matriz Digitada ---\n");
37     matriz_imprimir(matriz);
38     printf("-------------------------\n\n");
39
40      // Cria a tabela de ocorrencias para otimizar a busca
41     TOcorrencias ocorrencias;
42
43     if (!ocorrencias_calcular(&ocorrencias, matriz)) {
44         fprintf(stderr, "Erro ao calcular ocorrencias.\n");
45         matriz_apagar(matriz);
46
47         return 1;
48     }
49
50     // variavel para armazenar a quantidade de palavras que o usuario
           queira buscar
51     int num_palavras;
52
53     printf("Digite a quantidade de palavras a serem buscadas: ");
54     if (scanf("%d", &num_palavras) != 1 || num_palavras < 0) {
55         // Verifica se a entrada foi valida (positiva) caso contrario
               encerra o programa com mensagem de erro
56         fprintf(stderr, "Numero de palavras invalido.\n");
57         matriz_apagar(matriz);
58         return 1;
59     }
60
61     if (num_palavras == 0) {
62         // Caso o usuário não queira buscar nenhuma palavra, encerra o
               programa
63         printf("Nenhuma palavra para buscar. \n Encerrando.\n");
64         matriz_apagar(matriz);
65         return 0;
66     }
67
68     // 'TPalavra* palavras' aloca dinamicamente o vetor de palavras que
           serão buscadas
69     TPalavra* palavras = (TPalavra*)malloc(num_palavras * sizeof(TPalavra
           ));
70     if (!palavras) {
71         // se a alocaçao falhar, ira exibir mensagem de erro e encerrar o
               programa
72         fprintf(stderr, "Erro ao alocar memoria para palavras.\n");
73         matriz_apagar(matriz);
74
75         return 1;
```

```
76      }
77
78      // Preenche o vetor de palavras com os dados fornecidos pelo usuário
79      if (!palavras_preencher(palavras, num_palavras)) {
80          // Se ocorrer erro ao preencher, libera memória e encerra o
                programa
81          fprintf(stderr, "Erro ao preencher palavras.\n");
82          free(palavras);
83          matriz_apagar(matriz);
84          return 1;
85      }
86
87      // se matriz_solucionar retornar 0 (falha), imprime mensagem de erro.
88      if (!matriz_solucionar(matriz, &ocorrencias, palavras, num_palavras))
            {
89          fprintf(stderr, "Processo de soluçao do caca palavras concluido.\
                n");
90      }
91
92      printf("\n********** RESULTADO **********\n");
93      // imprime as palavras e suas posiçoes na matriz
94      palavras_imprimir_solucao(palavras, num_palavras);
95
96
97      // Verifica se houve erro ao liberar a memória dos textos das
            palavras e exibe um aviso caso a operação falhe
98      if (!palavras_apagar_todas(palavras, num_palavras)) {
99          fprintf(stderr, "Aviso: Problema ao desalocar textos das palavras
                individuais.\n");
100     }
101     // Libera o vetor de palavras
102     free(palavras);
103
104     // esse if verifica se houve erro ao liberar a memória das ocorrê
            ncias e exibe um aviso caso a operação falhe
105     if (!ocorrencias_apagar(&ocorrencias)) {
106         fprintf(stderr, "Aviso: 'apagar' ocorrencias não teve efeito ou
                falhou.\n");
107     }
108
109     // o if verifica se ocorreu erro ao liberar a memória da matriz e
            exibe um aviso caso a operação falhe
110     if (!matriz_apagar(matriz)) {
111         fprintf(stderr, "Aviso: Problema ao apagar matriz.\n");
112     }
113
114     // Mensagem final indicando término do programa
115     printf("\nCaca palavras finalizado.\n");
116     return 0;
117 }
```

3 Análise de Complexidade dos Algoritmos

Nesta seção, será feita a análise de complexidade de tempo de execução de todas as funções do programa e do programa principal. A análise considerará as seguintes variáveis:

- L: o número de linhas da matriz.
- C: o número de colunas da matriz.

- $N = L \times C$: o número total de células na grade da matriz.
- P: o número de palavras a serem buscadas.
- M_k: o comprimento da k-ésima palavra, e M_{max} o comprimento da maior palavra.
- Occ_{max}: o número máximo de ocorrências de uma única letra na matriz.

Funções do TAD Coordenada (coordenada.c)

`coordenada_criar`: Está função executa apenas um número fixo de operações de atribuição para inicializar os campos da `struct`. Portanto, sua complexidade é constante, $O(1)$.
`coordenada_verificar`: Executa um conjunto fixo de comparações lógicas para verificar se a coordenada está dentro dos limites da matriz. A complexidade também é constante, $O(1)$.

Funções do TAD Matriz (matriz.c)

A implementação da matriz duplamente encadeadas impacta significativamente a complexidade das operações de acesso e inserção, que não são de tempo constante como seriam em um vetor 2D.
`matriz_criar`: A função aloca a estrutura principal e a célula cabeça principal ($O(1)$). Em seguida, executa dois laços independentes: um para criar as C cabeças de coluna e outro para criar as L cabeças de linha. Portanto, a complexidade é a soma dessas operações, resultando em $O(L + C)$.
`matriz_obter_letra`: Para encontrar a letra na posição (l, c), a função primeiro percorre a lista de cabeças de linha até a linha l (pior caso $O(L)$) e depois percorre a lista de dados daquela linha até a coluna c (pior caso $O(C)$). A complexidade total é a soma dessas travessias, sendo $O(L + C)$.
`matriz_inserir_letra`: Similar à `matriz_obter_letra`, a função precisa encontrar a posição correta para inserção, o que envolve percorrer as listas de cabeças e de dados. A complexidade para encontrar o ponto de inserção e ajustar os ponteiros é de $O(L + C)$.
`matriz_preencher`: Esta função possui dois laços aninhados que iteram L e C vezes, totalizando N iterações. Em cada iteração, ela chama `matriz_inserir_letra`, que tem custo $O(L + C)$. Portanto, a complexidade total para preencher a matriz é $O(N \cdot (L + C))$.
`matriz_imprimir` e `matriz_apagar`: Ambas as funções precisam percorrer todas as células de dados e todas as células cabeça para, respectivamente, imprimir ou liberar a memória. O número total de células é $N(dados) + L(cabeçasdelinha) + C(cabeçasdecoluna) + 1(cabeçaprincipal)$. A complexidade é, portanto, proporcional ao número total de nós na estrutura, resultando em $O(N + L + C)$. Como N é o termo dominante, simplifica-se para $O(N)$.

Funções do TAD Ocorrências (ocorrencias.c)

`ocorrencias_inicializar` e `ocorrencias_apagar`: Ambas executam laços sobre o tamanho do alfabeto e/ou `MAX_OCORRENCIAS`, que são valores constantes. Portanto, sua complexidade é $O(1)$.
`ocorrencias_adicionar`: O custo desta função é dominado pela chamada a `matriz_obter_letra`, que, como vimos, é $O(L + C)$. Logo, a complexidade é $O(L + C)$.
`ocorrencias_calcular`: Esta função itera por todas as N posições da matriz e, para cada uma, chama `ocorrencias_adicionar` ($O(L+C)$). Assim como em `matriz_preencher`, a complexidade total é $O(N \cdot (L + C))$.
`ocorrencias_buscar_palavra`: A função realiza um cálculo de índice e um acesso direto a um vetor com base na primeira letra da palavra. Trata-se de uma operação de tempo constante, $O(1)$.

Funções do TAD Palavra e Resolução (palavra.c, resolver.c)

`palavra_criar`: A complexidade depende do tamanho da palavra (M_k) a ser criada, devido às funções `strlen`, `malloc` e ao laço de cópia. Portanto, a complexidade é $O(M_k)$.
`verificar_palavra_em_direcao`: Esta é uma função auxiliar crítica. Ela possui um laço que executa M_k vezes (o comprimento da palavra). Dentro deste laço, ela chama `matriz_obter_letra`, que tem custo $O(L+C)$. Portanto, a complexidade para verificar uma palavra em uma única direção é $O(M_k \cdot (L+C))$.
`palavra_buscar_pos`: Esta função chama `verificar_palavra_em_direcao` um número constante de vezes (8 direções). Portanto, sua complexidade é a mesma da função auxiliar: $O(M_k \cdot (L + C))$.
`matriz_solucionar`: Esta é a função central da busca. Ela possui um laço principal que executa P vezes (para cada palavra). Dentro dele, há um segundo laço que itera sobre as ocorrências da primeira letra

(no pior caso, Occ_{max} vezes). Dentro deste segundo laço, é chamada a função `palavra_buscar_pos`. A complexidade total é, portanto, a soma do custo para cada palavra, resultando em $O(P \cdot Occ_{max} \cdot M_{max} \cdot (L + C))$.

Programa Principal (main.c)

O programa principal chama sequencialmente as funções descritas. Para determinar sua complexidade total, somamos as complexidades das operações mais custosas:

- `matriz_preencher`: $O(N \cdot (L + C))$
- `ocorrencias_calcular`: $O(N \cdot (L + C))$
- `palavras_preencher`: $O(P \cdot M_{max})$
- `matriz_solucionar`: $O(P \cdot Occ_{max} \cdot M_{max} \cdot (L + C))$

A complexidade do programa como um todo é regida pelos termos dominantes. Tanto a preparação da matriz quanto a busca dependem do fator $(L+C)$, que é o custo de acesso na matriz. A complexidade final é a soma dos custos de preparação e de busca:

$$O(N \cdot (L + C) + P \cdot Occ_{max} \cdot M_{max} \cdot (L + C))$$

Fica claro que o custo de acesso da estrutura de dados escolhida ($O(L+C)$) é um fator multiplicativo em quase todas as operações principais, tornando-se o principal gargalo de desempenho em tempo de execução, apesar de sua eficiência em uso de memória.

4 Apresentação e discussão dos resultados

Vários testes foram realizados com o objetivo de verificar o funcionamento correto do programa em diferentes cenários, cobrindo desde casos de uso típicos até situações de borda. Esses testes buscaram validar a lógica de busca, a manipulação da estrutura de dados de matriz esparsa e o gerenciamento de memória.

Os testes foram executados em um ambiente computacional com a seguinte configuração: Processador **AMD Ryzen 5 5600H**, **16 GB de RAM**, e sistema operacional **Windows 11**. O programa foi compilado com o **GCC dentro do Terminal do Visual Studio Code** e apresentou desempenho satisfatório para matrizes de tamanho moderado (teste realizado com exemplo dado para ser testado no trabalho 1, de tamanho 5x6), com as operações de busca sendo concluídas em tempo imperceptível para o usuário.

A seguir, são detalhados os principais casos de teste executados.

Teste 1: Caso de Uso Básico

Este teste valida o funcionamento geral do programa com uma matriz e palavras que cobrem múltiplas direções de busca (horizontal, vertical e diagonal).

- **Configuração:** Utilizou-se uma matriz 5x6 e uma lista de 6 palavras conhecidas.
- **Entrada:** A matriz e as palavras foram inseridas conforme a execução mostrada nas figuras.
- **Saída Esperada:** O programa deveria encontrar as palavras presentes na matriz, reportar corretamente suas coordenadas de início e fim, e indicar quais palavras não foram encontradas.
- **Resultado Observado:** O programa comportou-se como o esperado. A Figura 1 ilustram a interação e o resultado. As palavras foram corretamente localizadas e suas coordenadas exibidas, enquanto as palavras ausentes foram devidamente sinalizadas.

Figura 1: Entrada de dados para o teste básico: dimensões, conteúdo da matriz e palavras a serem buscadas. Logo Após sai o resultado, que é: A matriz "desenhada após digitar as letras; e o "RESULTADO FINAL"que é a palavra e a sua esquerda a sua coordenada.

```
PS C:\Users\caiom\Desktop\tp2\tp2> ./jogo.exe
********** CACA PALAVRAS **********
Digite as dimensoes da matriz (ex: 5 6): 5 6
Digite as letras da matriz (5 linhas, 6 caracteres por linha):
Linha 1: sccelr
Linha 2: uehayr
Linha 3: geaimc
Linha 4: cavrea
Linha 5: cxepla

--- Matriz Digitada ---
s c c e l r
u e h a y r
g e a i m c
c a v r e a
c x e p l a
----------------------

Digite a quantidade de palavras a serem buscadas: 6
Digite as 6 palavras (uma por linha):
chave
mel
erva
veu
uva
cama

********** RESULTADO **********
0 2 4 2 chave
2 4 4 4 mel
3 4 3 1 erva
3 2 1 0 veu
0 0 0 0 uva
0 2 3 5 cama

Caca palavras finalizado.
```

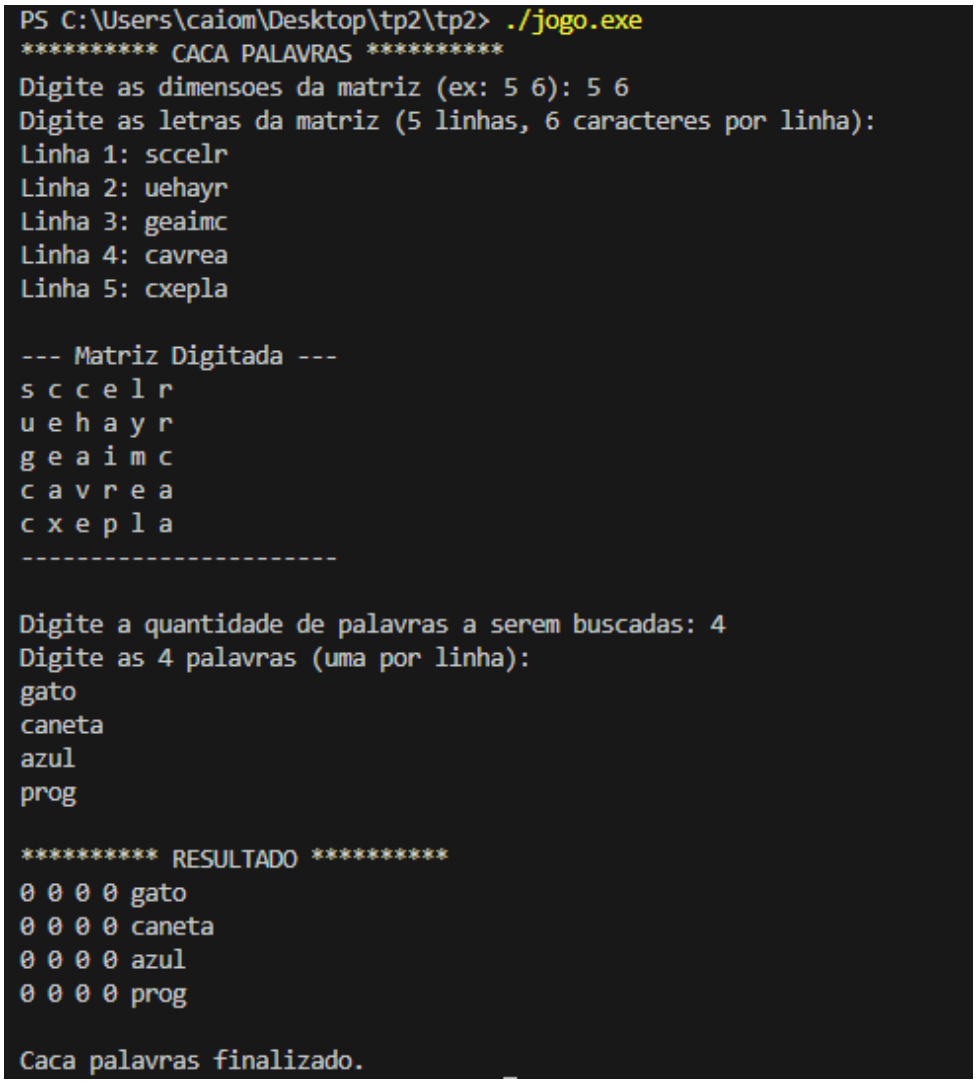
Teste 2: Palavras Inexistentes

O objetivo deste teste foi verificar se o programa lida corretamente com palavras que não estão na matriz, evitando falsos positivos.

- **Configuração:** Utilizou-se a mesma matriz do teste anterior.
- **Entrada:** Foi fornecida uma lista de palavras que sabidamente não existem na matriz.
- **Saída Esperada:** O programa deveria reportar que nenhuma das palavras foi encontrada, exibindo as coordenadas "0 0 0 0"para cada uma.

- **Resultado Observado:** Conforme esperado, o programa indicou que todas as palavras da lista não foram localizadas, validando a corretude da lógica de busca para casos de falha (Figura 2).

Figura 2: Resultado para a busca de palavras inexistentes dentro do caça palavras.



5 Conclusão

Discussão Geral dos Resultados

Os resultados obtidos nos testes confirmam que a implementação do caça-palavras está funcional para os cenários propostos. A modularização em Tipos Abstratos de Dados (TADs) provou ser eficaz, permitindo um desenvolvimento organizado e facilitando a depuração.

A escolha de uma matriz com listas duplamente encadeadas foi o ponto central do trabalho. Esta estrutura demonstrou sua principal vantagem teórica: a eficiência no uso de memória, pois apenas as células preenchidas são alocadas. Em um cenário com uma matriz muito grande e poucas letras, essa economia seria substancial.

No entanto, a análise de complexidade e a implementação revelaram o contraponto dessa escolha. O custo para acessar ou inserir um elemento, sendo $O(L + C)$, é significativamente maior do que o custo $O(1)$ de um vetor bidimensional. Esse fator impactou a complexidade de funções cruciais como `matriz_preencher`, `ocorrencias_calcular` e, principalmente, a busca de palavras, tornando-as mais lentas em teoria.

Identificou-se também uma limitação na otimização da busca: a constante `MAX_OCORRENCIAS`. Embora acelere a busca ao limitar o número de pontos de partida, ela poderia, em um caso extremo com uma

letra muito frequente, impedir que uma palavra seja encontrada se sua ocorrência correta estiver além desse limite.

Em suma, o trabalho cumpre seus objetivos e serve como uma excelente demonstração prática do *trade-off* (relação de compromisso) entre eficiência de memória e complexidade de tempo de execução, um conceito fundamental em estruturas de dados. Dentre as principais dificuldades encontradas, destaca-se o desafio inicial de migrar a estrutura de dados de um simples vetor bidimensional para a matriz com listas duplamente encadeadas. A princípio, houve uma barreira para consolidar o entendimento teórico sobre a lógica dos ponteiros e a dinâmica da estrutura. Posteriormente, mesmo após a compreensão conceitual obtida em sala de aula, a aplicação prática desse conhecimento em código se mostrou complexa. Essa dificuldade foi amplificada pela escassez de materiais de apoio e exemplos claros na internet sobre a implementação específica de uma matriz com esta abordagem. A substituição da estrutura não foi, portanto, uma tarefa direta, exigindo uma refatoração do código anterior, com a remoção de funções e a reescrita de certas etapas para que a nova matriz pudesse ser integrada de forma clara e funcional. Superar essa fase inicial foi, sem dúvida, a etapa mais desgastante, porém a mais valiosa para o aprendizado.

Referências

[1] CFBCursos. Aula 78 - struct em c (parte 1). https://www.youtube.com/watch?v=CAHSAiqHOj8, 2016. Acesso em: 11 jun. 2025.

[2] Curso em Vídeo. Como usar struct em c. https://www.youtube.com/watch?v=C8Fg4LkfTgA, 2021. Acesso em: 11 jun. 2025.

[3] Paulo Feofiloff. Listas encadeadas. https://www.ime.usp.br/ pf/algoritmos/aulas/lista.html, 2020. Acesso em: 19 jul. 2025.

[4] Luiz Otávio Lima. Struct em c – aula prática de struct na linguagem c. https://www.youtube.com/watch?v=Mook-z2uvpMt=414s, 2022. Acesso em: 11 jun. 2025.

[5] David Menoti. *Programação em C Um curso básico e abrangente*. Belo Horizonte, 1st edition, 2005.

[6] Pedro Pereira. Balancing trade-offs in machine learning algorithms: A contextual approach. https://medium.com/@pedrorp/balancing-trade-offs-in-machine-learning-algorithms-a-contextual-approach-7a4c382846a3, 2023. Acesso em: 18 jul. 2025.

[7] Programação Descomplicada | Linguagem C. Lista dinâmica duplamente encadeada | estrutura de dado em c. https://www.youtube.com/watch?v=4VoGEH0jnps, 2021. Acesso em: 12 jul. 2025.

[8] Simplicode. Struct em c – entenda de forma rápida e prática! https://www.youtube.com/watch?v=AfaT0ARp1TI, 2023. Acesso em: 11 jun. 2025.