RELATÓRIO: EXTRAÇÃO DE PALÍNDROMOS EM STRINGS

Trabalho final da disciplina de Algoritmos II

Emanuel Bebber
Universidade Tecnológica Federal do Paraná
Pato Branco, Brasil
emanuelb@alunos.utfpr.edu.br

Fernando Brocco
Universidade Tecnológica Federal do Paraná
Pato Branco, Paraná
fernandobrocco@alunos.utfpr.edu.br

Guilherme Santos Reis
Universidade Tecnológica Federal do Paraná
Pato Branco, Paraná
guilhermereis.2003@alunos.utfpr.edu.br

Otávio Augusto Viccini Bender
Universidade Tecnológica Federal do Paraná
Pato Branco, Paraná
otaviobender@alunos.utfpr.edu.br

Pedro Augusto Merisio
Universidade Tecnológica Federal do Paraná
Pato Branco, Paraná
merisio@alunos.utfpr.edu.br

Resumo: Este relatório apresenta os resultados da implementação de um algoritmo na linguagem C, com a ideia de identificar palavras palíndromo em uma dada string. O programa emprega o método guloso, primeiro identificando todos os palíndromos possíveis e depois otimizando a lista, removendo strings sobrepostas. Para isso, são implementadas diversas funções visando obter uma solução ótima para a suposta problemática.

Palavras-chave: método guloso, algoritmo, palíndromo, strings;

1. INTRODUÇÃO

Os Palíndromos são sequências de caracteres que têm aplicações em diversos campos da computação, pois podem ser lidos da mesma forma no sentido direto e inverso.

Um dos motivos para que a extração de palíndromos em strings fosse o tema escolhido pelo grupo, foi o fato de que, ao realizar uma breve pesquisa, não foram encontrados muitos materiais a respeito. Além disso, também havia o interesse mútuo do grupo em implementar um código fazendo uso de arranjos de caracteres (strings).

Assim sendo, o objetivo final é implementar um algoritmo eficiente para o nosso caso, utilizando do método guloso, para que o programa receba uma string e consiga extrair os palíndromos presentes nela. Dessa forma, o código deve retornar o número mínimo de cortes para separar estes palíndromos.

2. DESCRIÇÃO DO PROBLEMA E MOTIVAÇÕES

2.1 Definição de palíndromo

Palíndromos são palavras, frases, números ou outras sequências de caracteres que podem ser lidas da mesma forma, independentemente se são lidas da esquerda para direita ou da direita para esquerda. Eles ignoram os espaços, a pontuação e as diferenças entre maiúsculas e minúsculas (Marinho, 1999).

2.3 Problema Apresentado

O problema apresentado foi a extração de palíndromos em arranjos de caracteres (strings). Dada uma string str, na qual deve ser encontrada substrings palíndromo. O algoritmo deve encontrar o número mínimo de cortes na string, de forma que separe os palíndromos.

2.4 Estratégia Escolhida

O método guloso, escolhido para a resolução da problemática, foi definido por uma série de fatores como: simplicidade de implementação e resolução de problemas apropriados, tópico abordado por Cormen em seu livro: “Um algoritmo guloso sempre faz a escolha que parece ser a melhor no momento em questão. Isto é, faz uma escolha localmente ótima, na esperança de que essa escolha leve a uma solução globalmente ótima”. Além disso, esse método também visa maximizar o comprimento dos palíndromos encontrados.

3. DESCRIÇÃO DAS SOLUÇÕES E FUNÇÕES

O programa consiste em diversas funções projetadas para lidar com strings e palíndromos:

“EhPalindromo”: Esta função recebe como parâmetro três variáveis, sendo elas um ponteiro de char nomeado como “str”, e duas variáveis int que limitam o início e o final da subpalavra dessa string.

O código dentro da função recebe esses parâmetros e verifica se do início ao final dessa substring é um palíndromo, se sim retorna 1 senão retorna 0.

```c
int EhPalindromo(char *str, int in, int fin) {
    while (in < fin) {
        if (str[in] != str[fin]) {
            return 0;
        }
        in++;
        fin--;
    }
    return 1;
}
```

Figura 1. Autoria própria,2024

“ordenar_posicoes”: Esta função recebe como parâmetro uma matriz de inteiros de duas dimensões nomeada como "pos", onde cada posição da matriz armazena os valores de início e fim de cada palíndromo encontrado , e um inteiro "n" que representa o número de segmentos.

O código dentro da função opera sobre esses segmentos e os ordena em ordem decrescente com base no comprimento de cada segmento. Se o comprimento do segmento na posição "j" for maior que o comprimento do segmento na posição "i", os segmentos trocam de posição. O comprimento de um segmento é calculado subtraindo o valor de início do valor de fim.

```c
void ordenar_posicoes(int pos[][30], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = i+1; j < n; j++) {
            int len_i = pos[1][i] - pos[0][i];
            int len_j = pos[1][j] - pos[0][j];
            if (len_j > len_i) {
                int temp_start = pos[0][i];
                int temp_end = pos[1][i];
                pos[0][i] = pos[0][j];
                pos[1][i] = pos[1][j];
                pos[0][j] = temp_start;
                pos[1][j] = temp_end;
            }
        }
    }
}
```

Figura 2. Autoria própria,2024

“achar_todos_palindromos”: Esta função recebe como parâmetros um ponteiro de char nomeado "str" e um inteiro "tamanho" representando o tamanho da string. A função identifica todos os palíndromos na string e os armazena em uma matriz "pos". Em seguida, ela ordena os palíndromos por tamanho, remove subpalíndromos, reordena os palíndromos restantes pela posição inicial e imprime a string original com marcadores "|" nas posições de início e fim de cada palíndromo. Podemos dividir em pedaços para melhor entendimento:

“Identificação de palíndromos”: A função percorre todos os possíveis substrings de "str" e verifica se são palíndromos usando a função "EhPalindromo". Se um substring for um palíndromo, suas posições de início e fim são armazenadas na matriz "pos".

```c
void achar_todos_palindromos(char *str, int tamanho) {
    int Tamanho = strlen(str);
    int pos[2][30];
    int i = 0;
    //Identificar palindromo
    for (int start = 0; start < Tamanho; start++) {
        for (int end = start; end < Tamanho; end++) {
            if (EhPalindromo(str, start, end) && (end - start + 1 > 1)) {
                char pala[101];
                strncpy(pala, str + start, end - start + 1);
                pos[0][i] = start;
                pos[1][i] = end;
                i++;
                pala[end - start + 1] = '\0';

            }
        }
    }
```

Figura 3. Autoria própria,2024

“ordenar_posicoes(pos, i)”: Esta chamada de função ordena os palíndromos encontrados em ordem decrescente de tamanho, onde "i" é o número de palíndromos.

“Remover Sub Palíndromos”: Este trecho remove palíndromos que estão contidos em outros maiores. Ele marca os sub-palíndromos com valores de -1 na matriz "pos".

```c
ordenar_posicoes(pos, i); // ordenar do maior para o menor palindromo
// Remover subpalíndromos
    for (int j = 0; j < i; j++) { // removendo os que fazem parte de um maior e adicionando -1 no lugar
        for (int k = j + 1; k < i; k++) {
            if ((pos[0][k] <= pos[1][j]) && (pos[0][j] <= pos[1][k]) ) {
                pos[0][k] = -1;  // Marcar como removido
                pos[1][k] = -1;
            }
        }
    }
```

Figura 4. Autoria própria,2024

“Remover os -1 da matriz”: Este bloco de código reorganiza a matriz "pos" para remover os palíndromos marcados com -1, ajustando o valor de "i" para o novo número de palíndromos.

```c
    int new_i = 0; // remover os -1 da matriz
    for (int j = 0; j < i; j++) {
        if (pos[0][j] != -1) {
            pos[0][new_i] = pos[0][j];
            pos[1][new_i] = pos[1][j];
            new_i++;
        }
    }
    pos[0][new_i +1] = -1; // adicionado mais uma linha para não imprimir o ultimo | no final da string
    pos[1][new_i +1] = -1; // adicionado mais uma linha para não imprimir o ultimo | no final da string
    i = new_i;
```

Figura 5. Autoria própria,2024

“Ordenar a matriz”: Este bloco de código imprime o número de cortes mínimo necessário e ordena os palíndromos restantes na matriz "pos" em ordem crescente com base na posição de início.

```c
    printf("o numero de cortes minimos necessarios: %d\n", i);
    for (int l = 0; l < i - 1; l++) { // ordenando a matriz
        for (int j = l + 1; j < i; j++) {
            if (pos[0][l] > pos[0][j]) {
                int temp_start = pos[0][l];
                pos[0][l] = pos[0][j];
                pos[0][j] = temp_start;
                int temp_end = pos[1][l];
                pos[1][l] = pos[1][j];
                pos[1][j] = temp_end;
            }
        }
    }
```

Figura 6. Autoria própria,2024

“IMPRIMIR”: Este bloco de código imprime a string original "str" com marcadores "|". Os marcadores são colocados nas posições de início e fim dos palíndromos armazenados na matriz "pos".

```c
    //IMPRIMIR
    int n=0;
    int posiin=0;
    int posifin=0;
    while(n < tamanho){ //imprimindo com | nas posições indicadas que está cada palindromo no codigo. presente na matriz.
        if((pos[0][posiin]==n) && (pos[0][posiin]!=0)){
            printf(" | %c", str[n]);
            posiin++;
        }else if(pos[0][posiin]==0){
            printf("%c", str[n]);
            posiin++;
        }else if((pos[1][posifin] == n) && (pos[1][posifin +1] != -1) && ((pos[1][posifin] - pos[0][posiin]) != -1)){
            printf("%c | ", str[n]);
            posifin++;
        }else if ((pos[1][posifin] - pos[0][posiin]) == -1){
            printf("%c", str[n]);
            posifin++;
        }else{
            printf("%c", str[n]);
        }
        n++;
    }
```

Figura 7. Autoria própria,2024

4. ANÁLISE DE COMPLEXIDADE DE ESPAÇO E TEMPO DE CADA FUNÇÃO

4.1 Função ‘EhPalindromo’

Complexidade de tempo: O(n), onde n é o tamanho da substring sendo verificada como palíndromo.

Complexidade de espaço: O(1), pois não há alocação de memória adicional.

4.2 Função ‘ordenar_posicoes’

Complexidade de tempo: O(n^2), onde n é o número de palíndromos encontrados.

Complexidade de espaço: O(1), pois não há alocação de memória adicional

4.3 Função ‘achar_todos_palindromos’

Complexidade de tempo:

- O(n^3) para encontrar todos os palíndromos, onde n é o tamanho da string de entrada
- O(n^2) para ordenar os palíndromos encontrados.
- O(n^2) para remover sub-palíndromos.
- Total: O(n^3) + O(n^2) + O(n^2) = O(n^3)

Complexidade de espaço: O(n), pois é necessário armazenar as posições dos palíndromos encontrados.

4.4 Função ‘main’

Complexidade de tempo: O(n^3), pois chama a função achar todos palíndromos que tem complexidade tempo O(n^3).

Complexidade de espaço: O(n), pois é necessário armazenar a string de entrada e as posições dos palíndromos encontrados.

4.5 Complexidade geral

Com base no tópico supracitado pode-se ver que, no código, a complexidade total de tempo é O(n^3) e a complexidade total de espaço é O(n).

5. CONCLUSÃO

Neste trabalho, implementamos e discutimos um algoritmo eficiente para a identificação de palíndromos em strings usando técnicas de programação gulosa.

É notável que a utilização de algoritmos gulosos para aplicações práticas e de médio porte, assim como a nossa solução oferece uma boa combinação entre simplicidade de implementação e desempenho satisfatório. No entanto, a complexidade cúbica pode ser uma limitação para strings muito longas, para essas aplicações que requerem um desempenho ainda melhor, outras técnicas, como o algoritmo de Manacher, podem ser consideradas.

Em vista do exposto, esta implementação prática de um algoritmo de identificação de palíndromos também destacou a importância de escolher a estratégia certa para resolver problemas computacionais, alinhando teoria e prática de forma eficaz.

6. REFERÊNCIAS

ROCHA, Anderson. DORINI, Leysa. Unicamp. (2004). **Algoritmos gulosos: definições e aplicações.**

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., & STEIN, C. (2009). **Introduction to Algorithms** (3rd ed.). MIT Press.

MARINHO, R. **Você sabe o que palíndromo?** biblioteca.cl.df.gov.br, 1999.

OLIVA, Jefferson (2024). **material_grad,** Disponível em: https://github.com/jefferson-oliva/material_grad

7. DECLARAÇÃO DE AUTORIA

Declaramos para os devidos fins que este relatório é de nossa autoria. Todas as fontes utilizadas estão citadas no corpo do texto e nas referências.

Todos os autores deste trabalho tiveram papel importante na resolução da problemática e na implementação da solução. Analogamente, foram responsáveis pelo desenvolvimento do código de forma satisfatória. Apesar disso, algumas áreas foram designadas, com a finalidade de otimizar o trabalho.

Emanuel Bebber: Responsável pela qualidade do código e garantia de eficiência do algoritmo.

Fernando Brocco: Responsável pelo desenvolvimento do relatório, reunião dos membros e participação no desenvolvimento do algoritmo, atuando na execução do mesmo.

Guilherme Santos Reis: Encarregado pela documentação (‘readme’ e relatório) e participação no desenvolvimento do algoritmo.

Otávio Augusto Viccini Bender: Responsável pela documentação e participação no desenvolvimento do algoritmo.

Pedro Augusto Merisio: Encarregado da tarefa de formatação do relatório e da apresentação. Líder da equipe.