Relatório: Projeto II - identificação de prefixos e indexação de dicionários

Juliana Miranda Bosio, Lucas Furlanetto Pascoali

1 Introdução

O programa utiliza a estrutura **Trie** (conhecida como árvore de prefixos) para a indexação e recuperação eficiente de palavras em grandes arquivos de dicionários.

Dado o arquivo dicionário de entrada seguido de uma sequência de palavras contidas neste, a saída esperada seria, para cada palavra da sequência, sua posição no arquivo e determinar a existência de prefixos.

2 Primeiro problema: identificação de prefixos

A classe `Trie` é uma estrutura de árvore contituída por nós da classe `NoTrie`, cada um destes possui os atributos: "word"(o caractere contido no nó), "child"(um vetor fixo de 26 nós), "position"(posição dela no arquivo), "length"(comprimento, se maior que zero, indica último caracter de uma palavra) e "counter"(conta quantas vezes o nó foi visitado ao inserir prefixos na estrutura principal). A Trie implementada possui apenas duas funções: `add` e `find prefix`

2.1 Lógica das Funções da Trie

A função `add` insere uma palavra na estrutura de `trie`, caractere por caractere. Para cada caractere da palavra, calcula-se seu índice no vetor de filhos e verifica-se se o nó correspondente já existe. Caso contrário, cria-se um novo nó. Durante o processo, o contador (counter) de cada nó visitado é incrementado, indicando quantas palavras passam por aquele caractere. No final, o último nó da palavra recebe informações sobre sua posição e comprimento, completando a inserção.

A função `find prefix` verifica se um prefixo está presente na trie. A partir da raiz, percorre os caracteres do prefixo, navegando pela estrutura de nós de acordo com os índices calculados. Caso encontre um caractere cujo nó correspondente não exista, retorna nullptr, indicando que o prefixo não está na trie. Se todos os caracteres forem encontrados, retorna o nó correspondente ao último caractere do prefixo, permitindo acessar informações como o número de palavras que compartilham esse prefixo.

2.2 Implementação

```
void add(std::string word, unsigned long position, unsigned long length)
{
    NoTrie *current = root;
    for (char c : word) {
        int index = c - 'a';
        if (current->child[index] == nullptr) {
            current->child[index] = new NoTrie(c);
        }
        current->child[index]->counter++;
        current = current->child[index];
    }
    current->position = position;
    current->length = length;
}

NoTrie *find_prefix(std::string prefix) {
    NoTrie *current = root;
    for (char c: prefix) {
        int index = c - 'a';
        if (current->child[index] == nullptr) return nullptr; // Char not
        found
        current = current->child[index];
    }
    return current;
}
```

2.3 Figura

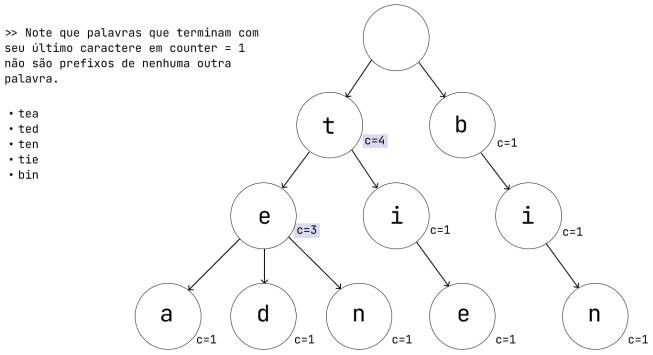

Figura 1: Exemplo figurativo do uso do counter em uma Trie

3 Segundo problema: indexação de arquivo de dicionário

O segundo problema envolve a indexação de um arquivo de dicionário, associando à palavra sua localização e o tamanho da linha no arquivo. Assim, é possível determinar não apenas a presença de um prefixo ou palavra no dicionário, mas também retornar informações detalhadas de sua localização. Para isso, utilizamos a função `populate trie` e por fim, na `main`, após popularmos a trie, utilizamos a função `find prefix`.

3.1 Lógica da Função

A função populate trie é responsável por carregar o dicionário em uma trie a partir de um arquivo. Ela utiliza o nome do arquivo como entrada e percorre linha por linha do arquivo. Cada palavra é extraída entre colchetes ([palavra]), e o método `add` da trie é chamado para inseri-la. Durante o processo, a função também rastreia a posição inicial da linha no arquivo (índice absoluto) e o comprimento da linha (sem o caractere de nova linha). Essas informações são armazenadas nos nós da trie correspondentes ao último caractere de cada palavra, permitindo a indexação precisa do dicionário.

No `main`, o programa inicia solicitando ao usuário o nome do arquivo de dicionário e carrega suas palavras na trie utilizando a função `populate trie`. Em seguida, entra em um loop que recebe palavras quaisquer até que "0"seja digitado. Para cada palavra, utiliza-se o método `find prefix` para buscar o prefixo correspondente na trie. Se o prefixo não for encontrado, uma mensagem é exibida indicando que a palavra não é um prefixo. Caso contrário, é retornada a quantidade de palavras que compartilham o prefixo. Adicionalmente, se a palavra for também encontrada como entrada válida no dicionário (isto é, seu nó final contém informações de posição e comprimento), a posição e o tamanho são exibidos. Assim, é possível diferenciar palavras que são prefixos de outras e palavras que existem no dicionário com suas respectivas localizações.

3.2 Figura

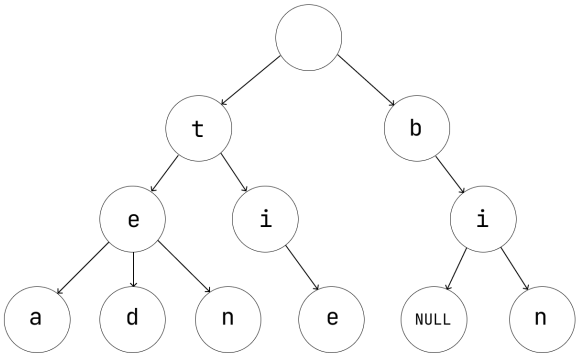

Figura 2: Exemplo figurativo da busca pelo prefixo "bit"(não encontrado)

3.3 Implementação

```
void populate_trie(const std::string& filename, Trie *trie) {
    std::ifstream file(filename);
    if (!file.is_open()) {
        std::cerr << "Error: Could not open file " << filename <<
        std::endl;
        return;
    }

    unsigned long position_counter = 0;
    std::string line;
    while (std::getline(file, line)) {
        std::string word = line.substr(1, line.find_first_of(']') - 1);

        trie->add(word, position_counter, line.size());
        position_counter += line.size() + 1;
    }
}
```

4 Conclusão

Em resumo, este trabalho foi fundamental para o aprendizado e a aplicação prática das estruturas de dados apresentadas em sala de aula. Enfrentamos algumas dificuldades ao longo do processo, mas, ao superá-las, conseguimos aprofundar nosso entendimento tanto dos conceitos da matéria quanto das práticas em C++. A seguir, apresentamos as principais dificuldades encontradas e como foram resolvidas.

- **Entendimento da estrutura Trie**: Uma das principais dificuldades foi compreender o funcionamento da estrutura de dados Trie, sua organização hierárquica de caracteres e a navegação por seus nós. Inicialmente, houve desafios em visualizar como os nós eram conectados e como as operações de inserção e busca interagiam com os filhos de cada nó. Para superar esse obstáculo, recorremos a materiais de apoio, como artigos, implementações simples em outras linguagens.

- **Leitura de arquivos no formato .dic**: Outro desafio enfrentado foi realizar a leitura de arquivos no formato .dic em C++. Apesar de termos experiência básica com manipulação de arquivos, interpretar o arquivo linha por linha e extrair as palavras delimitadas por colchetes exigiu um esforço adicional. Foi necessário utilizar as bibliotecas padrão do C++, como `¡fstream¿`, para lidar com a entrada de dados, e construir expressões que manipulassem strings para isolar as palavras de forma precisa. Além disso, a necessidade de armazenar a posição e o comprimento das linhas no arquivo demandou a implementação de uma lógica adicional para contar corretamente os caracteres e evitar erros na indexação.

5 Referências

Para a resolução do problema, foram utilizadas as seguintes referências:

ime.usp.br: **Tries (árvores digitais)**.
Em: `https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/tries.html`.
Acesso em: 14 dez. 2024.

towardsdatascience.com: **Implementing a Trie in Python (in less than 100 lines of code)**.
Em: `https://towardsdatascience.com/implementing-a-trie-data-structure-in-python-in-less-than-100-lines-of-code-a877ea23c1a1`.
Acesso em: 14 dez. 2024.

Youtube/@Lukas Vyhnalek: **Trie Data Structure (EXPLAINED)**
Em: `https://www.youtube.com/watch?v=-urNrIAQnNo&t=417s`
Acesso em: 14 dez. 2024.