Relatório

Projeto II

Identificação de Prefixos e Indexação de Dicionários

Alunos: Jessica Regina dos Santos, Myllena da Conceição Correa e Victor Henrique Labes de Figueiredo

Disciplina: Estrutura de Dados

Semestre: 2023/2

Resumo

Nosso projeto surgiu da necessidade de criar uma solução eficiente para indexação e recuperação de palavras em grandes arquivos de dicionários. Para atender a esse propósito, optamos pela implementação de uma estrutura de dados trie, uma árvore de prefixos que permite representar e buscar palavras de forma otimizada.

A aplicação tem como objetivos principais resolver dois desafios específicos: identificar prefixos de palavras e realizar a indexação de um arquivo de dicionário. A primeira tarefa consiste em construir uma trie em memória principal a partir das palavras contidas no dicionário. Posteriormente, a aplicação recebe uma série de palavras, pertencentes ou não ao dicionário, e determina se cada uma delas é um prefixo de outras palavras presentes no dicionário. A segunda tarefa envolve a construção da trie considerando a localização da palavra no arquivo e o tamanho da linha que a define.

Neste relatório, detalharemos as soluções adotadas, apresentando a lógica por trás dos algoritmos implementados. Serão discutidos os aspectos cruciais da estrutura da trie, os

desafios enfrentados durante o desenvolvimento, e figuras serão fornecidas para ilustrar a estrutura da trie.

Este projeto não apenas proporciona uma solução técnica, mas também destaca a importância da escolha de estruturas de dados adequadas para otimizar operações específicas.

Desenvolvimento

- Descrição da Solução

Nosso código implementa um programa em C++ que utiliza a estrutura de dados trie para resolver dois problemas específicos: identificação de prefixos e indexação de um arquivo de dicionário. Vamos explorar cada parte do código para entender como ele aborda esses desafios.

- Trie: Estrutura Hierárquica

Uma trie, ou árvore de prefixos, é representada por uma classe 'Trie' que contém uma classe interna 'TrieNode'. Cada nó ('TrieNode') na trie possui as seguintes características:

- nodeChar: Caractere representativo do nó.
- valid: Indicador se o nó representa uma palavra válida.
- children: Vetor de ponteiros para os filhos do nó, representando as letras subsequentes da palavra.
- position: Posição da palavra no arquivo de dicionário.
- length: Comprimento da linha que define a palavra no arquivo.
- prefix: Contador do número de palavras que compartilham o mesmo prefixo até este nó.

```
 8  // TrieNode representa um nó na estrutura de dados trie
 9  class TrieNode {
10  public:
11      explicit TrieNode(char c) : nodeChar(c), valid(false), position(0), length(0), prefix(0) {}
12
13      char nodeChar;
14      bool valid;
15      std::vector<TrieNode*> children;
16      unsigned long position;
17      unsigned long length;
18      unsigned long prefix;
19  };
20
21  // Trie representa a estrutura de dados trie
22  class Trie {
23  public:
24      Trie() {
25          root = new TrieNode('\0');
26      }
27
28      ~Trie() {
29          deleteRoot(root);
30      }
```

A classe 'Trie' representa a própria trie e mantém um ponteiro para o nó raiz. A classe 'TrieNode' representa um nó na trie. Cada nó armazena as informações listadas acima.

- Funções Principais
- Insert

```
32      // Inserir uma palavra na trie
33      void insert(TrieNode* node, const std::string& word, unsigned long position, unsigned long length) {
```

A função 'insert' é muito importante para a construção da trie. Ela recebe um ponteiro para o nó atual, a palavra a ser inserida, a posição da palavra no arquivo de dicionário e o comprimento da linha que define a palavra. Durante o processo de inserção, a função percorre cada caractere da palavra, atualizando os nós existentes ou criando novos nós conforme necessário. Ao final

da palavra, o nó correspondente é marcado como uma palavra válida, e suas informações de posição e comprimento são atualizadas.

```
34          for (char c : word) {
35              bool foundInChild = false;
36              node->prefix++;
37
38              for (TrieNode* child : node->children) {
39                  if (child->nodeChar == c) {
40                      node = child;
41                      foundInChild = true;
42                      break;
43                  }
44              }
45
46              if (!foundInChild) {
47                  TrieNode* newNode = new TrieNode(c);
48                  node->children.push_back(newNode);
49                  newNode->prefix++;
50                  node = newNode;
51              }
52          }
53
54          node->valid = true;
55          node->position = position;
56          node->length = length;
57      }
```

– Find Prefix

```
59      // Encontrar um prefixo na trie
60      TrieNode* findPrefix(const std::string& word) {
61          TrieNode* node = root;
```

A função ‘findPrefix’ é responsável por percorrer a trie em busca de um prefixo específico. Ela recebe a palavra de entrada e retorna o último nó correspondente ao prefixo encontrado. Durante a busca, para cada caractere na palavra, a função verifica se há um nó correspondente nos filhos do nó atual. Se um caractere não for encontrado, a função retorna *nullptr*. Se a palavra for completamente percorrida, o nó correspondente é retornado.

```
63          for (char c : word) {
64              bool foundInChild = false;
65
66              for (TrieNode* child : node->children) {
67                  if (child->nodeChar == c) {
68                      node = child;
69                      foundInChild = true;
70                      break;
71                  }
72              }
73
74              if (!foundInChild) {
75                  return nullptr;
76              }
77          }
78
79          return node;
80      }
```

– Delete Root

A função ‘deleteRoot’ é uma função privada da classe ‘Trie’ responsável por deletar recursivamente os nós da trie. Ela é utilizada no destrutor da classe ‘Trie’ para liberar corretamente a memória alocada para todos os nós da trie, evitando vazamentos de memória.

```
87  private:
88      TrieNode* root;
89
90      // Deletar recursivamente os nós da trie
91      void deleteRoot(TrieNode* node) {
92          if (node == nullptr) {
93              return;
94          }
95
96          for (TrieNode* child : node->children) {
97              deleteRoot(child);
98          }
99
100         delete node;
101     }
102 };
```

A função recebe como parâmetro um ponteiro para o nó da trie que se deseja deletar. Ela utiliza uma abordagem recursiva para percorrer todos os nós da subárvore, começando pelo nó passado como argumento.

O primeiro passo da função é verificar se o nó passado como parâmetro é nulo (*nullptr*). Se for nulo, a função retorna, encerrando a recursão para este ramo. Em seguida, a função percorre todos os filhos do nó e chama recursivamente a função ‘deleteRoot’ para cada filho. Isso garante que todos os nós da subárvore sejam deletados. Após a deleção recursiva dos filhos, o próprio nó é deletado utilizando o operador delete. Isso libera a memória alocada para este nó específico.

– Main 

```
104 int main(int argc, char* argv[]) {
105     std::string filename;
106     std::string line;
107     std::string dict;
108     std::ifstream file;
109     Trie tree;
```

A função principal (‘main’) coordena a execução geral do programa. Ela abrange as seguintes etapas:

1. Recebimento de Entrada: Verifica se o código está sendo executado fora do ambiente virtual Moodle. Se sim, espera um argumento de linha de comando (nome do arquivo de dicionário); caso contrário, lê o nome do arquivo de dicionário a partir da entrada padrão.

```
111 #ifndef MOODLE
112 // Receber o argumento de entrada
113 if (argc < 2) {
114     std::cout << "Error: No input filename provided.\n";
115     std::cout << "Usage: ./program [name_of_file.dic]\n"
116     return 1;  // Retornar um código de erro
117 }
118 filename = argv[1];
119 #endif
120
121 #ifdef MOODLE
122 std::cin >> filename;
123 #endif
```

2. Leitura do Arquivo de Dicionário: Abre o arquivo de dicionário, lê suas linhas e concatena os conteúdos em uma única *string* (‘dict’).

```
125 // Abrir arquivo
126 file.open(filename);
127
128 if (!file.is_open()) {
129     std::cout << "Error: Unable to open the file.\n";
130     return 1;  // Retornar um código de erro
131 }
132
133 // Ler palavras do dicionário do arquivo
134 while (getline(file, line)) {
135     dict += line + "\n";
136 }
137
138 file.close();   // Fechar arquivo
```

3. Extração de Palavras, Posições e Comprimentos: Utiliza um loop para extrair as palavras do dicionário, suas posições e comprimentos, inserindo essas informações em vetores separados.

```
140     // Extrair palavras do dicionário, posições e comprimentos
141     size_t offset = 0;
142     std::vector<std::string> dictWords;
143     std::vector<unsigned long> positions;
144     std::vector<unsigned long> lengths;
145
146     while (true) {
147         size_t begin = dict.find('[', offset);
148         size_t end = dict.find(']', offset);
149
150         if (begin == std::string::npos) {
151             break;
152         }
153
154         std::string word = dict.substr(begin + 1, end - begin - 1);
155
156         if (dict.find('\n', offset) == std::string::npos) {
157             offset = dict.length() + 1;
158         } else {
159             offset = dict.find('\n', offset) + 1;
160         }
161
162         dictWords.push_back(word);
163         positions.push_back(begin);
164         lengths.push_back(offset - begin - 1);
165     }
```

4. Inserir Palavras do Dicionário na Trie: Utiliza um loop for para iterar sobre todas as palavras do dicionário, representadas pelo vetor ‘dictWords’. Para cada palavra do dicionário, chama a função ‘insert’ da instância da classe ‘Trie’ (representada pela variável ‘tree’). A função insert recebe quatro parâmetros: node, word, position e length. A função ‘insert’ percorre a palavra caractere por caractere e a insere na trie. A cada iteração, ela cria novos nós conforme necessário e atualiza as informações relevantes no nó final da palavra.

```
167     // Inserir palavras do dicionário na trie
168     for (size_t i = 0; i < dictWords.size(); i++) {
169         tree.insert(tree.get_root(), dictWords[i], positions[i], lengths[i]);
170     }
```

5. Ler palavra de entrada: É criado um vetor ‘inputWords’ para armazenar palavras fornecidas pelo usuário. O programa entra em um loop infinito, onde cada iteração lê uma palavra da entrada padrão (‘std::cin’) e a armazena na variável ‘inputWord’. Se a palavra lida for "0", o loop é interrompido. Caso contrário, a palavra é adicionada ao vetor ‘inputWords’. Esse processo permite a leitura iterativa de palavras até que o usuário indique o término com o marcador "0", possibilitando o processamento posterior das palavras armazenadas no vetor.

```
172     std::vector<std::string> inputWords;
173     // Ler palavras de entrada
174     while (true) {
175         std::string inputWord;
176         std::cin >> inputWord;
177
178         if (inputWord == "0") {
179             break;
180         }
181
182         inputWords.push_back(inputWord);
183     }
```

6. Processar palavras de entrada e encontrar prefixos na trie: No fim do código, as palavras de entrada fornecidas pelo usuário são processadas por meio de um loop que itera sobre o vetor ‘inputWords’. Para cada palavra, a função ‘findPrefix’ da classe ‘Trie’ é utilizada para buscar o nó correspondente na trie construída a partir do dicionário. Se o nó existe, indicando que a palavra é um prefixo na trie, o programa determina o número de palavras com o mesmo prefixo e exibe essa informação. Além disso, se a palavra completa estiver presente na trie, são exibidos a posição inicial e o comprimento dessa palavra. Caso o prefixo não seja encontrado na trie, uma mensagem indicando que a palavra não é um prefixo é exibida. Essa parte do código é importante para identificar prefixos, contabilizar a quantidade de palavras com o mesmo prefixo e exibir informações quando a palavra está presente na trie.

```
185     // Processar palavras de entrada e encontrar prefixos na trie
186     for (const std::string& word : inputWords) {
187         TrieNode* tmp = tree.findPrefix(word);
188 
189         if (tmp) {
190             int tmp_prefix = tmp->valid ? tmp->prefix : tmp->prefix - 1;
191             std::cout << word << " is prefix of " << tmp_prefix << " words" << std::endl;
192 
193             if (tmp->length > 1) {
194                 std::cout << word << " is at (" << tmp->position << "," << tmp->length << ")" << std::endl;
195             }
196         } else {
197             std::cout << word << " is not prefix" << std::endl;
198         }
199     }
200 
201     return 0;
202 }
```

Figuras

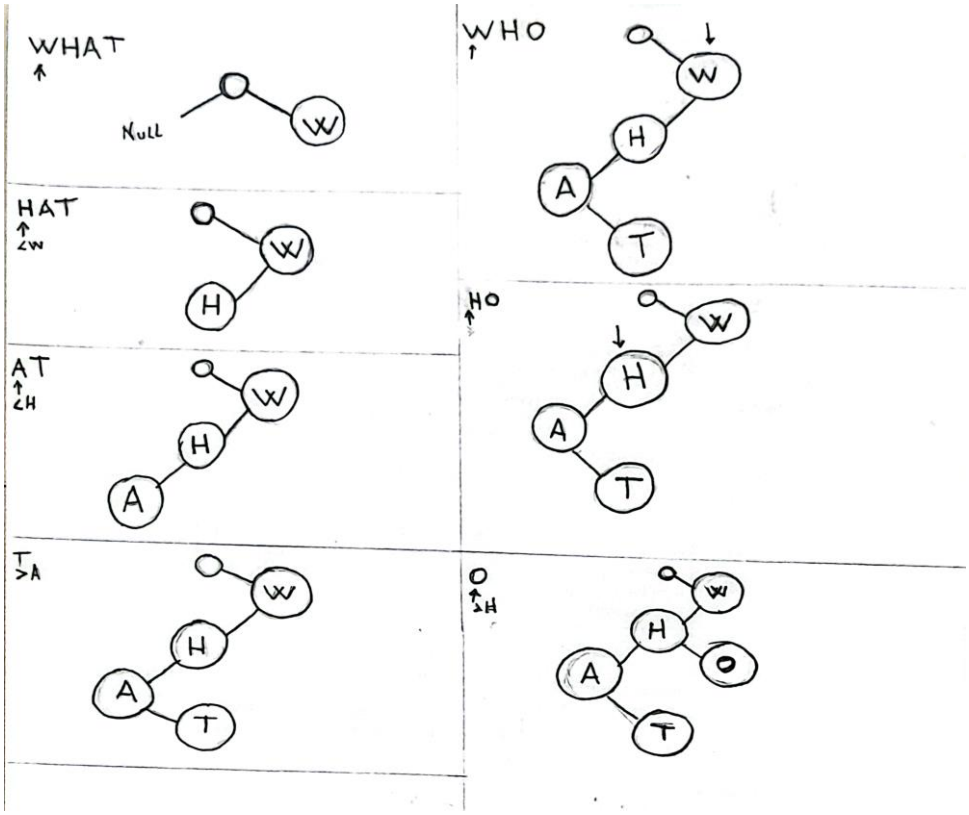

Figura 1. Exemplo do funcionamento da função 'insert' para as palavras "WHAT" and "WHO"

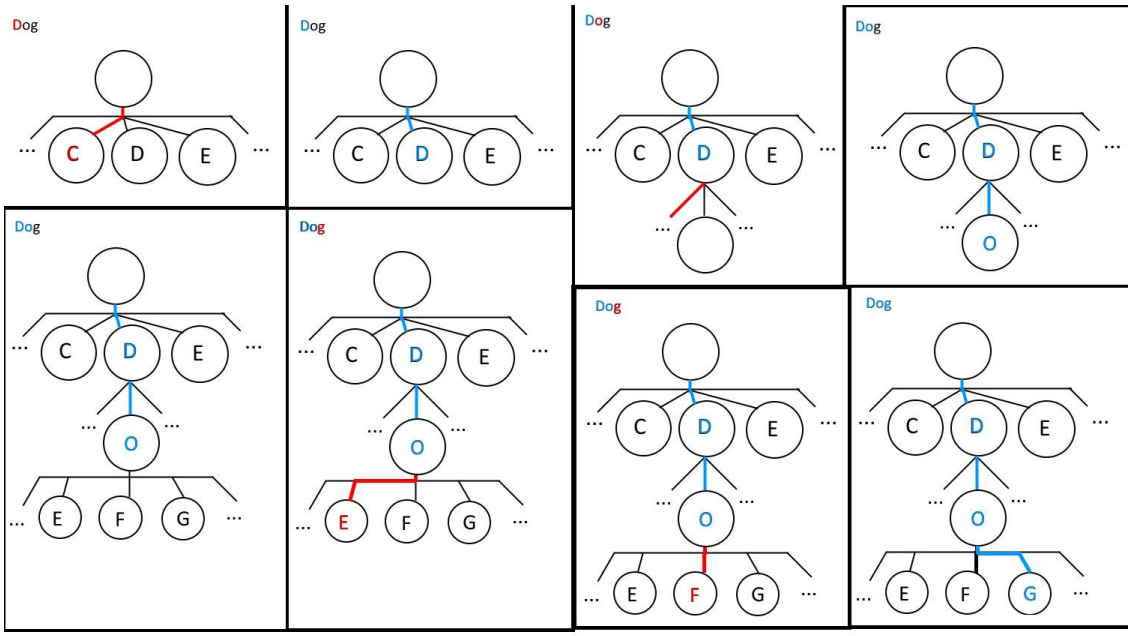

Figura 2. Exemplo do funcionamento da função 'find' para a palavra "DOG"

Dificuldades no desenvolvimento do Projeto II

Durante o desenvolvimento do projeto, enfrentamos algumas dificuldades:

- Implementação da Trie:
  A implementação da estrutura de dados trie, importante para a indexação correta das palavras do dicionário e a busca por prefixos, foi desafiadora devido à necessidade de gerenciar ponteiros, criar nós dinamicamente e garantir a integridade da trie durante as operações de inserção e busca.

- Manipulação dos Arquivos:
  A leitura e manipulação correta dos arquivos, para extrair as informações relevantes do dicionário, também foi desafiador (principalmente lidar com diferentes formatos de entrada e garantir a precisão na extração de dados).

- Interpretação de Requisitos e Formatos:
  Compreender completamente os requisitos do projeto foi essencial. A interpretação adequada dos requisitos representou um desafio na implementação.

- Lógica de Processamento de Palavras de Entrada:
  O desenvolvimento da lógica que processa as palavras de entrada, busca por prefixos na trie e exibe os resultados de maneira coerente pode ser complexo. Garantir a correta manipulação de ponteiros demandou atenção especial.

- Gerenciamento de Memória:
  A alocação e liberação adequadas de memória para os nós da trie e outros objetos dinâmicos foram importantes para evitar vazamentos de memória e garantir a eficiência do programa.

- Testes:
  Verificar se a trie está sendo construída corretamente, se os prefixos estão sendo identificados de maneira precisa e se as mensagens de saída estão formatadas adequadamente foram aspectos importantes durante a fase de testes.

- Manutenção do Código:
  Manter um código limpo e compreensível ao longo do desenvolvimento. A utilização de boas práticas de programação e a organização eficiente do código foram importantes para facilitar a manutenção e extensão do projeto.

Conclusão

O Projeto II focou na construção de uma aplicação para indexação eficiente de palavras em dicionários, utilizando uma trie. Ao resolver os desafios de identificação de prefixos e indexação de arquivos, enfrentamos obstáculos como a implementação cuidadosa da trie, manipulação eficiente de arquivos e formatação correta das saídas. Gostaríamos de destacar na conclusão a importância da compreensão detalhada dos requisitos, a aplicação de boas práticas de programação e a realização de testes para garantir a eficácia da aplicação. A implementação bem-sucedida da trie demonstrou eficiência na identificação de prefixos.

Este projeto nos proporcionou experiência prática, fortalecendo habilidades de programação e resolução de problemas. As referências abaixo nos ajudaram a superar desafios específicos. Em resumo, o Projeto II foi fundamental para a aplicação prática de estruturas de dados e algoritmos, oferecendo insights valiosos no desenvolvimento de solução.

Referencias

1. GeeksforGeeks - Trie Data Structure: https://www.geeksforgeeks.org/trie-insert-and-search/

2. Paulo Feofiloff - Estruturas de Dados: Tries (Árvores de Prefixos): https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/tries.html

3. Towards Data Science - Implementing a Trie Data Structure in Python in Less Than 100 Lines of Code: https://towardsdatascience.com/implementing-a-trie-data-structure-in-python-in-less-than-100-lines-of-code-a877ea23c1a1