Universidade Federal de Santa Catarina

INE5408 - Estruturas de Dados

Relatório Técnico: Projeto de Verificação de Cenários e Determinação de Área Limpa

Autor:

Carlos Hayden Junior

Florianópolis
Maio de 2025

Conteúdo

1 Resumo e Contextualização

Este projeto foi desenvolvido para resolver dois problemas principais relacionados ao funcionamento de um robô aspirador autônomo: (1) validação de arquivos XML contendo cenários de operação e (2) cálculo da área limpa a partir da posição inicial do robô em matrizes binárias. A solução implementa estruturas de dados lineares (pilhas e filas) para processamento eficiente dos dados, seguindo as melhores práticas de programação em C++.

2 Soluções Implementadas

2.1 Validação de Arquivos XML

A função `validar_xml` emprega uma estrutura de pilha para garantir que todas as tags XML sejam corretamente abertas e fechadas, respeitando a ordem hierárquica do aninhamento. Cada vez que uma tag de abertura é lida, ela é empilhada. Quando uma tag de fechamento aparece, é verificado se ela corresponde ao topo da pilha. Se não corresponder ou a pilha estiver vazia, o XML é considerado inválido.

Algoritmo: A validação do XML utiliza uma pilha para verificar o correto aninhamento das tags:

→ `Tags de abertura são empilhadas`

→ `Tags de fechamento devem corresponder ao topo da pilha`

→ `Desempilha quando encontra tag de fechamento correspondente`

Verificações de Erro:

- Tag de fechamento sem correspondente de abertura
- Tag de fechamento incorreta (ordem inversa)
- Tags não fechadas no final do arquivo

Explicação da Implementação da Função `validar_xml()`:

1. **Assinatura e Estrutura:**

```
// Funcao para validar o XML
bool validar_xml(string& texto) {
stack<string> tag_stack;          // Declaracao de Pilha
   para controle de aninhamento
size_t pos = 0;                  // Posicao atual no texto

while (pos < texto.length()) {
        // Logica de processamento
}
return tag_stack.empty();      // True se todas tags foram
    fechadas
}
```

2. **Processamento Passo a Passo:**

   - Fase 1: Detecção de Tags

```
        // Iterando ate encontrar um '<'
        if (texto[pos] != '<') {
            pos++;
            continue;
        }
        pos++;
        // Se nao houver mais caracteres para ler
        if (pos >= texto.length()) return false;

        // Verifica se e uma tag de fechamento
        bool is_closing = false;
        if (texto[pos] == '/') {
            is_closing = true;
```

```
        pos++;
    }

    string tag_name;
    // Extrai nome da tag ate encontrar '>'
    while (pos < texto.length() && texto[pos] != '>')
        {
        tag_name += texto[pos];
        pos++;
    }
    pos++; // Avanca alem do '>'

    // nome da tag nao pode ser vazio
    if (tag_name.empty()) return false;
    ...
```

- Fase 2: Lógica de Pilha

```
    if (is_closing) {              // Verifica se e
       uma tag de fechamento
        if (tag_stack.empty() || tag_stack.top() !=
           tag_name) {
            return false;          // Erro: tag nao
               aberta ou ordem incorreta
        }
        tag_stack.pop();           // Tag fechada
           corretamente - desempilha
    } else {
        tag_stack.push(tag_name);  // Nova tag aberta
            - empilha
    }
```

```
        ...
```

- Fase 3: Pós-Processamento

```
        return tag_stack.empty(); // Retorna true se
            todas as tags foram fechadas
```

3. **Complexidade Computacional:**

   - **Tempo**: $\mathcal{O}(n)$ — onde n é o número total de caracteres do XML.
   - **Espaço**: $\mathcal{O}(m)$ — onde m representa a profundidade máxima de aninhamento de tags.

4. **Exemplo de Execução:**

```
<cenario>
    <nome>teste</nome>
</cenario>
```

   - Passos:

```
→ Empilha "cenario"
→ Empilha "nome"
→ Desempilha "nome" (match com </nome>)
→ Desempilha "cenario" (match com </cenario>)
→ Retorna true (pilha vazia)
```

5. **Otimizações Implementadas:**

   - **Processamento em um único passe**: Evita múltiplas varreduras no texto
   - **Early return**: Retorna false imediatamente ao detectar erros
   - **Uso eficiente de memória**: A pilha só armazena nomes de tags, não o conteúdo completo

6. **Limitações Conhecidas:**

   - Não valida atributos em tags (`<tag attr=value>`)
   - Não verifica caracteres especiais em conteúdo
   - Case-sensitive (diferencia `<Tag>` de `<tag>`)

2.2 Processamento dos Cenários

A classe `Cenario` é responsável por extrair e armazenar os dados de cada cenário presente no XML. A extração é feita por meio da navegação pelas tags com a função `proxima_tag_conteudo`. A matriz é reconstruída como uma string de binários e seus dados são convertidos em uma estrutura 2D para facilitar o processamento.

2.3 Determinação da Área Limpa

Utilizou-se uma fila (estrutura de dados `queue`) para realizar a busca em largura (BFS). A área navegável é calculada a partir do ponto inicial fornecido. Apenas as células conectadas diretamente (4 direções) que contenham valor 1 e ainda não tenham sido visitadas são consideradas.

Algoritmo (BFS - Breadth-First Search): Para calcular a área conexa a partir da posição inicial:

1. **Inicialização:**
    - Conversão da matriz string para representação 2D
    - Criação de matriz de visitados
    - Fila para gerenciar pontos a serem processados
2. **Processamento:**
    - Para cada ponto, verifica vizinhança-4
    - Pontos válidos (valor 1 e não visitados) são adicionados à fila
    - Área incrementada para cada ponto processado

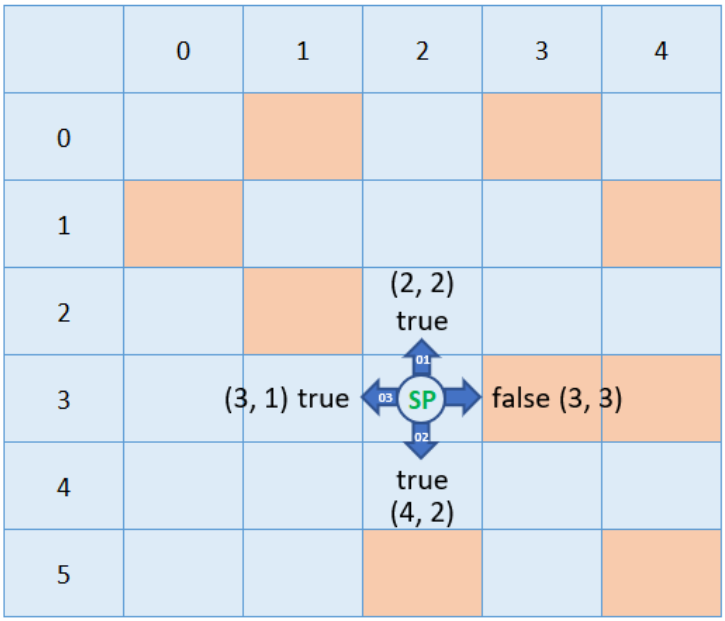

Figura 1: Exemplo de BFS em matriz binária mostrando expansão da busca

Explicação da Implementação da Função `calcular_area_limpa()`

1. **Propósito e Contexto:**

   A função `calcular_area_limpa()` implementa um algoritmo BFS (Breadth-First Search) para determinar a área conexa acessível por um robô aspirador em uma matriz binária, considerando movimentos na vizinhança-4 (cima, baixo, esquerda, direita):

2. **Assinatura e Estrutura:**

```
size_t calcular_area_limpa(const Cenario& cenario) {
    // 1. Pre-processamento da matriz
    // 2. Inicializacao das estruturas
    // 3. Nucleo do algoritmo BFS
    // 4. Retorno do resultado
}
```

3. **Processamento Passo a Passo:**

- Passo 1: Pré-processamento da Matriz

```
            vector<vector<int>> matriz(cenario.altura,
               vector<int>(cenario.largura, 0));

            // Converte a string linear para matriz 2D
            for (size_t i = 0; i < cenario.altura; i++) {
                for (size_t j = 0; j < cenario.largura; j
                   ++) {
                    size_t index = i * cenario.largura +
                       j;
                    if (index < cenario.matriz.size()) {
                        matriz[i][j] = cenario.matriz[
                           index] - '0'; // Converte char
                           '0'/'1' para int
                    }
                }
            }
        ...
```

- Passo 2: Inicialização

```
            // Verifica se a posicao inicial e valida
            if (cenario.x >= cenario.altura || cenario.y
               >= cenario.largura ||
                matriz[cenario.x][cenario.y] != 1) {
                return 0;
            }

            // Estruturas auxiliares
            queue<pair<size_t, size_t>> fila;
```

```
        vector<vector<bool>> visitado(cenario.altura,
            vector<bool>(cenario.largura, false));

        // Configuracao inicial
        fila.push(make_pair(cenario.x, cenario.y));
        visitado[cenario.x][cenario.y] = true;
        size_t area = 0;
    ...
```

- Passo 3: Núcleo do BFS

```
        // Direcoes da vizinhanca-4 (cima, baixo,
           esquerda, direita)
        const int dx[] = {-1, 1, 0, 0};
        const int dy[] = {0, 0, -1, 1};

        while (!fila.empty()) {
            auto [x, y] = fila.front();
            fila.pop();
            area++;

            // Explora os 4 vizinhos
            for (int i = 0; i < 4; i++) {
                size_t nx = x + dx[i];
                size_t ny = y + dy[i];

                // Verifica se o vizinho e valido
                if (nx < cenario.altura && ny <
                   cenario.largura &&
                    !visitado[nx][ny] && matriz[nx][
                       ny] == 1) {
                    visitado[nx][ny] = true;
```

```
                        fila.push(make_pair(nx, ny));
                    }
                }
            }
```

4. **Fluxo de Execução Exemplificado:**

   - Para uma matriz 3x3:

```
1 1 0
1 0 1
0 1 0
```

     Com ponto inicial (0,0):

     – Iteração 1: Processa (0,0), área=1

       Adiciona (1,0) e (0,1) à fila

     – Iteração 2: Processa (1,0), área=2

       Adiciona (2,0) - mas (2,0) é 0 (inválido)

     – Iteração 3: Processa (0,1), área=3

       Adiciona (1,1) - mas (1,1) é 0 (inválido)

       Adiciona (0,2) - mas (0,2) é 0 (inválido)

     – Resultado Final: Área = 3

5. **Complexidade Computacional:**

Operação	**Complexidade**	**Descrição**
Conversão da matriz	$\mathcal{O}(n^2)$	n = altura × largura
BFS	$\mathcal{O}(V + E)$	V = vértices, E = arestas (máx. $4V$)
Espaço	$\mathcal{O}(n^2)$	Matriz de visitados

6. **Otimizações Críticas:**

   - **Matriz de Visitados**: Evita reprocessamento
   - **Checagem de Limites**: Antes de cada acesso à matriz
   - **Early Return**: Retorna 0 imediatamente se posição inicial inválida
   - **Alocação Estática**: Uso de vetores pré-alocados para desempenho

7. **Diagrama de Estados do BFS:**

```
[Posição Inicial]
→ [Marcar como Visitada]
→ [Adicionar à Fila]
→ [Enquanto Fila não vazia]
    → [Processar Vizinhança]
        → [Área++]
→ [Retornar Total]
```

8. **Alternativas Consideradas e Rejeitadas:**

   - **Vizinhança-8**:

     Prós: Consideraria diagonais

     Contras: Não atendia aos requisitos do projeto

   - **Union-Find**:

     Prós: Eficiente para múltiplas consultas

     Contras: Overhead desnecessário para caso de uso único

3 Dificuldades e Desafios

3.1 Validação de XML

- Problema relativamente simples com uso direto de pilha
- Dificuldade principal foi no parsing correto das tags considerando todos os casos extremos

3.2 Cálculo da Área Limpa

- Desafio significativo na conversão da matriz string para representação 2D
- Dificuldade na implementação eficiente do BFS com vizinhança-4
- Problemas com índices e limites da matriz
- Necessidade de otimização para evitar processamento duplicado
- Testes extensivos para garantir corretude em todos os cenários

4 Referências Bibliográficas

1. DROZDEK, Adam. *Estruturas de dados e algoritmos em C++*. 4. ed. São Paulo: Cengage Learning, 2013.
2. LAFORE, Robert. *Estruturas de dados & algoritmos em Java*. 2. ed. Rio de Janeiro: Ciência Moderna, 2005.
3. Documentação C++: `https://en.cppreference.com/w/`
4. Tutorial C++: `https://www.cplusplus.com/doc/tutorial/`

5 Conclusão

O projeto demonstrou a eficácia das estruturas de dados lineares na resolução de problemas complexos. Enquanto a validação de XML foi relativamente simples com uso de pilha, o cálculo da área limpa exigiu maior esforço na implementação e testes do algoritmo BFS. A solução final atende a todos os requisitos, mostrando robustez e eficiência no processamento dos cenários.