UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS
INE-DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
INE5408 - ESTRUTURA DE DADOS

PEDRO HENRIQUE TAGLIALENHA (22203674)
MATHEUS FERNANDES BIGOLIN (22200371)

RELATÓRIO PROJETO 1

FLORIANÓPOLIS
2023

1 INTRODUÇÃO.....3
2 CÓDIGO DESENVOLVIDO.....4
3 LÓGICA DO CÓDIGO.....8
3.1 VALIDAÇÃO DO ARQUIVO XML.....8
3.2 EXTRAÇÃO DE CENÁRIOS.....8
3.3 DETERMINAÇÃO DA ÁREA DE LIMPEZA.....9
3.4 LEITURA DE ARQUIVO.....9
3.5 FLUXO PRINCIPAL.....9
4 FLUXOGRAMAS DO CÓDIGO.....10
4.1 FLUXOGRAMA DA VALIDAÇÃO DO XML.....10
4.2 FLUXOGRAMA DA DETERMINAÇÃO DA ÁREA DE LIMPEZA.....11
5 DIFICULDADES ENCONTRADAS.....12
6 CONCLUSÃO.....13
7 REFERÊNCIAS.....14

1 INTRODUÇÃO

Este relatório apresenta a solução desenvolvida para o Projeto 1 da disciplina de Estrutura de Dados, que envolve a utilização de estruturas lineares e a aplicação de conceitos de pilha e fila para processar arquivos XML contendo matrizes binárias que representam cenários de ação de um robô aspirador. O projeto tem como objetivo resolver dois problemas distintos: a validação de arquivos XML e a determinação da área do espaço que o robô deve limpar.

O primeiro problema diz respeito à validação do arquivo XML. Neste contexto, buscamos verificar o alinhamento e fechamento das marcações (*tags*) no arquivo XML. Qualquer erro de aninhamento resulta em uma mensagem de erro exibida na tela. A abordagem utilizada para resolver este problema é baseada no conceito de pilha (LIFO). Ao encontrar uma marcação de abertura, o identificador correspondente é empilhado. Quando uma marcação de fechamento é encontrada, verifica-se se o topo da pilha possui o mesmo identificador, desempilhando-o. Erros podem ocorrer caso o identificador seja diferente do esperado ou se a pilha estiver vazia ao tentar fechar uma marcação. Ao concluir a análise do arquivo, a pilha deve estar vazia; caso contrário, outro erro é sinalizado, indicando marcações não fechadas.

O segundo problema envolve a determinação da área do espaço que o robô deve limpar. Cada arquivo XML contém matrizes binárias representadas pelas marcações <altura>, <largura> e <matriz>. O objetivo é determinar a área que o robô deve limpar, com base na posição inicial (linha <x> e coluna <y>) do robô. Para isso, é necessário identificar quantos pontos com valor 1 estão na região do robô, que são considerados pertencentes ao espaço a ser limpo. Para essa tarefa, adotamos um algoritmo de reconstrução de componente conexo, utilizando uma fila (FIFO). O processo consiste em fazer uma passagem por valor da matriz de entrada para a função (ou seja, uma cópia) e iniciar a fila com a coordenada inicial do robô (caso essa coordenada esteja na área a ser limpa). Enquanto a fila não estiver vazia, as coordenadas dos vizinhos com intensidade 1 e que ainda não foram visitados são inseridas na fila, marcando essas coordenadas como visitadas na matriz cópia. O resultado final é a quantidade de 1s no único componente conexo encontrado, dado pela variável de contagem.

2 CÓDIGO DESENVOLVIDO

Figura 1 - Código solução

```cpp
// Dupla:
// Matheus Fernandes Bigolin (22200371)
// Pedro Henrique De Sena Trombini Taglialenha (22203674)

#include <array>
#include <cstddef>
#include <fstream>
#include <ios>
#include <iostream>
#include <iterator>
#include <queue>
#include <sstream>
#include <stack>
#include <stdexcept>
#include <string>
#include <utility>
#include <vector>

bool esta_valido_xml(const std::string xml) {
  // Pilha de identificadores.
  std::stack<std::string> idents{};

  auto inicio{ xml.find("<", 0) };
  // Enquanto existir marcadores de abertura ("<") no XML.
  while (inicio != std::string::npos) {
    // Marcador de fim.
    auto fim{ xml.find(">", inicio + 1) };
    auto prox_inicio{ xml.find("<", inicio + 1) };

    // Não foi encontrado ">" ou ">" está após próximo "<".
    if (fim == std::string::npos || fim > prox_inicio) {
      return false;
    }

    bool esta_fechando{ xml[inicio + 1] == '/' };

    // Obtém o identificador do marcador.
    auto ident{ xml.substr(inicio + 1 + esta_fechando,
                           fim - inicio - 1 - esta_fechando) };

    if (esta_fechando) {
      if (idents.empty() || idents.top() != ident) {
        return false;
      }

      idents.pop();
    } else {
      idents.push(ident);
    }

    inicio = prox_inicio;
```

```
  }

  return idents.empty();
}

// Retorna o corpo do próximo marcador dado pelo identificador.
std::string encontra_proximo(const std::string xml, const std::string ident) {
  const auto abertura{ "<" + ident + ">" };
  const auto fechamento{ "</" + ident + ">" };

  auto inicio{ xml.find(abertura, 0) + abertura.size() };
  auto fim{ xml.find(fechamento, inicio) };

  auto corpo{ xml.substr(inicio, fim - inicio) };

  return corpo;
}

template <typename T>
using Matriz = std::vector<std::vector<T>>;

struct Cenario {
  std::string nome;
  Matriz<bool> matriz;
  std::size_t x;
  std::size_t y;
};

// Retorna todos os cenários contidos no XML.
std::vector<Cenario> obter_cenarios(std::string xml) {
  std::vector<Cenario> cenarios{};

  auto inicio{ xml.find("<cenario>", 0) };
  while (inicio != std::string::npos) {
    auto fim{ xml.find("</cenario>", inicio + 1) };
    auto prox_inicio{ xml.find("<cenario>", inicio + 1) };

    auto cenario{ xml.substr(inicio + 9, fim - inicio - 9) };

    auto nome{ encontra_proximo(cenario, "nome") };

    std::size_t x{}, y{};
    std::stringstream(encontra_proximo(cenario, "x")) >> x;
    std::stringstream(encontra_proximo(cenario, "y")) >> y;

    std::size_t altura{}, largura{};
    std::stringstream(encontra_proximo(cenario, "altura")) >> altura;
    std::stringstream(encontra_proximo(cenario, "largura")) >> largura;

    std::stringstream matriz_str(encontra_proximo(cenario, "matriz"));

    Matriz<bool> matriz(altura, std::vector<bool>(largura));

    // Obtém a matriz da string.
    for (std::size_t i{ 0 }; i < altura; ++i) {
      std::string linha{};
```

```
      matriz_str >> linha;

      for (std::size_t j{ 0 }; j < largura; ++j) {
        matriz[i][j] = linha[j] == '1';
      }
    }

    cenarios.push_back({ nome, matriz, x, y });

    inicio = prox_inicio;
  }

  return cenarios;
}

// Variações de x e y para obter cada um dos quatro vizinhos de um ponto.
constexpr std::array<std::pair<int, int>, 4> DIRECOES{
  { { 0, 1 }, { 1, 0 }, { -1, 0 }, { 0, -1 } }
};

std::size_t calcular_area_limpa(Matriz<bool> matriz, std::size_t x,
                                std::size_t y) {
  // Fila de pontos (x, y).
  std::queue<std::pair<std::size_t, std::size_t>> fila{};

  std::size_t cont{ 0 };
  // Verifica se o primeiro elemento será limpo na matriz.
  if (matriz[x][y]) {
    fila.push({ x, y });
    matriz[x][y] = false;
    ++cont;
  }

  const auto m{ matriz.size() }, n{ matriz[0].size() };

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

    // Visita todos os vizinhos de (x, y).
    for (auto [dx, dy] : DIRECOES) {
      auto nx{ x + dx };
      auto ny{ y + dy };

      // Verifica se o vizinho está dentro da matriz.
      if ((dx == -1 && x == 0) || nx >= m || (dy == -1 && y == 0) || ny >= n) {
        continue;
      }

      if (matriz[nx][ny]) {
        fila.push({ nx, ny });
        matriz[nx][ny] = false;
        ++cont;
      }
    }
  }
```

```
  return cont;
}

// Lê um arquivo e retorna o seu conteúdo.
std::string ler_arquivo(std::string nome_arq) {
  std::ifstream arq(nome_arq, std::ios_base::binary | std::ios_base::in);

  if (!arq.is_open()) {
    throw std::runtime_error("Não foi possível abrir " + nome_arq);
  }

  const std::istreambuf_iterator<char> it{ arq }, fim;
  const std::string conteudo(it, fim);

  if (!arq) {
    throw std::runtime_error("Não foi possível ler " + nome_arq);
  }

  arq.close();
  return conteudo;
}

int main(void) {
  char xmlfilename[100];

  std::cin >> xmlfilename;

  auto xml{ ler_arquivo(xmlfilename) };

  if (!esta_valido_xml(xml)) {
    std::cerr << "erro\n";

    return EXIT_FAILURE;
  }

  auto cenarios{ obter_cenarios(xml) };

  for (auto &cenario : cenarios) {
    auto area{ calcular_area_limpa(cenario.matriz, cenario.x, cenario.y) };

    std::cout << cenario.nome << ' ' << area << '\n';
  }

  return EXIT_SUCCESS;
}
```

Fonte: Desenvolvido pelos autores(2023)

3 LÓGICA DO CÓDIGO

Nesta seção, descreveremos a lógica do código desenvolvido para resolver os problemas propostos para o projeto 1. O código foi organizado em funções que abordam diferentes aspectos do processamento de arquivos XML e da determinação da área de limpeza do robô.

3.1 VALIDAÇÃO DO ARQUIVO XML

A função esta_valido_xml é responsável por verificar se o arquivo XML está bem formado, ou seja, se as marcações estão aninhadas e fechadas corretamente.

Uma pilha é utilizada para acompanhar as marcações encontradas no arquivo. Quando uma marcação de abertura é encontrada, seu identificador é empilhado. Quando uma marcação de fechamento é encontrada, o código verifica se o identificador corresponde ao identificador no topo da pilha. Erros são tratados, caso contrário.

A função percorre o arquivo XML em busca de marcações de abertura e fechamento e retorna verdadeiro se o arquivo estiver bem formado e falso se houver problemas de aninhamento ou fechamento.

3.2 EXTRAÇÃO DE CENÁRIOS

A função obter_cenarios é responsável por extrair informações sobre os cenários do arquivo XML, incluindo seus nomes, dimensões, matrizes e posições iniciais do robô.

Ela procura por marcações <cenario> no XML, extrai o conteúdo de cada cenário e analisa os valores de altura, largura, x e y, além da matriz que representa o cenário.

Os dados extraídos são organizados em uma estrutura de dados que facilita o processamento posterior.

3.3 DETERMINAÇÃO DA ÁREA DE LIMPEZA

A função calcular_area_limpa é responsável por determinar a área que o robô deve limpar em um cenário.

Ela utiliza uma fila (FIFO) para realizar uma busca em largura na matriz, começando pela posição inicial do robô.

À medida que a busca se expande, os pontos visitados são marcados como já processados na matriz. A função continua até que todos os pontos conectados à posição inicial tenham sido visitados.

A quantidade de pontos visitados representa a área que o robô deve limpar.

3.4 LEITURA DE ARQUIVO

A função ler_arquivo lida com a leitura do conteúdo do arquivo XML. Ela recebe o nome do arquivo como entrada e retorna o conteúdo em formato de string.

3.5 FLUXO PRINCIPAL

O fluxo principal do programa está contido na função main. Ele lê o nome do arquivo XML da entrada padrão, lê o conteúdo do arquivo usando a função ler_arquivo e, em seguida, executa a validação com esta_valido_xml.

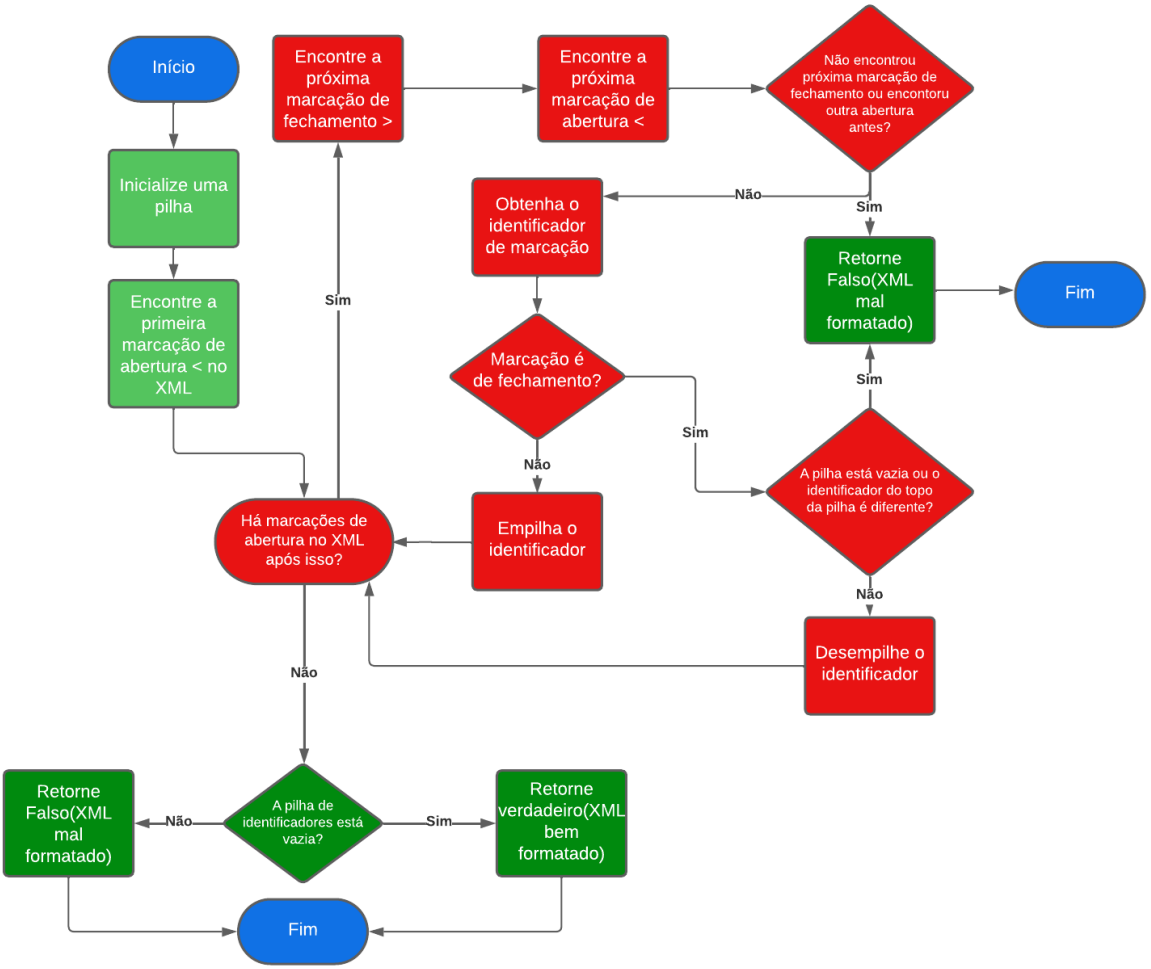
Se o arquivo XML for válido, o código extrai informações dos cenários usando obter_cenarios e, para cada cenário, calcula a área de limpeza com calcular_area_limpa. Os resultados são impressos na saída padrão.

4 FLUXOGRAMAS DO CÓDIGO

Nesta seção, serão apresentados fluxogramas que descrevem a lógica do código desenvolvido. Esses fluxogramas abrangem a validação de arquivos XML e a determinação da área de limpeza em cenários do robô aspirador.

4.1 FLUXOGRAMA DA VALIDAÇÃO DO XML

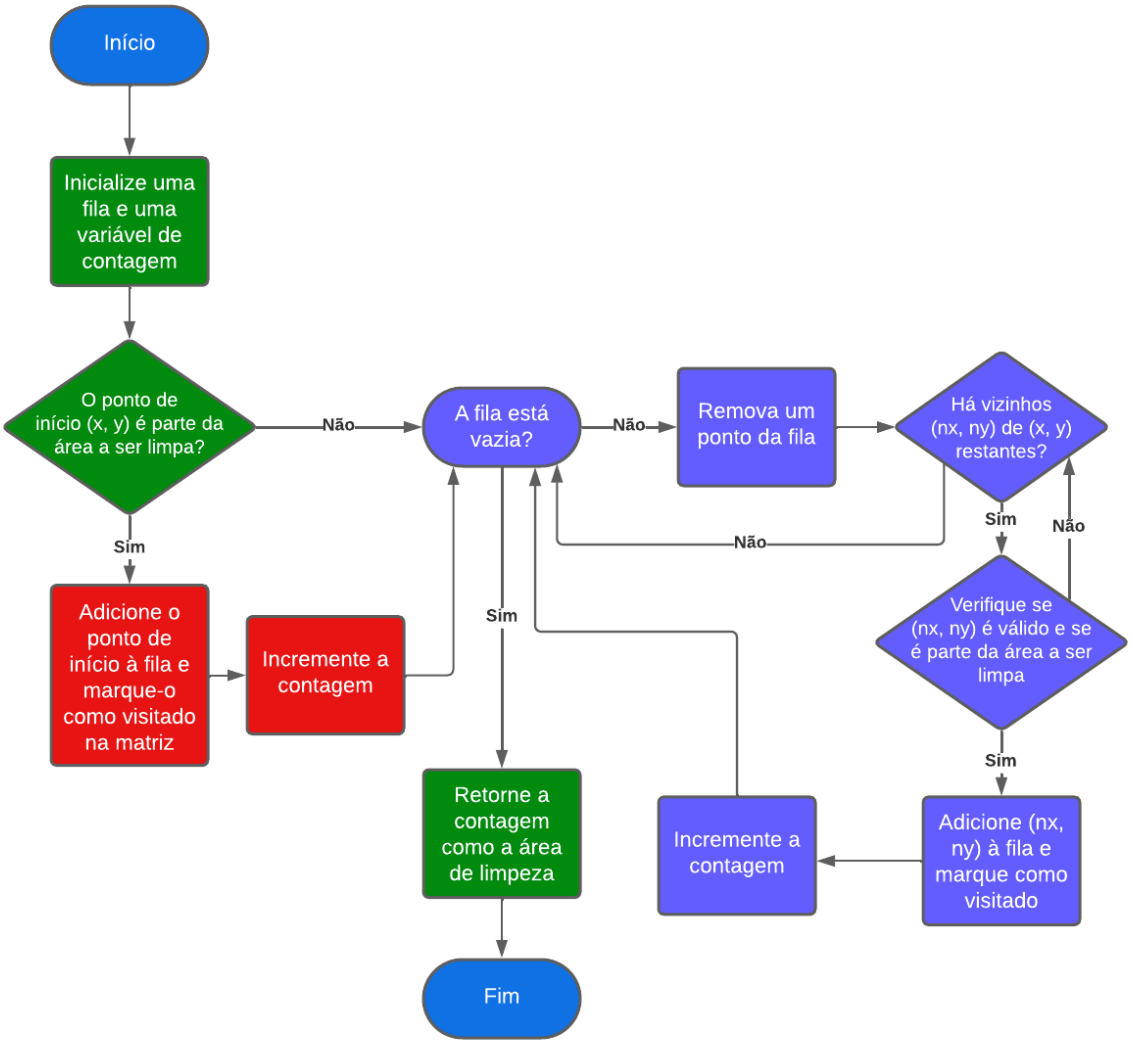
Figura 2 - Validação do XML



Fonte: Elaborado pelos autores(2023)

4.2 FLUXOGRAMA DA DETERMINAÇÃO DA ÁREA DE LIMPEZA

Figura 3 - Fluxograma da Área de limpeza



Fonte: Elaborado pelos autores(2023)

5 DIFICULDADES ENCONTRADAS

Para esse projeto de estrutura de dados, a maior dificuldade e a parte mais trabalhosa não foi a implementação dos algoritmos em si para resolver o primeiro e o segundo problema, mas o processo de extração de cenários do arquivo XML. Considerando a necessidade do uso de procedimentos para tratamento de *streams* da biblioteca padrão do C++, a resolução desse problema, desse modo, proporcionou um maior entendimento acerca de como o processamento de arquivos e a interpretação de dados (*parsing*) são feitos utilizando a linguagem, além de promover uma maior compreensão acerca de sua biblioteca padrão.

6 CONCLUSÃO

O desenvolvimento deste projeto proporcionou uma sólida compreensão das estruturas de dados e algoritmos essenciais na área de estrutura de dados. A utilização eficaz de pilhas e filas demonstrou a importância dessas estruturas para a resolução de problemas práticos, como a validação de XML e o processamento de matrizes. Além disso, a abordagem de extração de informações dos cenários e o cálculo da área de limpeza reforçam a aplicação de estruturas de dados para tarefas do mundo real.

O projeto também ressaltou a importância da organização e estruturação do código para facilitar a manutenção e compreensão. A documentação das operações e os fluxogramas contribuíram para a clareza do código e forneceram uma referência útil para entender sua lógica subjacente. Em resumo, este projeto serviu como uma oportunidade valiosa para aplicar conceitos teóricos de estrutura de dados em um contexto prático e real.

7 REFERÊNCIAS

https://www.portalgsti.com.br/2018/04/listas-pilhas-e-filas-em-c.html
https://cplusplus.com/reference/list/list/
https://en.cppreference.com/w/cpp/io/basic_stringstream