Universidade Federal de Santa Catarina
Curso de Graduação em Ciências da Computação
INE5408 - Estruturas de Dados

Caio César Rodrigues de Aquino
Luan da Silva Moraes

Projeto I - Relatório

Florianópolis
2024

SUMÁRIO

1 INTRODUÇÃO

Este projeto foi desenvolvido com o objetivo de resolver dois problemas propostos na disciplina: 1. a validação da correta formação de um arquivo do tipo XML; 2. a determinação da área limpa por um robô aspirador quando este é posicionado em uma matriz binária, tendo em vista que ele apenas realiza a limpeza de células cujo valor é 1.

2 DESENVOLVIMENTO

As seções que se seguem discorrerão sobre as soluções, dificuldades encontradas e o funcionamento do projeto

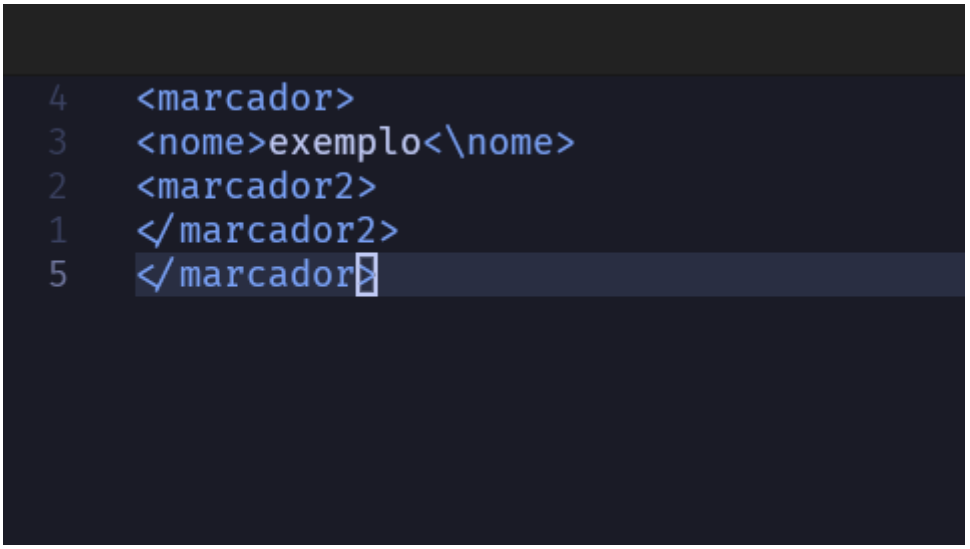
2.1 SOLUÇÃO PARA O PROBLEMA 1

O problema 1 pode ser facilmente resolvido com o emprego de uma pilha. Para implementar a solução, foi escrita uma classe chamada *XmlParser*, responsável por ler o conteúdo do arquivo XML passado como parâmetro, validar sua sintaxe e construir uma árvore de nós que representam cada marcação presente no arquivo XML – Esta última funcionalidade de *XmlParser* será discutida posteriormente, na seção 2.2.1.

Dando início à leitura do conteúdo do arquivo, caractere por caractere, o programa para ao encontrar uma *tag* (ou marcação). Se a *tag* for de abertura, isto é, estiver no formato “<identificador>”, basta empilhar o identificador. Caso for uma marcação de fechamento, disposta no formato “</identificador>”, compara-se o identificador de fechamento com o identificador da *tag* presente no topo da pilha e levanta exceção caso difiram. Além disso, a exceção também é levantada se a pilha estiver vazia, pois a marcação de fechamento está presente quando nenhuma marcação foi aberta. Ao terminar de ler o arquivo, é feita uma checagem no

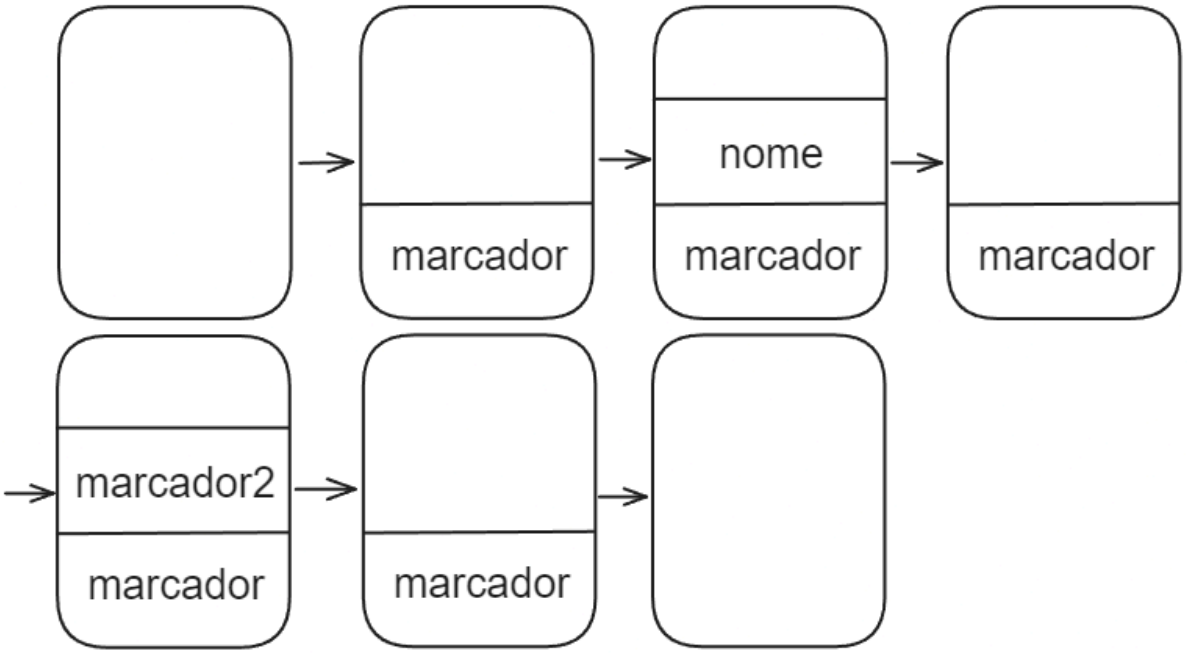
tamanho da pilha, uma vez que se ela não estiver vazia significa que existe uma ou mais *tags* que não foram fechadas. Nesta situação, uma exceção é levantada.

Figura 1.a - Exemplo de XML corretamente formado



Fonte: acervo dos autores

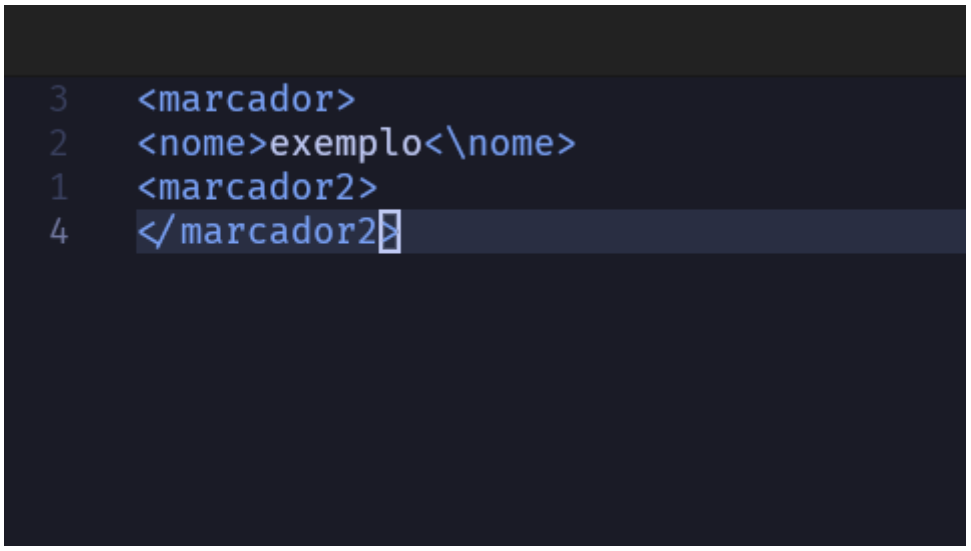
Figura 1.b - Processo de validação do arquivo XML



Fonte: acervo dos autores

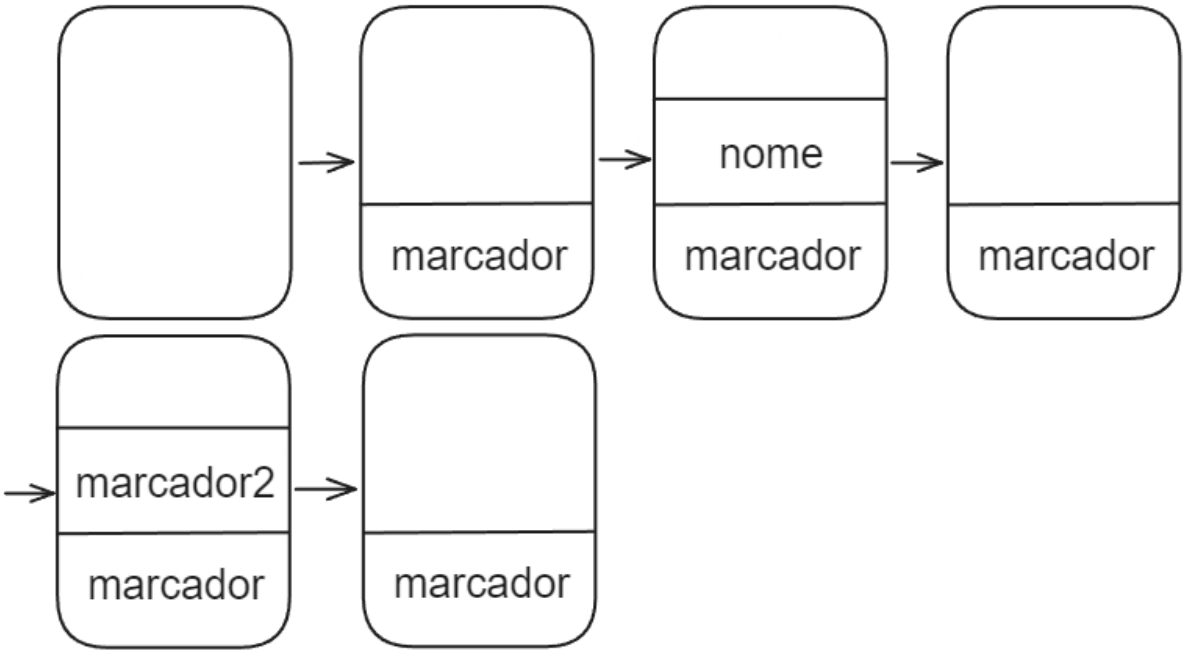
Pela figura 2, o algoritmo de validação passa lendo o arquivo XML até encontrar uma marcação, a qual tem o identificador empilhado. O primeiro elemento, no caso do código da figura 1, é o que tem o identificador “marcador”. A próxima *tag* no arquivo é a “nome”. Como uma nova *tag* está se abrindo, ela se caracteriza como um filho de “marcador”. De forma semelhante, o identificador “nome” é empilhado. Se alguma *tag* de fechamento aparecer na sequência, ela deve possuir identificador “nome”, caso contrário não é um arquivo válido. Isso é precisamente o que acontece neste primeiro exemplo. Portanto, a *tag* nome é desempilhada e o programa prossegue a leitura. Em seguida uma *tag* “marcador2” é aberta e fechada corretamente e, finalmente, a *tag* “marcador” é fechada ao terminar de ler o arquivo, o que qualifica um arquivo de XML bem formado.

Figura 2.a - Exemplo de arquivo XML mal formado



Fonte: acervo dos autores

Figura 2.b - Processo de validação do arquivo XML



Fonte: acervo dos autores

Neste segundo exemplo, a diferença é que o elemento “marcador” não possui uma *tag* de fechamento. Portanto, todo o processo intermediário se mantém igual ao do primeiro exemplo. Ao terminar de ler o arquivo é esperado que a pilha contenha um identificador: o “marcador”. Neste caso, isso constitui um arquivo XML inválido.

2.2 SOLUÇÃO PARA O PROBLEMA 2

O problema diz respeito à encontrar a área que um robô aspirador irá limpar ao ser posicionado em uma matriz binária. No entanto, isso implica em um outro problema que constitui uma dependência do problema 2: para encontrar a área limpa pelo robô em cada cenário, deve-se, antes, ler corretamente as informações destes cenários fornecidas nos arquivos XML. Por esta razão, se faz necessário discutir como essas informações foram acessadas e estruturadas.

2.2.1 SOLUÇÃO PARA A LEITURA DAS INFORMAÇÕES NO ARQUIVO XML

A solução encontrada pela dupla foi implementar uma estrutura de dados do tipo árvore para a representação, em memória, da hierarquia apresentada em um tipo de arquivo como o XML, visto que a sua estrutura é naturalmente recursiva. A árvore mencionada, é implementada pela classe *XmlTree*, que é construída pela classe *XmlParser* ao chamar seu método estático, *parse*. O método supracitado recebe como parâmetro uma *std::string*, o caminho para o arquivo XML a ser "*parseado*".

Figura 3 - Definição da classe *XmlTree*

```
27  class XmlParser;
26
25  class XmlTree {
24  public:
23
22      XmlTree();
21
20      XmlTree(const XmlTree& other);
19
18      XmlTree& operator=(const XmlTree& other);
17
16      std::vector<XmlTree> operator[](const std::string& identifier);
15
14      void addChild(XmlTree& child);
13
12      std::string getContent() const;
11
10      void setContent(const std::string& content);
9
8       std::vector<XmlTree> getChildren() const;
7
6       std::string getIdentifier() const;
5
4   private:
3       friend class XmlParser;
2
1       XmlTree(const std::string& identifier);
34
1       XmlTree(const std::string& identifier, const std::string& content,
2               const std::vector<XmlTree>& children);
3
4       std::string identifier;
5       std::string content;
6       std::vector<XmlTree> children;
7   };
```

Fonte: acervo dos autores

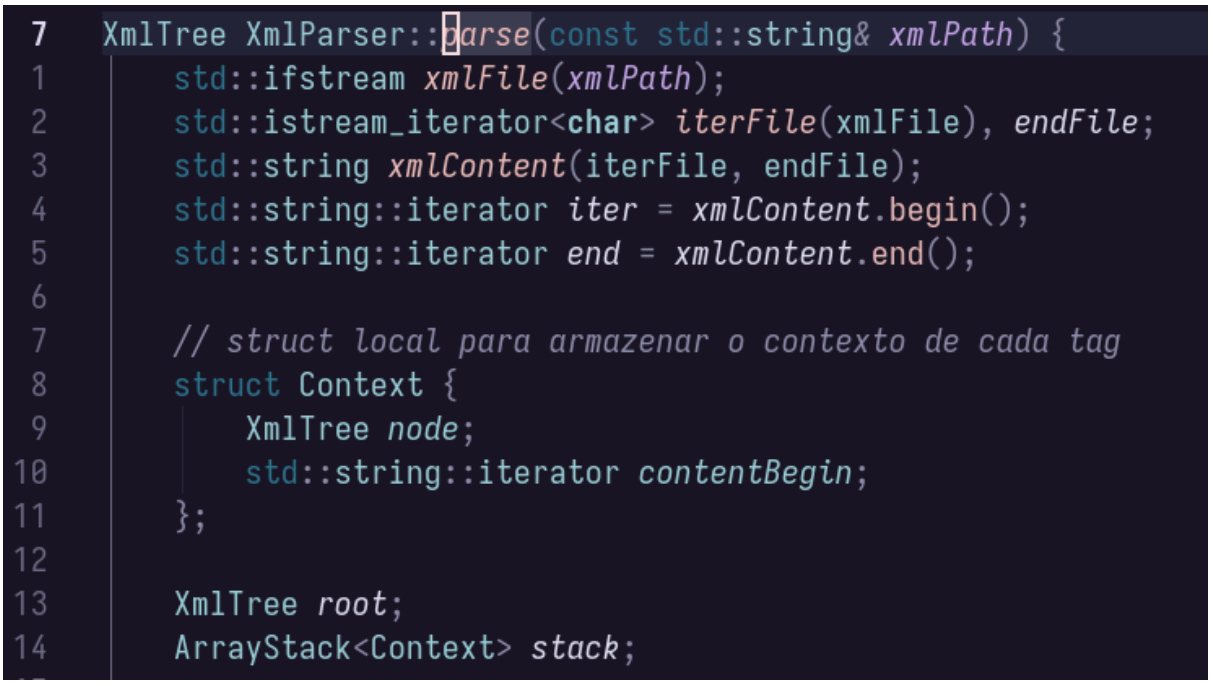
Figura 4 - Definição da classe *XmlParser*

```
3   class XmlParser {
2     public:
1       static XmlTree parse(const std::string& xmlPath);
12
1     private:
2       static std::string readIdentifier(std::string::iterator& iter);
3
4       // Utilizado para ler o conteúdo dentro de uma tag, a partir de iteradores
5       // que apontam para o começo e para o fim do conteúdo.
6       static std::string readContent(std::string::iterator contentBegin,
7                                      std::string::iterator contentEnd);
8   };
9
10  #endif
```

Fonte: acervo dos autores

Como mencionado anteriormente, o método que realiza a validação do arquivo XML e o converte para uma árvore em memória é o *XmlParser::parse*. Os demais métodos privados da classe servem como utilitários para o método *parse*.

Figura 5 - Início do método *parse*



```
7   XmlTree XmlParser::parse(const std::string& xmlPath) {
1       std::ifstream xmlFile(xmlPath);
2       std::istream_iterator<char> iterFile(xmlFile), endFile;
3       std::string xmlContent(iterFile, endFile);
4       std::string::iterator iter = xmlContent.begin();
5       std::string::iterator end = xmlContent.end();
6
7       // struct local para armazenar o contexto de cada tag
8       struct Context {
9           XmlTree node;
10          std::string::iterator contentBegin;
11      };
12
13      XmlTree root;
14      ArrayStack<Context> stack;
```

Fonte: acervo dos autores

Primeiramente, é feita a leitura do conteúdo presente no arquivo residente do caminho especificado via argumento. Em seguida, para melhorar a legibilidade do código e facilitar o agrupamento de dados, cria-se uma *struct* chamada *Context*. Ela é importante porque no momento em que uma *tag* é aberta, esta precisa ser empilhada. Todavia, se for decidido empilhar apenas o identificador da *tag*, não será possível ler o seu conteúdo de forma trivial quando seus elementos descendentes, se houverem, forem tratados, pois o programa deveria ter que encontrar a *tag* de abertura novamente, navegando em sentido ao começo do arquivo, o que é desnecessário. O mesmo vale para os próprios descendentes. Dito isso, o empilhamento também de um ponteiro para o começo do conteúdo do elemento se mostra como uma alternativa promissora. Para esclarecimento, o projeto corrente está definindo como “conteúdo de um elemento XML” todo o conteúdo que se encontra entre a sua abertura e seu fechamento, na forma de *string*.

Figura 6 - Trecho principal de *parse*

```
23  // lê caractere por caractere
1   while (iter != end) {
2       if (*iter == '<') {
3           // caso seja uma tag de fechamento
4           if (*(++iter) == '/') {
5               // o fim do conteúdo da tag que está fechando é em '<'
6               std::string::iterator contentEnd = iter - 1;
7               iter++;
8
9               std::string identifier = readIdentifier(iter);
10              // se for uma tag de fechamento e a pilha estiver vazia
11              // ou o identificador da tag de fechamento não condiz com
12              // a tag no topo da pilha, significa que o arquivo xml está
13              // mal formado
14              if (stack.empty() || identifier != stack.top().node.getIdentifier()) {
15                  throw std::runtime_error("Invalid XML");
16              }
17
18              // recupera o contexto da tag atual
19              Context current = stack.pop();
20              std::string content(readContent(current.contentBegin, contentEnd));
21              current.node.setContent(content);
22              // se após desempilhar a pilha ficar vazia, significa que
23              // a tag desempilhada é a mais externa, indicando que é a raíz
24              if (stack.empty()){
25                  root = current.node;
26              } else {
27                  // caso contrário, adiciona a tag atual na lista de filhos
28                  // da tag do topo, que é pai da atual
29                  stack.top().node.addChild(current.node);
30              }
```

Fonte: acervo dos autores

O programa, então, inicia o processamento do conteúdo do arquivo, caractere por caractere. Assim que uma *tag* de fechamento é encontrada, salva-se um ponteiro para o caractere ‘<’ desta *tag* – Relembrando que uma *tag* de fechamento possui o formato “</identificador>”. Este ponteiro caracteriza o final do conteúdo pertencente ao elemento que está atualmente no topo da pilha de contextos. Após uma verificação da corretude da formação do arquivo XML, desempilha-se o elemento cujo fechamento foi encontrado. Neste momento é possível constatar a utilidade do ponteiro para início do conteúdo do elemento que foi empilhado, já que basta ler o conteúdo no intervalo [*current.contentBegin*,

contentEnd. Feito isso, o nó atual é adicionado na lista de filhos do, agora, nó presente no topo da pilha, se houver. Do contrário, o nó atual só pode ser a raiz da árvore.

Figura 7 - Trecho final do método *parse*

```
            } else {
                // se for uma tag de abertura, apenas empilha-a com
                // um ponteiro para o início do seu conteúdo
                std::string identifier = readIdentifier(iter);
                stack.push({XmlTree(identifier), ++iter});
                continue;
            }
        }
        iter++;
    }

    // se ao final da leitura do arquivo a pilha não estiver vazia,
    // o arquivo xml está mal formado
    if (!stack.empty()) {
        throw std::runtime_error("Invalid XML");
    }

    // retorna a raíz (caso nao deu pra entender)
    return root;
}
```

Fonte: acervo dos autores

No caso negativo, isto é, caso a *tag* encontrada seja de abertura, o programa apenas empilha o seu identificador juntamente com o ponteiro para o início de seu conteúdo.

Ao final do *loop while*, incrementa-se o iterador principal para prosseguir com o processamento até que alcance o final do arquivo.

2.2.2 RETORNANDO AO PROBLEMA 2

Com a árvore de elementos XML em mãos, agora é possível prosseguir com a resolução do problema 2. Na função *main* do programa é onde todas as classes se encontram.

Figura 8 - função *main*

```
31      XmlTree xml;
30      try {
29          xml = XmlParser::parse(xmlPath);
28      }
27      catch (const std::runtime_error& e) {
26          std::cout << "erro\n";
25          return 0;
24      }
23
22      for (auto& scenario : xml["cenario"]) {
21          int width = std::stoi(scenario["dimensoes"][0]["largura"][0].getContent());
20          int height = std::stoi(scenario["dimensoes"][0]["altura"][0].getContent());
19
18          // a matriz vem do xml como uma string unidimensional.
17          // Portanto, transforma-se-a em uma matriz bidimensional antes de
16          // computar a área varrida pelo robô
15          std::string xmlMatrix = scenario["matriz"][0].getContent();
14          std::vector<std::vector<int>> matrix(height);
13          for (int i = 0; i < height; i++) {
12              for (int j = 0; j < width; j++) {
11                  matrix[i].push_back(xmlMatrix[i * width + j] - '0');
10              }
9           }
8
7           int robotX = std::stoi(scenario["robo"][0]["x"][0].getContent());
6           int robotY = std::stoi(scenario["robo"][0]["y"][0].getContent());
5           Coordinate robotPos = {robotX, robotY};
4
3           int totalArea = Solver::solve(matrix, robotPos);
2           std::cout << scenario["nome"][0].getContent() << " " << totalArea << '\n';
1       }
```

Fonte: acervo dos autores

Após efetuar o *parsing* do arquivo XML, o programa utiliza um laço para processar todos os cenários fornecidos por ele. Como conteúdo do nó com o

identificador “matriz” é guardado como uma única *string*, sem quaisquer caracteres de quebra de linha, optou-se por construir uma matriz bidimensional a partir da matriz unidimensional proveniente do arquivo para que o acesso à cada célula da matriz binária seja mais legível. O programa chama, então, o método escrito para resolver o problema 2 depois de adquirir todas as informações necessárias do cenário em questão.

Figura 9 - Método *solve*

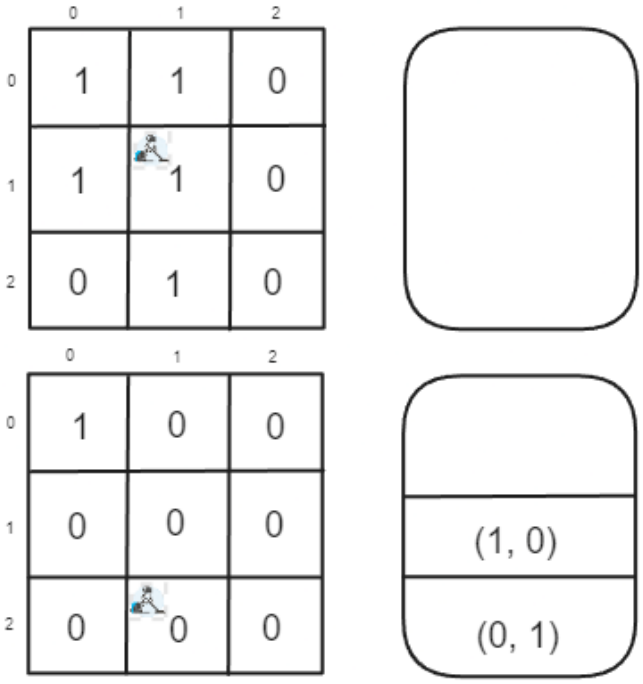
```
31  int Solver::solve(std::vector<std::vector<int>>& matrix, Coordinate robotPos) {
30      if (matrix[robotPos.x][robotPos.y] == 0) {
29          return 0;
28      }
27
26      int height = matrix.size();
25      int width = matrix[0].size();
24
23      ArrayStack<Coordinate> cellsToVisit(2350);
22
21      matrix[robotPos.x][robotPos.y] = 0;
20      int totalArea = 1;
19
18      Coordinate offsets[] = { {0, -1}, {-1, 0}, {0, 1}, {1, 0} };
17      while (true) {
16          for (const auto& offset : offsets) {
15              Coordinate newPos = {robotPos.x + offset.x, robotPos.y + offset.y};
14              if (newPos.x < 0 || newPos.x >= height || newPos.y < 0 || newPos.y >= width) {
13                  continue;
12              }
11              if (matrix[newPos.x][newPos.y] == 1) {
10                  // marca a posição empilhada como 0 para não ser empilhada novamente
9                   // futuramente
8                   matrix[newPos.x][newPos.y] = 0;
7                   totalArea++;
6                   cellsToVisit.push(newPos);
5               }
4           }
3
2           if (cellsToVisit.empty()) {
1               return totalArea;
34          }
1
2           robotPos = cellsToVisit.pop();
3       }
4   }
```

Fonte: acervo dos autores

A solução adotada foi empilhar os vizinhos da vizinhança-4 da posição atual do robô que possuem o valor 1, zerando suas células para que não ocorra, novamente, o empilhamento delas num momento posterior. Para cada vizinho empilhado, incrementa-se a variável acumuladora *totalArea*. Ao final do ciclo, atualiza-se a posição atual do robô para aquela que se encontra no topo da pilha e o processo é repetido até que a pilha se esvazie, retornando o valor de *totalArea*.

Para ilustrar este algoritmo, segue algumas figuras para um exemplo em que o robô é posicionado em uma matriz quadrada de ordem 3.

Figura 10.a - Primeira e segunda etapas

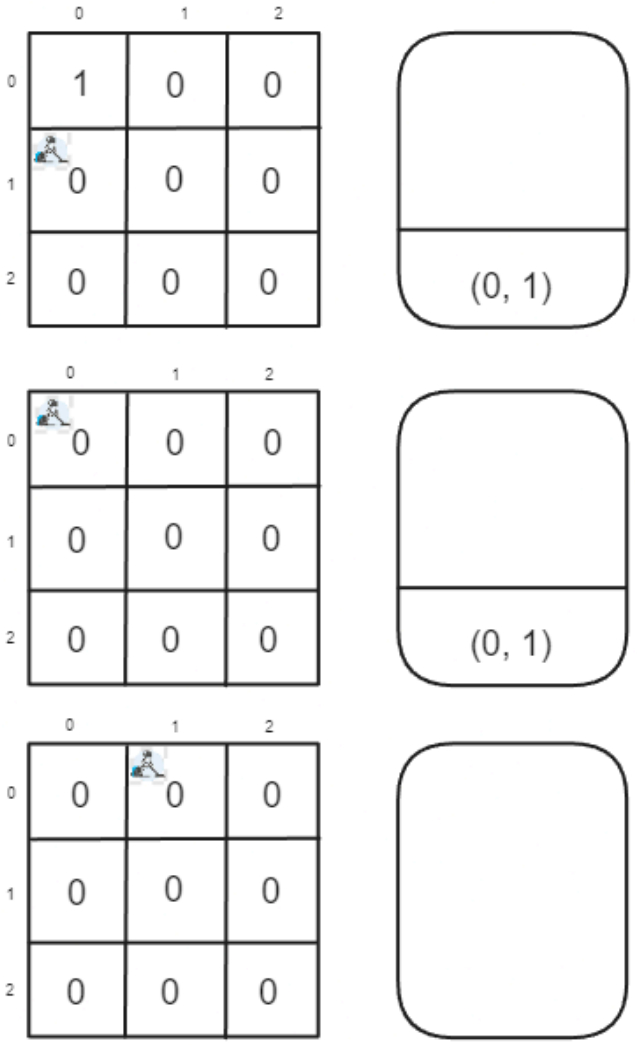


Fonte: acervo dos autores

À esquerda da imagem está o robô aspirador na matriz binária; à direita, a pilha de células a serem visitadas. Neste exemplo hipotético, a posição inicial do robô é (1, 1). Como os vizinhos (0, 1), (1, 0) e (2, 1) valem 1, eles serão empilhados para uma visita num momento futuro. No momento em que são empilhados, todos são marcados como 0 e a área é incrementada em 1 para cada vizinho empilhado.

Na primeira etapa, a célula do robô também é contabilizada. Ao final da iteração do laço, um *pop* é efetuado na pilha, atualizando a posição do robô. Por este motivo, a pilha na segunda etapa não contém a célula (2, 1). Nesta posição, nenhum vizinho do robô pertence à região conexa, então nada acontece além de atualizar a sua posição novamente. O procedimento descrito permanece acontecendo até que a pilha se esvazie e nenhum vizinho pertencente à região conexa seja encontrado, conforme a figura 10.b

Figura 10.b - Demais etapas do exemplo



Fonte: acervo dos autores

Vale notar que quando o robô chega em (0, 0), o vizinho (0, 1) não é empilhado novamente, o que resultaria em um procedimento sem fim, porque ele foi marcado como 0 quando foi empilhado.

3 CONCLUSÃO

O projeto proposto teve como objetivo exercitar a aplicação de variadas estruturas de dados em problemas de computação. Os problemas descritos no enunciado do trabalho foram muito importantes, não só para a consolidação da teoria vista previamente em aula, como também para o desenvolvimento de um pensamento crítico e a capacidade de cooperação, uma vez que cabe aos alunos discutirem e decidirem como estruturar o programa e as soluções.

No decorrer do desenvolvimento deste trabalho, três dificuldades foram encontradas. A primeira: pensar numa maneira de ler o conteúdo de cada marcação no arquivo XML, a qual levou um tempo considerável para ser superada; a segunda, por sua vez, foi com relação aos deslocamentos do robô dentro da matriz sem que ocorressem *segmentation fault*s, mas foi resolvido rapidamente; por fim, a terceira foram os múltiplos empilhamentos de vizinhos já presentes na pilha de vizinhos.

No mais, este trabalho foi uma ótima oportunidade para praticar mais a linguagem de programação C++ e retirar os conceitos de estruturas de dados do papel.

4 REFERÊNCIAS

1. C++ reference. C++ reference, 2024. Disponível em: https://en.cppreference.com/w/. Acesso em: maio de 2024.
2. MDN Web Docs. MDN Web Docs, 2024. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/API/Document_Object_Model/Introduction. Acesso em: maio de 2024.