Relatório: Projeto I - verificação de cenários e determinação de área limpa por um robô aspirador

Juliana Miranda Bosio

1 Introdução

O programa simula um robô de limpeza que processa dados em formato XML para identificar e calcular a área limpável em um cenário representado por uma matriz binária, onde cada célula corresponde à uma unidade de área e é marcada como 1 (área pertencente ao espaço a ser limpo) ou 0 (não pertencente).

A partir de um arquivo XML com um ou mais cenários, o programa verifica a estrutura correta das tags com o uso de uma pilha, e, para cada cenário, calcula a "área de limpeza"ao redor da posição inicial do Robô.

A área de limpeza é definida como o número de células 1 contíguas alcançáveis a partir de uma posição inicial, considerando apenas células adjacentes nas direções cima, baixo, esquerda e direita. Esse cálculo é feito usando uma busca em largura (BFS), implementada com uma fila.

2 Primeiro problema: validação de arquivo XML

A função `verificarAninhamentoXML` é responsável por verificar se o arquivo XML que descreve os cenários do robô de limpeza está bem estruturado, ou seja, se todas as tags XML estão corretamente aninhadas e fechadas, para evitar falhas no cálculo das áreas e impedir problemas de interpretação pelo robô.

2.1 Lógica da Função

A função usa uma pilha (`ArrayStack`), incluída por meio de um arquivo ".h", para armazenar as tags de abertura encontradas no XML, garantindo que as tags sejam fechadas na ordem inversa de abertura.

Percorre-se o texto XML caractere por caractere. Quando encontra um caractere `<`, que indica o início de uma tag, a função busca o `>` para identificar o final da tag.

Se não encontrar um `<`, a função retorna `false`, indicando erro de aninhamento, pois isso significaria uma tag não fechada. Caso encontre, a função extrai o conteúdo da tag entre os símbolos `<` e `>`. Esse conteúdo pode ser uma tag de abertura (ex: `<nome>`) ou de fechamento (ex: `(</nome>)`.

Se a tag começa com '/', ela é de fechamento. A função então verifica se a pilha contém uma tag de abertura correspondente no topo. Se não houver correspondência ou a pilha estiver vazia, o XML não está corretamente aninhado e a função retorna false.

Caso contrário, remove-se o item do topo da pilha, pois a tag foi corretamente fechada.

Se a tag não começa com '/', ela é uma tag de abertura e é adicionada ao topo da pilha.

Ao final da análise, se a pilha estiver vazia, isso indica que todas as tags de abertura foram corretamente fechadas, e a função retorna true, confirmando o XML bem aninhado. Caso contrário, retorna false.

2.2 Figura

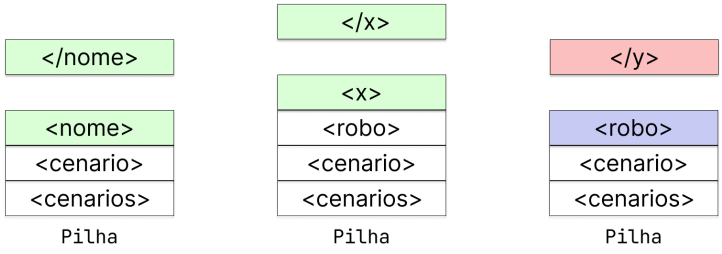

Figura 1: Exemplo figurativo do uso da Pilha e Erro de Aninhamento

2.3 Implementação

```
bool verificarAninhamentoXML(string& texto) {
    ArrayStack<string> pilha;
    size_t i = 0;

    while (i < texto.size()) {
        if (texto[i] == '<') {
            // Procura a tag de fechamento a partir da posição i
            size_t j = texto.find('>', i);
            // Verifica se a busca falhou (retornou npos)
            if (j == string::npos) {
                // Erro: Tag não fechada
                return false;
            }

            // Extrai o nome da tag
            string tag = texto.substr(i+1, j-i-1);

             // Verifica se a tag possui um '<' dentro dela
            if (tag.find('<', i) != string::npos) return false;

            if (!tag.empty() && tag[0] == '/') {
                // É uma tag de fechamento
                if (pilha.empty()  pilha.top() != tag.substr(1)) {
                    // Se a pilha não conter a tag de abertura
                    // Ou se tentamos fechar uma tag que não é a esperada
                    return false; // Erro de aninhamento
                }
                pilha.pop();
            } else {
                // É uma tag de abertura
                pilha.push(tag);
            }
            i = j; // Move para depois de '>'
        }
        i++;
    }
    return pilha.empty();  // Se a pilha estiver vazia, o XML está bem aninhado
}
```

3 Segundo problema: determinação de área do espaço que o robô deve limpar

A determinação da área que o robô deve limpar envolve calcular o espaço total que ele consegue alcançar e limpar a partir de uma posição inicial em um cenário representado por uma matriz binária. Nesta matriz, cada célula é marcada como '1' (área limpável) ou '0' (obstáculo), com o objetivo de identificar todas as células limpáveis conectadas à posição inicial, delimitando o espaço acessível ao robô para a tarefa de limpeza. A função utiliza o algoritmo de busca em largura (BFS), que percorre a matriz a partir do ponto inicial e conta as células vizinhas, de 4 em 4, adjacentes que também podem ser limpas. A BFS é implementada usando uma estrutura de fila, importada de um arquivo auxiliar ".h", que permite explorar cada célula e sua vizinhança de forma sistemática, garantindo que todas as áreas alcançáveis e conectadas sejam contabilizadas. Dessa forma, a função determina a área total que o robô pode limpar de forma contínua, fornecendo uma medida precisa do espaço acessível para a limpeza a partir da posição inicial.

3.1 Lógica da Função

A função começa convertendo o texto XML do cenário em uma matriz de caracteres (matriz), na qual cada posição indica se é possível ou não realizar a limpeza. Caso a posição inicial fornecida (x0, y0) seja uma célula marcada como '0', a função retorna imediatamente 0, já que o robô não teria nenhum espaço a limpar a partir dali. Caso contrário, a função prossegue criando uma segunda matriz de controle (R), que será usada para registrar quais posições já foram visitadas ao longo do processo de busca. Essa matriz de controle é importante para evitar que o robô revisite áreas já contabilizadas, garantindo que cada célula seja contada apenas uma vez.

Para conduzir a busca, a função define uma fila (fila) para gerenciar as coordenadas de cada célula que deve ser explorada, implementando assim o algoritmo BFS. Inicialmente, a posição inicial é enfileirada e marcada como visitada, e a área é inicializada com o valor 1, representando a primeira célula que o robô alcança. Além disso, vetores de deslocamento (dx e dy) são configurados para possibilitar a movimentação nas quatro direções principais: cima, baixo, esquerda e direita. Dessa forma, a busca pode verificar todas as células adjacentes, expandindo-se de maneira controlada em torno da posição inicial.

Enquanto a fila não estiver vazia, a função processa cada célula na frente da fila, verificando suas células adjacentes. Cada uma dessas células vizinhas é validada para garantir que está dentro dos limites da matriz, que ainda não foi visitada, e que representa uma área limpável (ou seja, tem o valor '1' na matriz de cenário). Quando essas condições são satisfeitas, a célula é enfileirada, marcada como visitada na matriz R, e a área de limpeza é incrementada, indicando que o robô conseguiu acessar mais uma posição.

Ao final do processo de busca, a função libera a memória alocada para

as matrizes auxiliares matriz e R, evitando vazamentos de memória. Por fim, a função retorna o valor total da área, representado pela variável area, que contabiliza o número de células limpáveis conectadas à posição inicial. Dessa forma, a função `CalcularAreaLimpeza` simula o processo de limpeza do robô, permitindo que ele navegue por todas as áreas conectadas e acessíveis no cenário, fornecendo uma medida precisa da área total que o robô pode limpar de forma contínua a partir da posição dada.

3.2 Figura

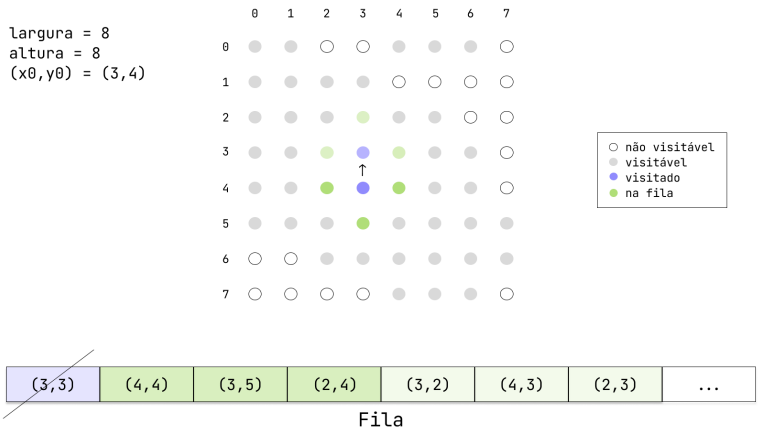

Figura 2: Exemplo figurativo do uso da Fila em uma BFS

3.3 Implementação

```
int CalcularAreaLimpeza(string& matriz_texto, int x0, int y0, int altura, int largura) {
    char** matriz = CriaMatriz(matriz_texto, altura, largura, 0);
    if (matriz[x0][y0] == '0') return 0;
    // Matriz R para controlar os pontos visitados
    char** R = CriaMatriz(matriz_texto, altura, largura, 1);

    // Definir a fila que aceitará valores do tipo pares de inteiros (coordenadas)
    ArrayQueue<std::pair<int, int>> fila(altura*largura);

    fila.enqueue({x0, y0});
    R[x0][y0] = 1;
    int area = 1;

    int dx[] = {-1, 1, 0, 0}; // Movimentos Verticais
    int dy[] = {0, 0, -1, 1}; // Movimentos Horizontais

    while (!fila.empty()) {
        pair<int, int> posicao_atual = fila.dequeue();

        int x = posicao_atual.first;
        int y = posicao_atual.second;

        for (int i = 0; i < 4; i++) {
            int novo_x = x + dx[i];
            int novo_y = y + dy[i];
            if (novo_x >= 0 && novo_x < altura &&   // Posições válidas
                novo_y >= 0 && novo_y < largura &&
                R[novo_x][novo_y] == 0 &&
                matriz[novo_x][novo_y] == '1') {

                fila.enqueue({novo_x, novo_y});
                R[novo_x][novo_y] = 1; // Marca o lugar da nova matriz como visitado
                area++;
            }
        }
    }
    DestroiMatriz(matriz, largura);
    DestroiMatriz(R, largura);
    return area;
}
```

3.4 Funções Auxiliares

As funções auxiliares de criação e destruição de matriz são essenciais para o processamento eficiente da área de limpeza e estão representadas abaixo:

```
char** CriaMatriz(string& texto, int linhas, int colunas, bool zeros) {
    char** matriz = new char*[linhas];
    for (size_t i = 0; i < linhas; i++) {
        matriz[i] = new char[colunas];
        for (size_t j = 0; j < colunas; j++) {
            if (!zeros) {
                matriz[i][j] = texto[i*colunas + j];
            } else {
                matriz[i][j] = 0;
            }
        }
    }
    return matriz;
}

void DestroiMatriz(char** &matriz, int altura) {
    for (size_t i = 0; i < altura; i++) {
        delete[] matriz[i];
    }
    delete[] matriz;
    matriz = nullptr;
}
```

4 Conclusão

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

- **Armazenamento de pares na fila**: Inicialmente, pensei em criar uma struct para inserir uma tupla de coordenadas na fila. No entanto, ao pesquisar mais sobre outras abordagens, descobri uma estrutura muito útil já disponível no C++: a `std::pair`.Esta facilita o acesso a cada um dos itens do par.
  `ArrayQueue<std::pair<int, int>> fila(altura*largura);`

- **Definição do tamanho da fila**: Esqueci de inicializar explicitamente o tamanho da fila, o que não foi necessário para a pilha. Por causa disso, logo no início dos testes, enfrentei problemas com a fila indicando estar cheia. Resolvi definindo o tamanho como `altura x largura` unidades ao inicializar a fila, garantindo que a fila conseguisse, no pior dos casos, alocar todas as coordenadas da matriz.

- **Problemas ao destruir as Matrizes**: Ao realizar a deleção das matrizes no final da função `CalculaAreaLimpeza`, tudo funcionava perfeitamente. Criei funções auxiliares para melhorar a estrutura do código, mas, após essa reorganização, o código deixou de funcionar no VPL. Como resultado, acabei retornando à implementação anterior. No entanto, no relatório, incluí a implementação alternativa e descrevi a função de forma detalhada.

- **Problemas ao capturar as Matrizes**: Adicionei algumas linhas de código para verificar o valor da variável `ultimo indice`, pois, embora as saídas estivessem corretas ao rodar no ambiente de execução, elas não funcionavam corretamente no ambiente de avaliação. A solução foi usar a função `find last of` para localizar o último dígito '0' ou '1' que pertencesse à matriz e, em seguida, comparar os retornos para identificar qual realmente seria o último índice relevante. Isso garantiu uma delimitação precisa da matriz e resolveu o problema.

5 Referências

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

cppreference.com: **vector**.
Em: `https://en.cppreference.com/w/cpp/container/vector/vector`.
Acesso em: 2 nov. 2023.

GeeksforGeeks: **Breadth-first search or BFS for a graph**.
Em: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/.
Acesso em: 2 nov. 2023.

Youtube/@Reducible: **Breadth First Search (BFS): Visualized and Explained**
Em: https://www.youtube.com/watch?v=xlVX7dXLS64&t=150s
Acesso em: 2 nov. 2023.

GeeksforGeeks: **pair in cpp stl**.
Em: https://www.geeksforgeeks.org/pair-in-cpp-stl/.
Acesso em: 2 nov. 2023.

cplusplus: **Files in C++**.
Em: http://www.cplusplus.com/doc/tutorial/files/.
Acesso em: 2 nov. 2023.

cplusplus: **The C++ Standard Library reference**.
Em: http://www.cplusplus.com/reference/string/string/.
Acesso em: 2 nov. 2023.