Projeto 1 - Relatório

Gustavo Beretta Gonçalves
Benedek Uzoma Kovacs

1. Introdução

O projeto consiste no desenvolvimento de um algoritmo que leia arquivos XML específicos. Cada um desses arquivos contém algumas matrizes binárias que correnspondem à área que um robô de limpeza deve percorrer. Cada elemento 1 da matriz corresponde a uma unidade de área que deve ser limpa, enquanto cada elemento 0 corresponde a uma unidade de área que já está limpa. Os arquivos são organizados e divididos por tags, que identificam diferentes cenários, matrizes e dimensões.

O primeiro desafio do projeto é validar a implementação dessas tags, afinal, elas devem obedecer uma ordem de abertura e fechamento, além de que todas as tags abertas devem ser fechadas, assim como não se pode fechar uma tag que não foi aberta.

O segundo desafio é calcular efetivamente à área que o robô deverá limpar. O robô inicia em uma posição específica dada pelo arquivo e ele não deve passar por áreas já limpas, ou seja, deve-se considerar apenas os elementos 1 que estão no componente conexo onde o robô inicia.

2. Desenvolvimento

- **1° Desafio:**

A solução para o primeiro desafio consiste em um algoritmo de validação das tags do arquivo analisado (**bool validacao(std::string nome_do_arquivo)**), baseado na implementação de uma pilha. Basicamente, essa função cria uma pilha (**pilha_de_tags**) que armazena o identificador de todas as tags abertas e utiliza isso para saber se elas estão sendo fechadas corretamente

A variável **char caractere_atual** recebe o caractere em análise no momento, conforme o algoritmo percorre o arquivo através da estrutura **while**.

A variável **bool lendo_tag** é inicializada com valor **false** e recebe o valor **true** quando o caractere em análise faz parte do identificador da tag.

A variável **bool tag_de_fechamento** é inicializada com valor **false** e recebe o valor **true** quando a tag em análise é de fechamento.

A variável **std::string identificador** armazena o identificador da tag que está sendo analisada.

Primeiramente, o arquivo é aberto. Caso não seja possível, a função já retorna **false**, o que acarretará em uma mensagem de erro posteriormente.

Enquanto o arquivo não chegar ao final, a variável **caractere_atual** será atualizada com o valor do novo caractere a ser analisado.

A primeria verificação feita é para saber se o caractere representa o término da declaracão de uma tag, ou seja, se **caractere_atual == '>'**. Caso positivo, verifica-se se esse tag é de fechamento, ou seja, se possui um '/' antes do identificador, através da variável **tag_de_fechamento**.

Se essa verificação também for positiva, é necessário conferir se a pilha não está vazia, pois, caso esteja, significa que estamos fechando uma tag que não foi aberta, o que resulta em uma mensagem de erro. Caso a pilha não esteja vazia, fazemos uma última verificação para analisar se o o topo da pilha (identificador da última tag aberta) é igual ao identificador da tag que está sendo fechada. Caso negativo, é exibida uma mensagem de erro. Caso positivo, o último elemento da pilha é desempilhado, o que significa que a tag foi fechada com sucesso. Além disso, as variáveis voltam ao esatdo que foram declaradas.

Se a tag não for de fechamento, o identificador é empilhado e as variáveis também resetam.

Caso a primeira verificação não seja verdadeira, partimos para analisar se estamos percorrendo o identificador de uma tag, ou seja, se **lendo_tag == true**. Caso positivo, verifica-se se o caractere sendo analisado é igual a '/', o que implicaria em atualizar a variável **tag_de_fechamento** para **true**. Caso negativo, apenas adicionamos o caractere analisado à variável **identificador**.

Caso a verificação de **lendo_tag** seja falsa, verificamos se **caractere_atual == '<'**, o que representa o início da declaração de uma tag, implicando na atualização da variável **lendo_tag** para **true**.

Após percorrer todos os caracteres do arquivo, a função ainda verifica se a pilha está vazia, pois, se não estiver, significa que há tags que não foram fechadas, o que resulta em uma mensagem de erro também.

Primeiro caractere

arquivo >> caractere_atual
Próximo caractere

caractere_atual == '>'
(fim de declaração de tag)

Sim

Não

tag_de_fechamento == true

Não

Sim

Tudo certo

lendo_tag == true
(caractere dentro do identificador da tag)

Não

Sim

caractere_atual == '<'
(início de declaração de tag)

Sim

lendo_tag = true

Não

caractere_atual == '/'
(declaração de tag de fechamento)

Não

Sim

caractere_atual
adicionado a
identificador

tag_de_fechamento = true

identificador empilhado e variáveis resetadas

pilha_de_tags.empty()

Não

Sim

pilha_de_tags.top() == identificador

Não

Erro

Sim

Não

Sim

Desempilha e reseta variáveis

pilha_de_tags.empty()

Não

Sim

arquivo.eof()
(último caractere)

Não

Sim

- **2° Desafio:**

O 2° desafio começa a ser resolvido ainda na função **main**.

Primeiramente, o conteúdo do arquivo é salvo na variável **string_conteudo** do tipo string. Em seguida, é calculado o número de cenários presentes no arquivo análisado, já que, cada arquivo pode ter um número diferente de cenários e cada cenário pode ter uma resolução diferente. O número de cenários fica armazenado na variável **n_cenarios** e é encontrado através da busca e contagem das tags “<cenário>” dentro do conteúdo do arquivo.

Então, o código entra em um loop **for**, que representa a resolução individual de cada cenário.

Ele inicia salvando a posição da primeira tag “**<cenario>**” em **startPos** para utilizar como parâmetro nas operações, assim é chamada a função **retorna_info** para resgatar os de 'nome', 'altura', 'largura', 'robo_x', 'robo_y' do cenário analisado no arquivo.

Essa função recebe como parâmetro **string_conteudo** (o próprio arquivo em forma de string), **pos_cenario** (o valor da posição na string em que as informações do cenário em execução está), **tag** (especifica qual tag do .xml a função está buscando) e **var** (auxilia na especificação da informação buscada). Na string **string_conteudo**, se baseando na posição **pos_cenario**, ele busca pela primeira ocorrência da **<tag>**. Se encontrar, busca pela primeira ocorrência de **<var>***,* se também encontrar, ela passa a buscar o fechamento **</var>**. Se todas as condições forem atendidas, a função extrai as informações entre **<var>** *e* **</var>**, armazenando-a na variável **dado**, que é retornada pela função.

Utilizando o mesmo método, agora resgatamos a matriz em string do arquivo .xml e armazenamos na variável **matriz_string**. Em seguida, utilizamos a função **gerador_matriz** para criar uma matriz de inteiros e atribuí-la a variável **matriz_cenario** e outra contendo apenas zeros em suas casas, atribuindo ela a variavel **matriz_zerada**.

Primeiramente a função aloca memória para a matriz, criando assim a variável **matriz**, um array de ponteiros que possui como tamanho o argumento **altura** recebido. Cada elemento do array é um ponteiro para um array de inteiros que representam as linhas da matriz. Também salva a **matriz_string** em uma nova variável string **matriz_valores**, para que possa haver uma melhor manipulação dos dados. Após isso, através da biblioteca **<algorithm>** é utilizado o método **erase**, em combinação com o argumento **remove** para remover todos os caracteres de quebra de linha (**\n**) da string **matriz_valores**, sobrando assim apenas os números na string para serem manipulados. Em seguida, finalmente, um **for** loop é iniciado para preencher a matriz, ele percorre todas as linhas e aloca um array de inteiros com o tamanho do argumento **largura** recebido em cada uma delas. Em seguida, outro **for** é executado para percorrer cada coluna da matriz. Dentro desse loop, o valor correspondente na string **matriz_valores** é obtido usando a fórmula **(i * largura + j)**, onde **i** representa o índice da linha, **j** representa o índice da coluna atual e **largura** é um dos argumentos recebidos. O resultado é armazenado na variável **valor**, que é

do tipo char. Assim, o valor da matriz é atribuído de acordo com a condição booleana **zerada**. Se **zerada** for verdadeiro, o valor é definido como 0. Caso contrário, o valor é convertido de caractere para inteiro subtraindo o caractere “0”, que tem um valor numérico correspondente. A condição booleana existe para que a mesma função possa ser utilizada para gerar tanto a matriz do cenário e quanto a matriz zerada que é utilizada pelo robô, assim otimizando o código. Por fim, após todos os valores da matriz serem atribuídos em seus respectivos espaços, a variável **matriz** é retornada pela função.

Voltando para a função **main**, agora com todos os dados necessários, a operação de limpeza do robô é iniciada por meio da função **resolucao** e após ela realizar todas as operações é retornado o valor da quantidade de casas que foram limpas pelo robô, sendo esse valor atribuído à variável **resultado**.

Essa função tem como finalidade realizar o cálculo da área que deverá ser limpa, contando o número de “1” presentes na área onde o robô se encontra. A função utiliza como parâmetros duas matrizes de inteiros, sendo uma a matriz do arquivo XML **matriz_cenario**, que contém a área que deverá ser calculada, e outra que possui as mesmas dimensões da primeira, porém sendo preenchida por zeros **matriz_zero**, e também seus valores de altura e largura e as coordenadas iniciais do robô. Inicialmente, verifica-se se as coordenadas onde o robô se encontra na **matriz_cenario** contém o valor “1”. Em caso negativo, a coordenada não é adicionada na fila, pois o robô se encontra em uma região “limpa”, de modo que a área seja zero. Em caso afirmativo, atribui-se o valor “0” na coordenada da **matriz_cenario** e o valor “1” para a mesma coordenada na **matriz_zero**, e também adiciona a coordenada em uma fila. Atribuindo o valor “0” para a posição na **matriz_cenario** torna a coordenada “limpa”, impedindo que a mesma coordenada seja adicionada mais de uma vez na fila. Após isso, o algoritmo passa por um condicional **while**, que é executado enquanto a fila não está vazia. Dentro do **while**, inicialmente remove-se a primeira coordenada da fila e atribui seu valor a uma variável **first**. Após isso, calcula-se as 4 coordenadas vizinhas de **first**, atribuindo seus valores a 4 estruturas do tipo **Coordenada**. Depois disso, as 4 coordenadas calculadas passam por algumas condições, para verificar se todas são válidas, ou seja, se estão contidas na matriz. Em caso afirmativo, verifica-se se os valores da **matriz_cenario** nas coordenadas são iguais a “1”, buscando pelas áreas que devem ser limpas e, caso sejam, o valores da **matriz_zero** nas coordenadas são alterados para “1”, e os valores da **matriz_cenario** nas coodendas são alterados para “0”, de modo a “limpar” a área, garantido que posição não seja contada mais de uma vez. Além disso, tais coordenadas também são adicionadas na fila, para que futuramente seja realizada a verificação de “1” em seus arredores. No final da função, a **matriz_zero** é percorrida, e o número de “1” é contado, adicionando “1” na variável **resultado**, que é o valor retornado pela função, referente a área a ser limpa pelo robô. Dessa forma, as coordenadas que possuem valor igual a “1”, uma a uma, por ordem de adição na fila, tem seus arredores verificados, checando se possuem valores iguais a “1” e, caso possuam, tais casas vizinhas tem sua coordenada adicionada à fila, de modo a passarem pela mesma verificação, sendo possível

então realizar a contagem do componente conexo, pois todas as casas que possuam valores iguais a “1” e estejam conectadas são computadas. Ao final do algoritmo, a **matriz_zero** possuirá a área de valores “1” onde o robô foi inicialmente alocado, e a **matriz_cenario** terá esta mesma área apagada, ou seja, preenchida por “0”.

Para terminar, na função **main**, é printado na tela o **nome** e **resultado** do cenário atual, e é utilizado o método **find** para encontrar a tag **</cenario>** de fechamento do cenário, assim pegando sua posição e adicionando na variável **aux**, que irá servir como posição base para o próximo loop de operações. Após todos os loops serem concluídos e printados, o programa termina sua execução.

➔ **Explicação com figuras:**

Supondo que a matriz do cenário que desejamos calcular seja uma matriz 5x5. O algoritmo cria uma matriz de mesma ordem, porém preenchida com zeros. Dessa forma, teremos duas matrizes para auxiliar no cálculo da área do componente conexo. Vamos supor também que a coordenada que o robô inicia seja (3, 3), na linha 3 e coluna 3, no valor ‘1’ marcado em vermelho na imagem abaixo.

matriz_cenario

0	0	0	0	0
0	1	0	1	0
0	1	1	1	0
0	1	1	1	1
0	0	0	0	0

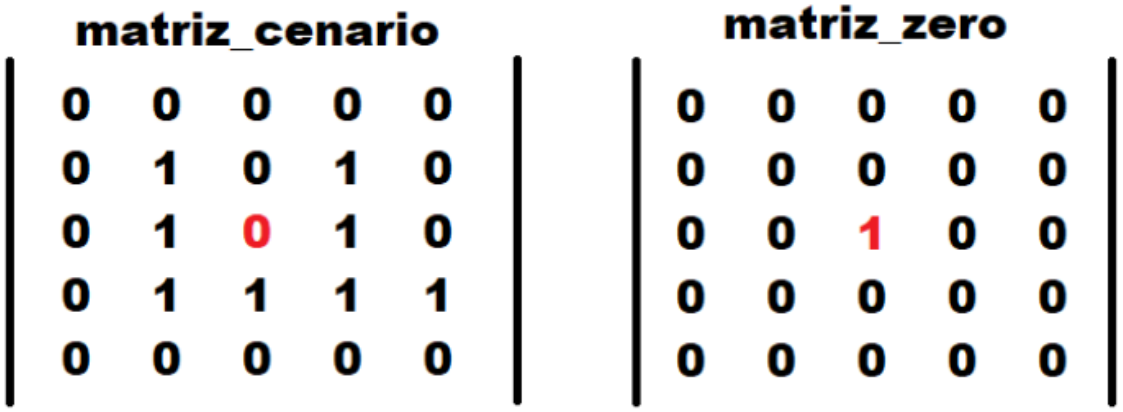
matriz_zero

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

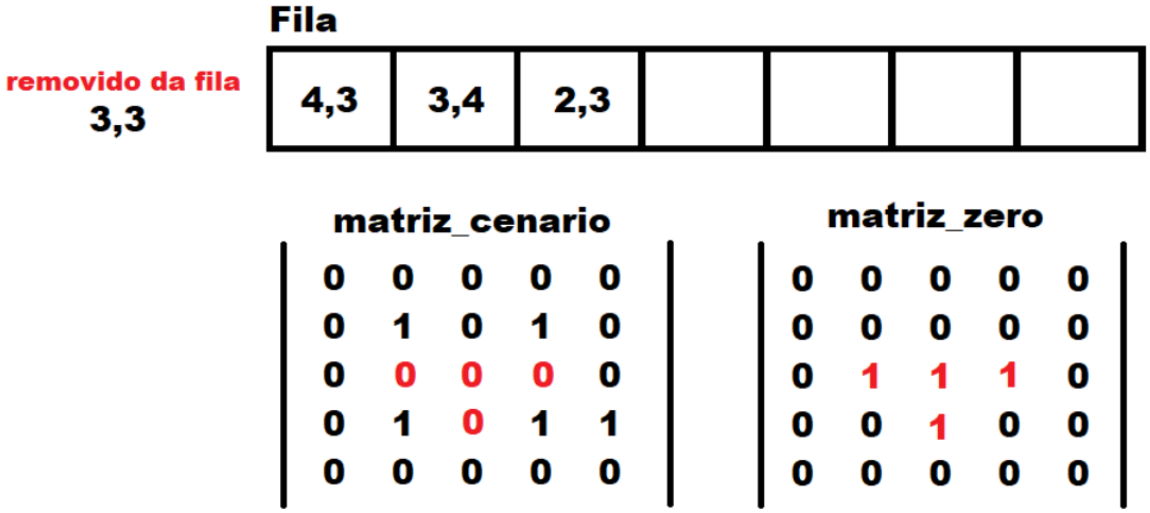
A coordenada inicial do robô será adicionada à fila, e o valor da matriz_cenario nesta coordenada será alterado para ‘0’, e o valor da matriz_zero será alterado para 1.

Fila

3,3						



Após isso, o algoritmo irá checar as casas vizinhas da primeira coordenada da fila, que será removida da mesma. Caso tais casas possuam o valor ‘1’, suas coordenadas são adicionadas à fila, para que sejam submetidas a mesma verificação, e a matriz_cenario e também a matriz_zero tem seus valores alterados nestas coordenadas, para ‘0’ e ‘1’, respectivamente.



Dessa forma, todas as casas que possuem valor ‘1’ são adicionadas à fila, para que seus arredores sejam verificados, arredores estes que, caso também possuam valor, serão adicionados à fila, para que passem pela mesma verificação.

3. Conclusão

Sem dúvidas, o principal problema encontrado durante a implementação do projeto foi a falta de domínio da linguagem C++. É perceptível que ainda falta prática para eu me sentir confortável ao utilizar essa linguagem. Vários algoritmos tiverem que ser testados várias vezes até funcionarem da maneira correta como eu havia planejado.

Além disso, a modelagem da resolução do segundo desafio também foi um pouco dificultosa. Tive que passar alguns bons minutos pensando em como seria a solução deste problema.

De forma geral, o projeto foi muito interessante e serviu para eu aprender a trabalhar com arquivos em C++, já que eu tive que ir atrás de entender como manipulá-los.

4. Referências

CFBCURSOS. **Curso de C++ #51 - Operações com arquivos (ifstream) - Parte 2**. Youtube, 2017. Disponível em: <https://www.youtube.com/watch?v=Tczymt0OkYo&ab_channel=CFBCursos>. Acesso em: 16 de Outubro de 2023.

CPLUSPLUS.COM. **Input/output with files**. Disponível em: <https://cplusplus.com/doc/tutorial/files/>. Acesso em: 16 de Outubro de 2023.