RELATÓRIO PROJETO 1

Alunos: Luan Diniz Moraes *(21204000)* e Leonardo Guimaraes de Melo Brito *(21200896)*
Matéria: Estrutura de Dados (INE5408, turma 03208/A),
30 de outubro de 2022

Observação: Nesse projeto foram utilizados pilha encadeada *(*linked_stack.h) e a fila encadeada (linked_queue.h).

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include "./linked_stack.h"
#include "./linked_queue.h"
using namespace std;

bool stack_verification (int pos, std::string linha, structures::LinkedStack<std::string> &pilha, structures::LinkedQueue<string> &png_info);
void file_verification(string arquivo, structures::LinkedQueue<string> &png_info);

typedef struct {
    int x;
    int y;
} coordenadas;
```

Declaração de bibliotecas, arquivos, funções utilizadas e também a definição de um novo tipo (*coordenadas*).

Primeiro problema: validação do arquivo XML.

Para resolver esse problema, utilizou-se a pilha. Primeiramente, o arquivo XML é aberto utilizando a biblioteca *fstream*, o nome do arquivo é armazenado no *xmlfilename*, enquanto a string *line* irá armazenar uma linha inteira do arquivo (uma por vez, através da função *getline()*).

```cpp
cin >> xmlfilename;  // entrada

ifstream xmlfile;
xmlfile.open(xmlfilename);

if (xmlfile.is_open()) {
    while (!xmlfile.eof() && !error) {
        getline(xmlfile,line);
        if (line[0] == '0' || line[0] == '1') {
            continue;
        }
        for (int i = 0; i < line.length(); i++) {
            if (line[i] == '<') {
                error = stack_verification(i, line, pilha, png_info);   // Checa o que vai fazer com relacao a pilha :D
            }
        }
    }
    xmlfile.close();
} else {
    throw out_of_range("Arquivo xml não existente");
}
```

Após o usuário digitar o nome do arquivo a ser lido ele é aberto (é feita uma verificação de que o arquivo existe) e então inicia-se um *while* que a cada iteração lê e armazena a próxima linha do arquivo em *line*, o loop dura até ser alcançado o fim do arquivo ou algum erro ser detectado.

Line é um vetor, portanto o algoritmo percorre todas as suas posições em busca do char '<', que indica que uma tag está abrindo (ou fechando). Ao identificá-lo a função *stack_verification()* é chamada. Esse procedimento primeiramente irá procurar o char '>' no *line,* pois assim é possível obter a tag (substring dos índices em que '<' e '>' foram encontrados) e também saber se é uma tag de abertura ou uma tag de fechamento (índice do '<' + 1 é '\').

A tag é armazenada na string *result* e, caso seja de abertura, ela é adicionada na pilha, dessa maneira saberemos se o sistema hierárquico é respeitado. (Após isso, se a tag for "height", "width" ou "name", o valor após ela é armazenada na fila *png_info*, isso será usado no segundo problema e é possível fazer isso sem nenhum tratamento adicional pois essas tags aparecem sempre na mesma ordem).

Caso a tag seja de fechamento, é retirado um elemento da pilha e, se for diferente da tag armazenada em *result*, significa que o arquivo não é válido, portanto a função irá retornar *true.*

```
if (xmlfile.is_open()) {
    while (!xmlfile.eof() && !error) {
        getline(xmlfile,line);
        if (line[0] == '0' || line[0] == '1') {
            continue;
        }
        for (int i = 0; i < line.length(); i++) {
            if (line[i] == '<') {
                error = stack_verification(i, line, pilha, png_info);   // Checa o que vai fazer com relacao a pilha :D
            }
        }
    }
    xmlfile.close();
} else {
    throw out_of_range("Arquivo xml não existente");
}

if (!pilha.empty()) {
    error = true;
}

if (!error) {  //Exercicio 2
    file_verification(xmlfilename, png_info);
} else {
    cout << "error" << std::endl;
}

return 0;
```

Após identificar o erro, o loop é quebrado e é imprimido no console "error".

```cpp
bool stack_verification (int pos, string linha, structures::LinkedStack<string> &pilha, structures::LinkedQueue<string> &png_info) {
    bool deu_erro = false;
    int pos_fim = pos;
    while (linha[pos_fim] != '>') {
        pos_fim++;
    }
    if (linha[pos+1] != '/') {
        string result = linha.substr(pos+1, pos_fim-pos-1);
        pilha.push(result);
        if (result.compare("height") == 0 || result.compare("width") == 0 || result.compare("name") == 0) {
            int pos_fim_fim = pos_fim;
            while (linha[pos_fim_fim] != '<') {
                pos_fim_fim++;
            }
            string data = linha.substr(pos_fim+1, pos_fim_fim-pos_fim-1);
            png_info.enqueue(data);                    //Vai sair nome,altura e largura nessa ordem. (FIFO)
        }

    } else {
        string result = linha.substr(pos+2, pos_fim-pos-2);
        string comp = pilha.pop();
        if (comp.compare(result) != 0) {
            deu_erro = true;
        }
    }
    return deu_erro;
}
```

A função *stack_verification()*

Segundo Problema: contagem de componentes complexos.

O algoritmo para resolver este problema está inteiramente dentro da função *file_verification()*, que só é executada após o algoritmo do primeiro problema terminar de executar sem erros.

Aqui, será usado o tipo *coordenadas* para o enqueue e dequeue da fila principal, basicamente é um tipo que possui dois inteiros, *coordenadas.x* e *coordenadas.y* para armazenar as "coordenadas" (o índice) de um elemento da matriz.

Um *while* é executado enquanto a fila *png_info* não estiver vazia, a cada iteração é dado dequeue três vezes nela, obtendo as informações de altura, largura e nome da imagem, que foram obtidos durante o *stack_verification()*. Além disso, o começo do loop faz *label = 0,* e cria duas matrizes dinamicamente, E (armazena os valores da imagem) e R(armazena as posições já visitadas). Essas matrizes são deletadas no final do loop.

```cpp
while (!png_info.empty()) {
    label = 0;
    png_name = png_info.dequeue();
    height = stoi(png_info.dequeue());
    width = stoi(png_info.dequeue());

    int **E;
    int **R;
    E = new int * [height];
    R = new int * [height];

    for (int i = 0; i < height; i++) {
        E[i] = new int[width];
        R[i] = new int[width];
    }
```

Antes do loop o arquivo é aberto novamente.

Então, é percorrida a imagem e armazenados seus valores na matriz E, concomitantemente, são atribuídos a todos os elementos da matriz R o valor 0.

```
getline(xmlarquivo,linha);
while ((linha[0] != '0' && linha[0] != '1') ) {
    getline(xmlarquivo,linha);
}

int i = 0;
while (linha[0] == '0' || linha[0] == '1') {
    for (int j = 0; j < linha.length(); j++) {
        E[i][j] = (int)linha[j] - 48;
        R[i][j] = 0;
    }
    getline(xmlarquivo,linha);
    i++;
}
```

Após isso, o código irá buscar e marcar os componentes conexos. Primeiramente, há um loop que irá procurar um novo componente conexo, ou seja, vai percorrer a matriz E inteira em busca de um elemento com valor 1 e que sua posição respectiva na matriz R o elemento seja 0. Cada vez que esse padrão for encontrado, aumentamos o valor do *label* em um e também adicionamos as coordenadas do elemento na fila principal, pois teremos garantidamente um novo componente conexo, devido à lógica que se segue (outro loop antes de ir para a próxima iteração):

- Enquanto a fila não estiver vazia, executar dequeue.
- Marcar as coordenadas que vieram do dequeue na matriz R.
- Verifica a vizinhança da coordenada (ou seja, (x-1,y),(x+1,y),(x,y+1),(x,y-1)) em busca do valor 1 que seu elemento respectivo na matriz R seja igual a 0..
- A coordenada em que 1 for encontrada é marcada (com o valor do label atual) na matriz R, além de ser mandada para a fila (enqueue).
- O processo se repete.

Dessa forma, todos os elementos do componente conexo serão marcados e então se inicia a próxima iteração do loop.

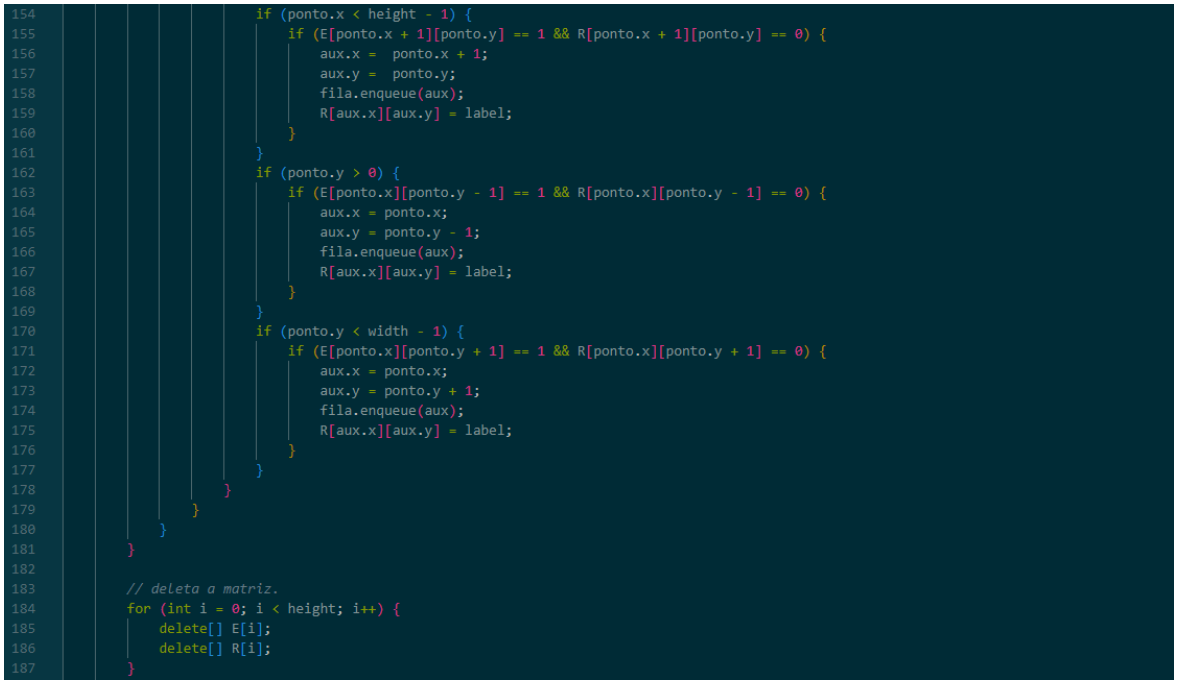
Em dado momento, a matriz E já terá sido percorrida por completo, então o algoritmo printa o nome da imagem e seu label, depois passando para a iteração seguinte do loop principal, analisando a próxima imagem.

A seguir a função *file_verification()* e o algoritmo descrito acima (na íntegra):

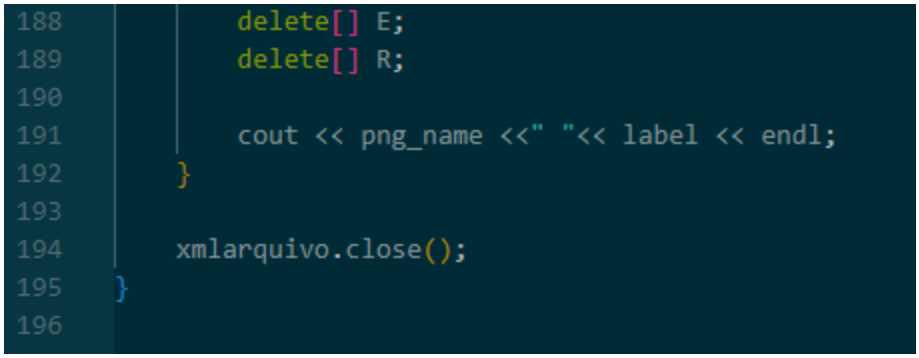
```
void file_verification(string arquivo, structures::LinkedQueue<string> &png_info) {
    structures::LinkedQueue<coordenadas> fila;
    ifstream xmlarquivo;
    xmlarquivo.open(arquivo);

    string png_name;
    string linha;
    int height;
    int width;
    int label;

    while (!png_info.empty()) {
        label = 0;
        png_name = png_info.dequeue();
        height = stoi(png_info.dequeue());
        width = stoi(png_info.dequeue());

        int **E;
        int **R;
        E = new int * [height];
        R = new int * [height];

        for (int i = 0; i < height; i++) {
            E[i] = new int[width];
            R[i] = new int[width];
        }

        getline(xmlarquivo,linha);
        while ((linha[0] != '0' && linha[0] != '1') ) {
            getline(xmlarquivo,linha);
        }
```

```
120         int i = 0;
121         while (linha[0] == '0' || linha[0] == '1') {
122             for (int j = 0; j < linha.length(); j++) {
123                 E[i][j] = (int)linha[j] - 48;
124                 R[i][j] = 0;
125             }
126             getline(xmlarquivo,linha);
127             i++;
128         }
129
130         for (int i = 0; i < height; i++) {
131             for (int j = 0; j < width; j++) {
132                 //if novo componente conexo => fila.enqueue()
133                 if (E[i][j] == 1 && R[i][j] == 0) {
134                     coordenadas aux;
135                     aux.x = i;
136                     aux.y = j;
137                     fila.enqueue(aux);
138                     label = label + 1;
139                     R[i][j] = label;
140
141                     while(!fila.empty()) {
142                         coordenadas aux;
143                         coordenadas ponto = fila.dequeue();
144                         R[ponto.x][ponto.y] = label;
145
146                         if (ponto.x > 0) {
147                             if (E[ponto.x - 1][ponto.y] == 1 && R[ponto.x - 1][ponto.y] == 0) {
148                                 aux.x =  ponto.x - 1;
149                                 aux.y =  ponto.y;
150                                 fila.enqueue(aux);
151                                 R[aux.x][aux.y] = label;
152                             }
153                         }
```

Na linha 133 começa o flood fill.



```
                    if (ponto.x < height - 1) {
                        if (E[ponto.x + 1][ponto.y] == 1 && R[ponto.x + 1][ponto.y] == 0) {
                            aux.x =  ponto.x + 1;
                            aux.y =  ponto.y;
                            fila.enqueue(aux);
                            R[aux.x][aux.y] = label;
                        }
                    }
                    if (ponto.y > 0) {
                        if (E[ponto.x][ponto.y - 1] == 1 && R[ponto.x][ponto.y - 1] == 0) {
                            aux.x = ponto.x;
                            aux.y = ponto.y - 1;
                            fila.enqueue(aux);
                            R[aux.x][aux.y] = label;
                        }
                    }
                    if (ponto.y < width - 1) {
                        if (E[ponto.x][ponto.y + 1] == 1 && R[ponto.x][ponto.y + 1] == 0) {
                            aux.x = ponto.x;
                            aux.y = ponto.y + 1;
                            fila.enqueue(aux);
                            R[aux.x][aux.y] = label;
                        }
                    }
                }
            }
        }

        // deleta a matriz.
        for (int i = 0; i < height; i++) {
            delete[] E[i];
            delete[] R[i];
        }
```



```
        delete[] E;
        delete[] R;

        cout << png_name <<" "<< label << endl;
    }

    xmlarquivo.close();
}
```

Ao fim da execução, o arquivo é fechado.

Dificuldades na resolução dos problemas

As únicas dificuldades que tivemos durante a elaboração das soluções foram relacionadas a corrigir erros do código. Por vezes o erro se dava em um ou outro caractere fora de lugar, ou em lugares trocados, como por exemplo, alguns dos erros que estávamos tendo foi resolvido após percebermos que um for dentro de um for estava acessando o índice *i* ao invés do índice *j*.

Além disso, um pequeno erro na implementação da linked_stack.h demorou para ser encontrado (a função pop() não estava fazendo size_ = size_ -1).

Por fim, os erros de segmentação provavelmente foram os mais demorados de serem corrigidos e eram baseados em um erro de lógica no algoritmo do segundo problema. Primeiro, nós não tínhamos colocado condicionais antes de verificar as vizinhanças da coordenada (x - 1, x + 1 por exemplo podem resultar em -1 e height + 1, o que dá segmentation fault.) e depois colocamos o comparador errado (if x <= height - 1, quando deveria ser x < height - 1).

Para tentar localizar todos esses erros, nós colocamos prints dentro do código para ver até onde ele rodava. (O VSCode foi especialmente útil para localizar os erros de segmentação).