Centro Universitário Barão de Mauá
Curso de Bacharelado em Ciência da Computação

RELATÓRIO FINAL DE ESTÁGIO SUPERVISIONADO
MÓDULO II – LABORATÓRIO DE ESTRUTURA DE DADOS

Nome complete do Aluno – Gustavo B. Moreira 2069049

Ribeirão Preto
2022

RELATÓRIO DE ESTÁGIO SUPERVISIONADO

Relatório técnico-científico apresentado ao Curso de Ciência da Computação do Centro Universitário Barão de Mauá, como parte dos requisitos para obtenção do título de Bacharel em Ciência da Computação.

Docente Orientador de Estágio: Prof. Dr. Felipe Carvalho Pellison

SUMÁRIO

Ribeirão Preto
2022

1. RESUMO

Utilizando os conceitos de estrutura de dados aprendidos nesse semestre, o objetivo é criar um algoritmo capaz de encontrar múltiplos caminhos que levem ao final do jogo dentro de um labirinto, mapeando-os e exibindo ao final do programa, quantos caminhos diferentes conseguem levar a solução desejada, armazenando-os em uma estrutura de pilhas.

1. **Palavras-chave:** Estrutura de Dados. Pilha. Labirinto. Vários Caminhos. Backtracking.

2. INTRODUÇÃO

O problema enfrentado nesse projeto, será criar um algoritmo que consiga encontrar todos os possíveis caminhos que levem até o final de um labirinto, porém, utilizando algum tipo de estrutura de dados para o armazenamento dessas soluções.

Esse desafio de encontrar a saída de um labirinto é bem utilizado na computação para estudos em diversas áreas, como por exemplo, no estudo de IA´s, algoritmos recursivos, entre alguns outros, apesar da maioria, apenas tentar encontrar 1 única solução para esse labirinto, sendo ela geralmente, a mais “rápida”.

Dessa forma, meu objetivo é criar um algoritmo que consiga encontrar todos os caminhos que levem até o final do jogo, mapeando-os e demonstrando os passos diferentes que levam até a mesma saída.

Por mais que seja um problema bem comum e estudado, como dito anteriormente, a maioria dos códigos apenas se preocupam em encontrar e demonstrar as soluções mais curtas ou em alguns caso, as piores soluções, além de utilizarem outros meios para o armazenamento desses dados. Portanto, esse será o diferencial do meu projeto, encontrando todos os caminhos e os armazenando em uma pilha.

Na primeira semana do projeto, busquei em diversos fóruns, maneiras diferentes e desenvolvidas para a solução do meu problema, além disso, criei o meu arquivo classe, com as funções básicas que julgo serem necessárias para a finalização do trabalho. Na 2° e 3° semana, utilizei novamente pesquisas de campo para encontrar alguns documentos que me auxiliasse nessa ideia, e baseando-me em artigos de backtracking, percebi que estava no caminho correto. A partir das últimas semanas, busquei inúmeras formas de implementar o meu algoritmo com base nos meus estudos de recursão e backtracking, tudo em conjunto do orientador do curso e ao final, criei um código que satisfez minhas expectativas e conclui o projeto.

3. REFERENCIAL TEÓRICO OU FUNDAMENTAÇÃO TEÓRICA

4. 1. LABIRINTO

A ideia de se resolver enigmas como labirinto é bem antiga dentro da programação, onde com o passar do tempo, inúmeros testes e métodos diferentes foram surgindo para tentar resolver esse problema. Porém, o principal problema foi mudando durante essas etapas, já que antes, bastava resolver o labirinto que já era suficiente, porém, os programadores queriam ir além, encontrando métodos mais rápidos, mais eficientes, com mais durabilidade e consistência em seus códigos. Com isso, os métodos foram evoluindo ao ponto em que, ou as diferenças são extremamente grandes, com resultados mais rápidos e diversos, ou mais simples, com apenas alguns segundos de diferença de outras técnicas. Por conta dessas diferenças, novamente, a vontade de encontrar algum método superior aos outros foi considerada, e baseado em métodos extremamente complexos como de Processamento Digital de Imagens, Watershed, Esqueletização e Morfologia Matemática, um estudo foi feito por João Henrique Laranja Capucho e Karin Satie Komati, do Instituto Federal do Espírito Santo (Ifes) Campus Serra, onde a conclusão final foi:

> "O objetivo deste trabalho foi o de comparar 3 diferentes técnicas de PDI para solucionar o problema de labirintos, sendo eles Watershed, Esqueletização e Morfologia Matemática. Ao final da comparação qualitativa, analisou-se que a abordagem via Morfologia Matemática obteve o melhor resultado entre as técnicas, em todos os cenários de testes. Através da separação em regiões, de forma a gerar melhores resultados, a técnica teve 100% de acertos para Labirintos Perfeitos e com duas rotas de saída, e foi a única a obter sucesso pela mistura de duas ou mais regiões. Portanto, podemos afirmar que a Morfologia obteve os melhores resultados para o estudo. Também foi possível avaliar que o pior resultado foi a abordagem via Esqueletização, que gera muitas ramificações em suas rotas."

Porém, por mais complexo que esses métodos pareçam, até mesmo eles utilizam de um conjunto de estrutura de dados, como pilhas, filas, arvores, com um famoso método chamado backtracking, que será o método utilizado nesse trabalho.

5. 2.ESTRUTURA DE DADOS:

Utilizando todo o conhecimento adquirido durante o curso, diversas ideias diferentes de como armazenar informações do código de uma forma segura

surgiram em minha mente e baseado nos ensinamentos sobre estrutura de dados, decide utilizar o “meio” pilha nesse trabalho. Por mais que pareça simples, o uso de pilhas é extremamente útil e até mais simples de ser implementado, onde seu uso e eficiência vão depender da necessidade de cada código e em conjunto do método de backtracking, também utilizado nesse trabalho, torna a pilha, uma ótima escolha, como bem apresentado em um blog feito por Lorenzo Maturano, utilizando citações do livro *The Algorithm Design Manual*, de *Steven Skiena* e anotações das aulas ministradas pelos professores Aldo von Wangenheim e Alexandre Gonçalves Silva:

> “Pilhas são muito usadas em praticamente todos os níveis de um sistema de computação.
>
> Em nível de arquitetura, pilhas são usadas para manipular interrupções https://www.geeksforgeeks.org/rat-in-a-maze-backtracking-2/https://www.geeksforgeeks.org/rat-in-a-maze-backtracking-2/Em nível de linguagem de programação, pilhas são usadas para armazenar informações sobre subrotinas de um programa (ex: C++ call stack).
>
> Em nível de programação, pilhas podem ser usadas para reverter strings, implementar um feature de undo/redo, ou até modelar um problema da vida real como a Torre de Hanói.”

6. 3. BACKTRACKING:

Para resolver o problema de encontrar os caminhos possíveis de um labirinto, o método backtracking é excelente. Esse algoritmo é baseado em sucessivas recursões no programa, sempre buscando satisfazer alguma condição especifica, eliminando as possibilidades que não estão de acordo com essa condição. Esse método é muito utilizado principalmente no estudo da recursão, que é como ele enfrenta seus desafios e são inúmeros problema famosos que são resolvidos por esse algoritmo, por exemplo, “N queen Problem”, “The Knight´s tour problem” e até mesmo jogos famosos, como Sudoku e quebra cabeças, além de claro, problemas relacionados a labirintos. A forma que o algoritmo age é baseado em tentativa e erro, ou seja, com o algum objetivo/condição a ser respeitada, ele passa a testar, no caso desse desafio, as 4 direções possíveis de todo caminho novo que ele encontrar, sempre demarcando quais posições ele já percorreu. Dessa forma, mesmo que ele vá para um caminho sem saída, ele consegue retornar em pontos específicos que ainda não foram testados, seguindo assim, para uma nova direção, até efetivamente encontrar alguma saída. Por ser uma forma útil e até fácil de ser

implementado, diversos blogs e sites fazem uma definição sobre o que é e como funciona o backtracking, mas na Wikipédia, a definição e usos desse algoritmo ficaram muito bem exemplificados, como pode ver na citação abaixo:

> **“**O precedimento é usado em linguagens de programação como Prolog. Uma busca inicial em um programa nessa linguagem segue o padrão busca em profuncidade, ou seja, a àrvore é percorrida sistematicamente de cima para baixo e da esquerda para direita. Quando essa pesquisa falha, ou é encotrado um nodo terminal da árvore, entra em funcionamento o macanismo de backtracking. Esse procedimento faz com que o Sistema retorne pelo mesmo caminho percorrido com a finalidade de encontrar soluções válidas. É inutil, por exemplo, localizar um determinado valor em uma tabela não ordenada. Quando aplicável, no entanto, o retrocesso é muitas vezes muito mais rápido do que à enuneração por força bruta de todos os candidatos completos, pois pode eliminar muitos candidatos com um único teste.”

7. DESENVOLVIMENTO

8. Arquivo.h:

O arquivo.h foi por onde eu iniciei o meu projeto, pois sabia que a estrutura de dados pilha, não seria tão complexa de ser implementada. A forma que ela foi criada, não possui quase nenhuma mudança de uma pilha comum, principalmente em seus métodos, que são os padrões. Tirando o fato da struct receber 2 valores inteiros, que serão as coordenadas dos caminhos, o resto é uma pilha convencional.

```
class Pia_Di{
public:
Pia_Di();
~Pia_Di();
void push(int x, int y);
void pop(int &x, int &y);
bool vazio();
bool cheio();
int print();
private:
struct StackNode{
int x, y;
StackNode *prox_endereco;
};
typedef StackNode *prox_pont;
prox_pont top;

};
};
```

9. Metodos da Estrutura de Dados:

Assim como no arquivo.h, imaginava que os métodos não seriam tão diferentes dos que foram apresentados em aula. Portanto, a única grande diferença é receber 2 valores ao invés de 1, já que estamos trabalhando com coordenadas, ou seja, eixo x e y. Sendo assim, a pilha serve como um simples armazenamento de dados para o nosso código, sendo constantemente atualizada por conta dos sucessivos pop´s e push´s, contendo também, um método de print completo da pilha. É interessante ressaltar que esse método de print possui uma característica fundamental da pilha, que é printar as informações no sentido contrário, já que a pilha é uma LIFO (last in first out), algo que poderia ser modificado caso desejado, ocasionando algumas diferenças no método.

```
#include <iostream>
#include "EstagioSupervisionado.h"
using namespace std;
//tem que ter um delete no pop
Pia_Di::Pia_Di(){
    //pré-condição: nenhuma
    //pós-condição: Pilha é criada e iniciada como vazia
    top = NULL;
};
Pia_Di::~Pia_Di(){
    //pré-condição: Pilha já tenha sido criada
    //pós-condição: Pilha é destruída
    int entrada1, entrada2;
    while(vazio() != true){
        pop(entrada1, entrada2);
    }
}
bool Pia_Di::vazio(){
    //pré-condição: Pilha já tenha sido criada
    //pós-condição: função retorna true se a pilha está vazia, false
caso contrário
    if(top == NULL){
        return true;
    }
    else{
        return false;
    }


}
bool Pia_Di::cheio(){
    //pré-condição: Pilha já tenha sido criada
    //pós-condição: função retorna true se a pilha está cheia, false
caso contrário
    return false;

}

void Pia_Di::push(int x, int y){
```

```
    //pré-condição: Pilha já tenha sido criada e não está cheia
    //pós-condição: Os itens x e y são armazenados no topo da pilha
    prox_pont p;
    p = new StackNode;
    if(p == NULL){
        cout << "Pilha cheia." << endl;
        abort();

    }
    else{
        p->x = x;
        p->y = y;
        p->prox_endereco = top;
        top = p;
    }



};
void Pia_Di::pop(int &x, int &y){
    //pré-condição: Pilha já tenha sido criada e não está vazia
    //pós-condição: O item no topo da pilha é removido e seu valor é
retornado nas variáveis x e y.
    if(vazio() == true){
        cout << "A pilha esta vazia, nao existe valores para serem re-
tirados." << endl;

    }
    else{
        prox_pont p;
        x = top->x;
        y = top->y;
        p = top;
        top = p->prox_endereco;
        delete p;
    }


}


int Pia_Di::print(){
    //pré-condição: Nenhuma
    //pós-condição: Printa na tela todos os valores armazenados na pi-
lha, começando pelo top
    prox_pont p;
    p = top;
    int i;
    for(i = 0; p != NULL; i++){
        cout << "(" << p->x << " " << p->y << ")" << "  ";
        p = p->prox_endereco;

    }
    return i;
}
```

10. Arquivo Main:

Diferente do .h e dos métodos, o driver definitivamente foi o maior problema, visto que é nele que o algoritmo de backtracking seria implementado. Primeiramente, desenvolvi tudo que achava necessário para conseguir percorrer o labirinto, então, criei uma struct com valores inteiros que representariam as coordenadas, para facilitar o armazenamento e os testes futuros. Além disso, obviamente, era preciso criar o labirinto que seria percorrido pelo código, então, preferi cria-lo em um arquivo txt separado e depois trazê-lo para a ".main" em forma de matriz. Foi criado também uma função que recebe como parâmetro, uma coordenada xy e retorna todas as direções possíveis para quem à chamou, ponto extremamente importante para o funcionamento do código, já que é função responsável por devolver todos os testes necessários para o algoritmo de backtracking. E por último, uma simples função que recebe uma matriz, no caso, o labirinto e printa ele na tela, para a visualização ficar completa sobre aquela solução encontrada.

```
#include <iostream>
#include <vector>
#include <fstream>
#include "EstagioSupervisionado.h"
using namespace std;

//Struct para armazenar as cordenadas
struct Cordenada{
    int x, y;
};
typedef struct Cordenada cord;

//Pré-Condição: Recebe a cordenada da posição atual.
//Pós-Condição: Retorna todos os caminhos ao redor da posição atual.
vector<Cordenada> Direcoes(cord posi){
    //Sequência original usada: direita, baixo, esquerda e cima.
    //Caso desejar, é possível modificar essa sequência alterando os
valores de incremento e decremento.
    //Vale lembrar, que por mais que as direções sejam modificadas,
apenas a sequência de soluções encontradas para o labirinto seram di-
ferentes, ou seja, ele continua com as mesmas soluções sempre.
    return {{posi.x , posi.y + 1}, {posi.x + 1, posi.y}, {posi.x,
posi.y - 1}, {posi.x - 1, posi.y}};
}

//Pré-Condição: Receber um vetor/matriz de strings que será o labi-
rinto.
//Pós-Condição: Carregar a matriz do arquivo .txt para o programa.
void Carregando_Matriz(vector <string> &labirinto){
    //Caso crie um novo arquivo, não esqueça de mudar a Constante No-
meDoArquivo nos defines iniciais.
    ifstream lab(NomeDoArquivo, ios::in);
    if(lab.good()){
        for(int i = 0; i < LINHA; i++){
```

```
            getline(lab, labirinto[i]);
        }
    }
    else{
        cout << "Erro ao abrir o arquivo." << endl;
        abort();
    }
}

void Printando_Matriz(vector <string> labirinto){
    //Pré-Condição: Receber um vetor/matriz de strings que será o la-
birinto.
    //Pós-Condição: Printa esse labirinto na tela.
    for(int i = 0; i < LINHA; i++){
        cout << "    " << labirinto[i] << endl;
    }
}
```

Com todas essas funções feitas, chegou a hora de desenvolver o algoritmo de backtracking. Criei uma função chamada “Jogo” que recebe como parâmetros: Uma matriz (labirinto), a coordenada do “player”, a coordenada final que representa a saída, uma estrutura de dados Pilha e 2 contadores, que servem apenas para armazenar a quantidade de passos do caminho encontrado e a quantidade total de caminhos encontrados. Por ser um algoritmo de recursão, criei primeiro qual seria a condição específica que a função deveria procurar atender, que seria, se posição do player é igual a posição da saída e caso for atendida, é porque uma saída foi encontrada. O ponto chave desse código é não deixar que o programa acabe no momento que ele encontrar uma saída, justamente porque estamos procurando todas as saídas possíveis. A partir disso, o programa começa a procurar os caminhos corretos que levem até a saída, sempre verificando 2 condições, além da de saída encontrada, que são: Se a posição atual é uma parede ou se ela já foi visitada anteriormente. Se nenhuma dessas condições for atendida, é porque esse é um novo caminho plausível, portanto o programa deve armazená-lo na pilha, marcá-lo como visitado e percorrer todas as direções dele. Porém, caso todo esse caminho seja sem saída, ele não entrara em nenhuma das condições anteriores, então é necessário que ele retorne para algum ponto ainda não testado, com uma sequência de pop´s na nossa pilha e com o algoritmo recursivo, ele consegue voltar em um ponto ainda não testado e continuar sua verificação, até que ele consiga encontrar o final do labirinto, e ao final, o programa acaba quando não existir mais caminhos para serem testados.

```
bool Jogo(vector<string> &matriz, cord posi_player, cord final, Pia_Di
&possibilidades, int cont, int &qtd_cmh){
```

```
    //Pré-Condição: Recebe uma matriz de strings, que é o nosso labi-
rinto, a posição inicial/atual, a posição que deseja encontrar, uma
ADT Pilha para armazenar as posições percorridas, e um simples conta-
dor de passos.
    //Ps: é preciso passar esse contador pela própria função por quê
caso você crie uma variável dentro dela, sempre que uma nova recursão
for chamada, essa variável será resetada para o valor atribuído inici-
almente, o mesmo vale para o qtd_cmh.
    //Pós-Condição: Retorna todas as possibilidades de caminhos possí-
veis através da matriz e o caminho percorrido pelo programa, e o nú-
mero de "passos" que foi dado até chegar a saída.
    int tempX, tempY;//variável apenas para armazenar os valores reti-
rados pela função Pop da pilha.
    //Condição principal do programa, que verifica se a posição atual
é igual a posição desejada.
    if(posi_player.x == final.x && posi_player.y == final.y){
        //caso for, é printado a matriz, o caminho percorrido e o nú-
mero de passos.
        qtd_cmh++;
        cout << endl << "-Solucao Encontrada-" << endl;
        Printando_Matriz(matriz);
        cout << "Caminho percorrido (Final -> Inicial): ";
        possibilidades.print();
        cout << "-Numero de passos: " << cont << endl;

    }
    //Condição para verificar se a nova posição é um caminho possível
ou se já foi visitado anteriormente.
    if(matriz[posi_player.x][posi_player.y] != Caminho || ma-
triz[posi_player.x][posi_player.y] == Marca){
        return false;
    }
    //Caso a condição anterior esteja errada, é porque essa nova posi-
ção, é um caminho possível.
    else{
        //A posição atual é marcada como visitada, a cordenada é arma-
zenada na pilha e o contador de passos é incrementado.
        matriz[posi_player.x][posi_player.y] = Marca;
        possibilidades.push(posi_player.x, posi_player.y);
        cont++;
        //Após isso, o programa faz um novo teste para cada 'caminho'
retornado pela funcao Direcoes, da posição atual.
        for(auto caminhos : Direcoes(posi_player)){
            if(Jogo(matriz, caminhos, final, possibilidades, cont,
qtd_cmh) != false){
                return true;
            }
        }

    }
    //Se o caminho atual não possuir mais possibilidades de direção
aceitáveis, ele é retirado da pilha, o contador é decrementado e a po-
sição volta a não ter nada.
    possibilidades.pop(tempX, tempY);
    cont--;
    matriz[posi_player.x][posi_player.y] = ' ';

    return false; //condicao de parada, caso nao exista mais nenhuma
saida possível. Essa devera ser a ULTIMA linha dessa funcao.

}
```

11. CONCLUSÃO

Na universidade, é muito bem dividido as aulas teóricas, com seus inúmeros materiais, livros, teorias e as atividades práticas, que para a grande maioria, são a parte mais interessante, já que é nela que conseguimos ver na prática todos aqueles argumentos e testes que são demonstrados anteriormente. Porém, por mais que essas aulas sejam interessantes e necessárias para entender bem o assunto, todo aluno busca praticar e tentar aquilo que ele tem mais interesse dentro daquele meio e muitas vezes, essas coisas mais interessantes, tendem a ser mais complexas, em níveis acima das aulas práticas.

Por isso, sair da zona de conforto e encarar o peso de pensar e criar um projeto do zero é extremamente importante para o desenvolvimento de um aluno. Aprender a pesquisar em fóruns, sites, livros, referências bibliográficas, perder horas lendo sobre um assunto que jamais imaginou que seria necessário aprender e inúmeros outros problemas que são necessários enfrentar para concluir o projeto, fazem parte do amadurecimento do aluno e o ensina a ter paciência, humildade de reconhecer que precisa de ajuda, interesse em buscar novos conhecimentos e acima de tudo, nunca desistir.

Pode acabar soando complexo demais, exaustivo e complicado no primeiro momento e por um lado, realmente é, porém, foi por isso que escolhi enfrentar esse desafio. Foi preciso aprender a lidar com erros, frustrações, prazos de entrega e tudo isso me fortaleceu como profissional, pude sentir na pelo como é ser talvez, um estagiário de uma empresa buscando aprovação através de um projeto.

Por fim, digo que esse estágio supervisionado me ensinou muito mais do que eu poderia imaginar, contribuindo, não só para o meu crescimento como profissional, mas como pessoa também, me mostrando como é necessário ter força e determinação para concluir meus objetivos. Apesar de todas as frustrações, tudo foi recompensado quando finalmente conclui o projeto e pude perceber que sou capaz de enfrentar qualquer desafio, bastando ter paciência, confiança nos estudos e determinação para vencer.

12. REFERÊNCIAS

JOGO & PROGRAMAÇÃO. Linguagem C labirinto 1, Youtube, 19 de junho de 2020. Disponível em: <https://www.youtube.com/watch?v=p13WFCzpnwU&ab_channel=Jogos%26Programa%C3%A7%C3%A3o>. Acesso em 12 de outubro de 2022.

MATURANO, Lorenzo. Pilha. Lorenzomaturano, 2020. Disponível em: < http://www.lorenzomaturano.com/blog/estruturas-de-dados/2020/09/pilha >. Acesso em 22 de outubro de 2022.

BACKTRACKIN ALGORITHMS. Geeksforgeeks, 2020. Disponível em: < https://www.geeksforgeeks.org/backtracking-algorithms/>. Acesso em 03 de novembro de 2022.

RAT IN A MAZE. Geeksforgeeks, 2020. Disponível em: < https://www.geeksforgeeks.org/rat-in-a-maze-backtracking-2/>. Acesso em 03 de novembro de 2022.

Assinatura do Aluno:

Assinatura do Docente Orientador do Estágio: