

UNIVERSIDADE FEDERAL DO PIAUÍ

CAMPUS SENADOR HELVÍDIO NUNES DE BARROS - CSHNB

Curso: Sistemas de Informação

Disciplina: Estruturas de dados 2

Docente: Juliana Oliveira de Carvalho

Discente: Marcos André Leal Silva

**UNIVERSIDADE FEDERAL DO PIAUÍ - UFPI**

Relatório técnico

**Picos-PI**

**04 de fevereiro de 2024**

**Resumo:** Este projeto, tem como objetivo demonstrar a prática e implementação de duas Estruturas de Dados: Grafos e hash. Essas estruturas foram utilizadas para a resolução de alguns problemas práticos proposto no projeto através da Linguagem de programação C. O mesmo abordará as etapas de cada questão e apresentará suas respectivas implementações, oferecendo uma visão abrangente das aplicações dessas estruturas de dados. Além do mais, serão feitos experimentos para verificar os tempos de desempenho tanto das questões que envolvem grafos quanto das questões que envolvem funções hash.

**Introdução:** As estruturas de dados são um conceito muito importante para a programação e ciência de dados. Tratam-se de maneiras de agregar e organizar dados na memória de um computador ou dispositivo, de forma que façam sentido e proporcionem um bom desempenho ao serem processados. Nesse sentido, estamos considerando dados como sendo blocos de programação que representam algo e têm a função de resolver problemas computacionais. Para isso, eles devem ter a possibilidade de ser simbolizados, armazenados e manipulados.

A teoria dos grafos ou de grafos é um ramo da matemática que estuda as relações entre os objetos de um determinado conjunto. Para tal são utilizadas estruturas chamadas de grafos, G(V,E) onde V é um conjunto não vazio de objetos denominados vértices (ou nós) e E (do inglês edges - arestas) é um subconjunto de pares não ordenados de V.

Uma função hash é um algoritmo que mapeia dados de comprimento variável para dados de comprimento fixo. Os valores retornados por uma função hash são chamados valores hash, códigos hash, somas hash (hash sums), checksums ou simplesmente hashes. Um uso é uma estrutura de dados chamada de tabela hash, amplamente usada em software de computador para consulta de dados rápida.

O objetivo desse projeto é através das estruturas de dados “Grafos” e “Tabela Hash”, criar três aplicações capazes de:

1. Aplicar um algoritmo de caminho mínimo para resolver o problema da Torre de Hanói com 4 discos a partir da combinação de modelagem de problemas com grafos, implementação de uma matriz de adjacência, aplicação de um algoritmo clássico de caminho mínimo (Ford-Moore-Bellman) e a análise do tempo de execução.
2. Encontrar o caminho mais confiável entre dois vértices em um grafo orientado. Neste contexto, a confiabilidade de uma aresta é representada por um valor no intervalo de 0 a 1, indicando a probabilidade de que o canal de comunicação associado não falhe. A busca pelo caminho mais confiável envolve encontrar a rota que maximize a probabilidade total ao longo das arestas entre dois vértices específicos.
3. Desenvolver dois programas que implementem diferentes funções de hashing e tratamento de colisões para organizar uma base de dados de 1000 funcionários. A organização é feita utilizando uma tabela de hash para localizar os dados dos funcionários com base em seus números de matrícula. A questão propõe duas funções de hashing distintas e pede para comparar o desempenho de ambas, tanto em termos de eficiência quanto no número de colisões geradas.

Para a implementação dos projetos, foi utilizado a linguagem de programação C, com seu paradigma de programação imperativo para a criação do código fonte. Através da ferramenta “Visual Studio Code (VS Code)”, esses programas foram criados, testados e executados.

O primeiro programa apresenta duas estruturas (Vertice e Grafo) para representar a implementação do grafo. “Vertice” representa um vértice no grafo. Cada vértice corresponde a uma configuração do jogo da Torre de Hanói (discos[4]: Array que armazena a configuração dos discos nos pinos), armazenando a disposição dos discos nos pinos. “Grafo” contém informações sobre o grafo que representa as configurações do jogo da Torre de Hanói (vertices: Ponteiro para um array de structs Vertice, representando os vértices do grafo; arestas: Matriz de adjacência representando as arestas do grafo. Cada entrada indica se há uma ligação entre dois vértices).

O segundo programa apresenta três estruturas (Aresta, Graco e FilaPrioridade) para representar a implementação do grafo. “Aresta” representa uma aresta no grafo, contendo informações sobre o vértice destino e sua confiabilidade (vertice: Número do vértice de destino; confiabilidade: Valor representando a confiabilidade da aresta). “Grafo” contém informações sobre o grafo, como as arestas que ligam os vértices (arestas: Ponteiro para um array de structs Aresta, representando as arestas conectadas a um vértice; num\_arestas: Número total de arestas conectadas ao vértice). “FilaPrioridade” utilizada para implementar uma fila de prioridade durante o algoritmo de Dijkstra, mantendo informações sobre a confiabilidade e o vértice (confiabilidade: Valor representando a confiabilidade do vértice; vertice: Número do vértice).

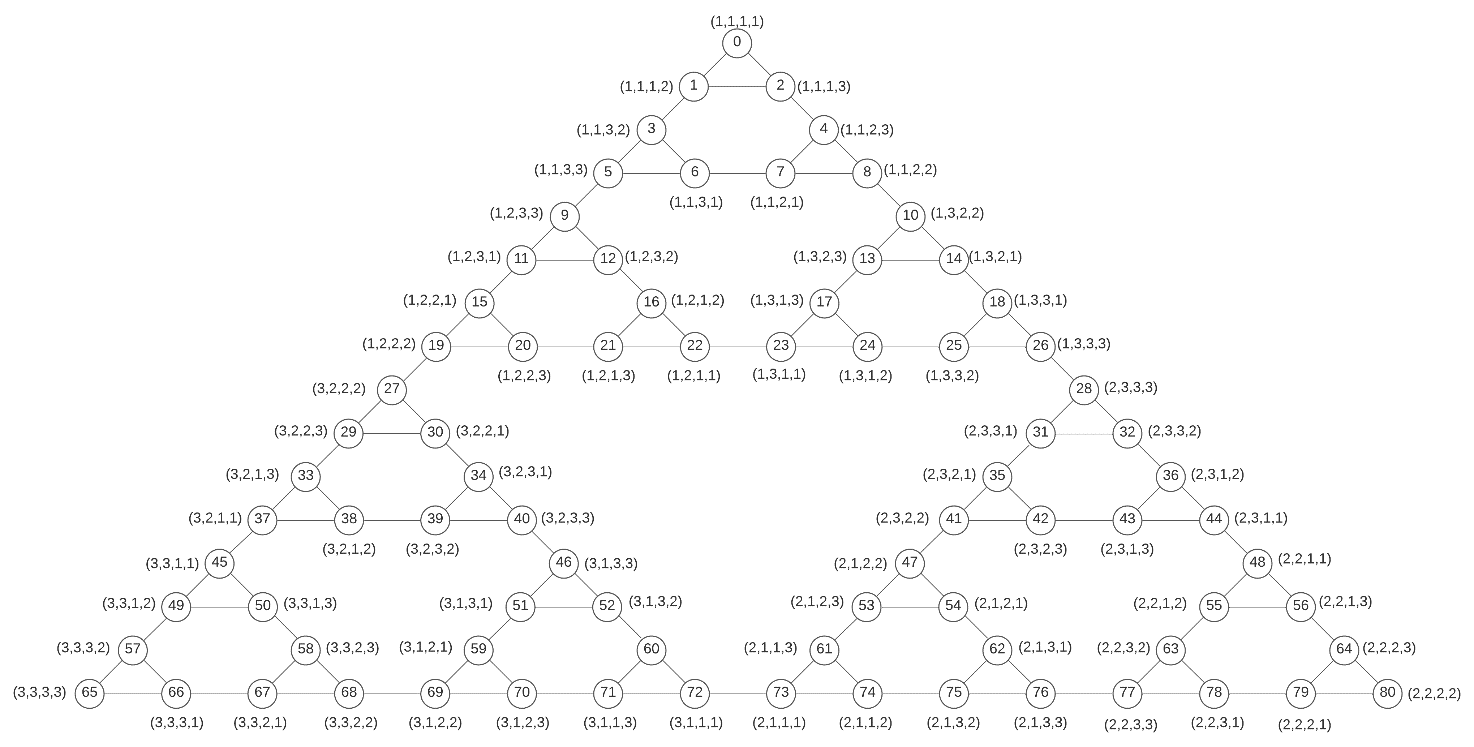
O terceiro programa apresenta uma estrutura (Func) para representar as informações sobre os funcionários (matricula: Uma string de até 6 caracteres que representa a matrícula do funcionário; nome: Uma string de até 49 caracteres que armazena o nome do funcionário; funcao: Uma string de até 19 caracteres que contém a função do funcionário; salario: Um valor em ponto flutuante que armazena o salário do funcionário). funcaoHashStringRotacao (char str): Realiza uma transformação na string de matrícula, efetua cálculos com alguns dígitos e retorna um índice para a tabela de tamanho 101 (MAX101).funcaoHashStringFoldShift (char str): Converte parte da string de matrícula em dois inteiros, soma-os e retorna um índice para a tabela de tamanho 150 (MAX150).

**Seções específicas:**

**Informações técnicas:** Para o desenvolvimento e testes deste projeto foi utilizado um notebook com um processador 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz, doze gigas de memória RAM ddr4, sistema operacional com a arquitetura de 64 bits, Windows 11 Home.

Os códigos foram feitos na IDE e editor de texto Visual Studio Code e compilados pelo compilador gcc (MinGW.org GCC Build-2) 9.2.0.

**Questão 1:**

****

O programa em questão é um jogo da Torre de Hanói implementado em C, que utiliza um grafo representado por uma matriz de adjacência para armazenar as configurações possíveis do jogo e as transições entre elas.

1. Estrutura do Grafo:

* O grafo é representado pela estrutura Grafo, que contém um vetor de vértices (vertices) e uma matriz de adjacência (arestas).
* Cada vértice é representado pela estrutura Vertice, que armazena as configurações dos discos nos pinos.

1. Criação do Grafo:

* A função Criar\_Grafo aloca dinamicamente a memória necessária para o grafo, seus vértices e a matriz de adjacência.

1. Leitura do Grafo a partir de um Arquivo:

* A função Inserir\_vertices\_e\_arestas lê as informações do arquivo torre\_hanoi.txt para preencher os valores dos vértices e a matriz de adjacência.

1. Impressões:

* A função Imprimir\_Matriz\_adjacente imprime a matriz de adjacência.
* A função Imprimir\_conteudo\_do\_vertice imprime as configurações de um vértice específico.

1. Algoritmo de Bellman-Ford:

* A função bellmanFord implementa o algoritmo de Bellman-Ford para encontrar o menor caminho em um grafo direcionado com arestas ponderadas.
* Verifica a existência de ciclos negativos no grafo.

1. Jogo da Torre de Hanói:

* As funções Vertices\_adjacentes, Verificar\_Vitoria, Verificar\_Movimento\_Valido e jogar implementam a lógica do jogo.
* O jogador pode escolher iniciar o jogo a partir de um vértice específico ou do vértice inicial.

1. Menu Interativo:

* O programa apresenta um menu interativo que permite ao usuário escolher diferentes funcionalidades, como imprimir valores dos vértices, matriz adjacente, jogar, etc.

1. Liberação de Memória:

* A função Liberar\_Grafo é responsável por liberar a memória alocada dinamicamente para o grafo.

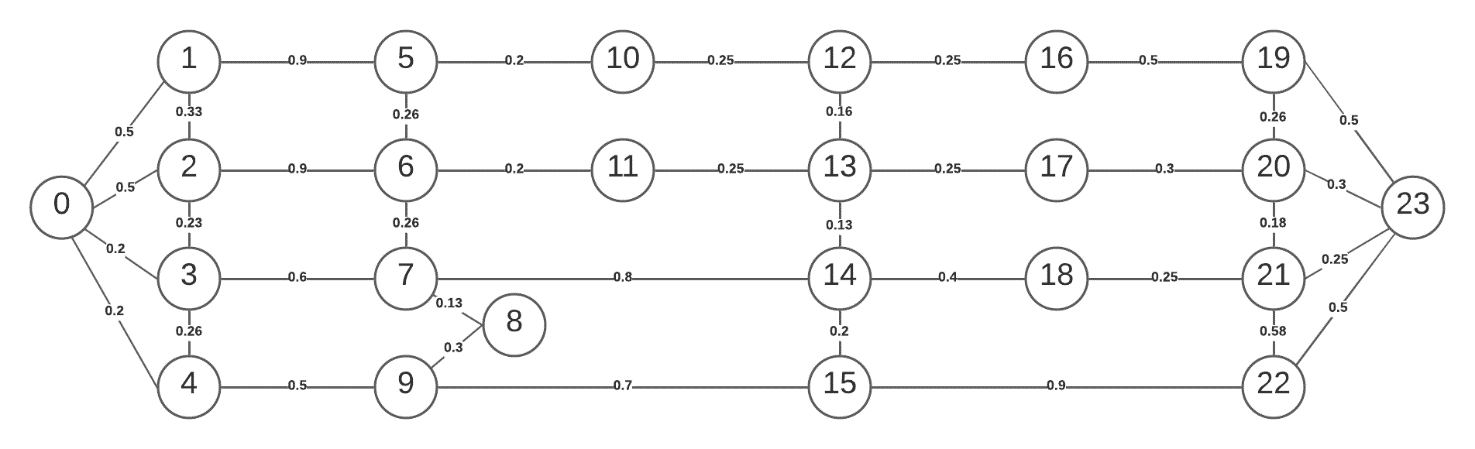
1. Tempo de Execução:

* O programa utiliza a função QueryPerformanceCounter para medir o tempo de execução do algoritmo de Bellman-Ford.

1. Encerramento do Programa:

* O programa continua em execução até que o usuário escolha a opção "0" no menu, indicando a saída.

**Questão 2:**

****

O programa em questão implementa o algoritmo de Dijkstra para encontrar o caminho mais confiável entre dois vértices em um grafo ponderado.

1. Estruturas de Dados:

* Utiliza as estruturas Aresta, Grafo, e FilaPrioridade para representar arestas, o grafo e a fila de prioridade, respectivamente.

1. Alocação Dinâmica:

* Usa alocação dinâmica de memória para criar o grafo e suas arestas, garantindo flexibilidade em relação ao número de vértices e arestas.

1. Leitura de Arquivo:

* Lê as informações do grafo a partir de um arquivo (grafo.txt) que contém o número de vértices, o número de arestas e as informações de cada aresta.

1. Preenchimento do Grafo:

* A função Ler\_Arestas\_Do\_Arquivo lê as arestas do arquivo e as insere no grafo utilizando a função Inserir\_Arestas.

1. Algoritmo de Dijkstra:

* A função dijkstra implementa o algoritmo de Dijkstra para encontrar o caminho mais confiável entre dois vértices em um grafo ponderado.
* Usa uma fila de prioridade para otimizar a seleção do próximo vértice a ser explorado.

1. Impressão do Resultado:

* Imprime o caminho mais confiável entre os vértices de origem e destino, bem como a confiabilidade desse caminho.

1. Liberação de Memória:

* A função Liberar\_Dijkstra\_Memoria é responsável por liberar a memória alocada durante o algoritmo de Dijkstra.

1. Verificação de Vértices Válidos:

* Verifica se os vértices de origem e destino fornecidos pelo usuário são válidos, ou seja, estão dentro dos limites do grafo.

1. Entrada do Usuário:

* Solicita que o usuário insira os vértices de origem e destino para calcular o caminho mais confiável entre eles.

1. Liberação de Memória ao Final:

* A função Liberar\_Grafo é utilizada para liberar a memória alocada para o grafo ao final do programa.

**Questão 3:**

O programa apresenta um sistema de gerenciamento de funcionários com duas tabelas hash diferentes: uma com 101 posições e outra com 150 posições.

1. Struct Func:

* Representa a estrutura de um funcionário com os campos matrícula, nome, função e salário.

1. Função lerFunc():

* Solicita ao usuário a entrada dos dados de um funcionário e retorna uma estrutura Func com essas informações.

1. Funções de Hash:

* colisaoRotacao: Lida com colisões usando o método de rotação.
* funcaoHashStringRotacao: Gera o valor hash utilizando o método de rotação.
* colisaoFoldShift: Lida com colisões usando o método de fold-shift.
* funcaoHashStringFoldShift: Gera o valor hash utilizando o método de fold-shift.

1. Funções inserirTabelaRotacao e inserirTabelaFoldShift:

* Inserem um funcionário na tabela de acordo com o método de hash escolhido. Contabilizam o número de colisões ocorridas.

1. Função imprimirFunc:

* Imprime as informações dos funcionários armazenados em uma tabela.

1. Menu de Opções:

* Permite escolher o tamanho da tabela (101 ou 150).
* Permite escolher o método de hash (rotação ou fold-shift).
* Oferece opções para inserir funcionários, imprimir a tabela e visualizar o número total de colisões.

1. Laços de Controle de Fluxo:

* Utiliza loops while para manter o programa em execução até a escolha da opção "0" para sair.

1. Alocação Dinâmica de Memória:

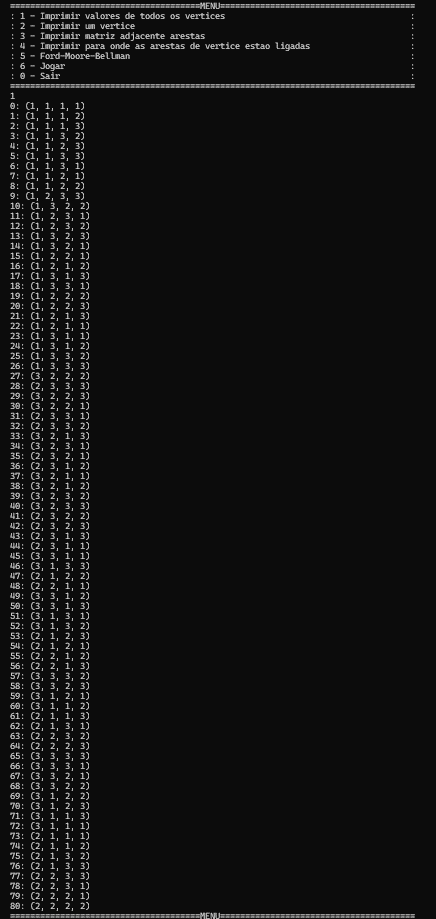
* Utiliza malloc para alocar memória para as tabelas de funcionários.

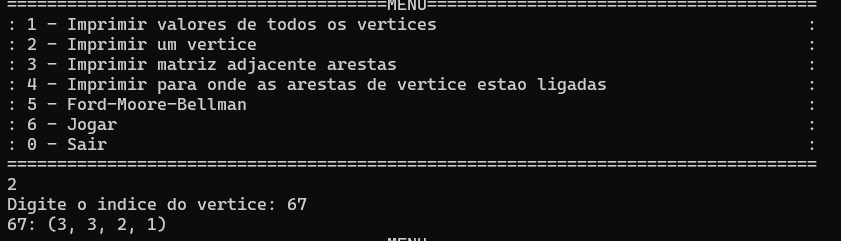
1. Liberação de Memória:

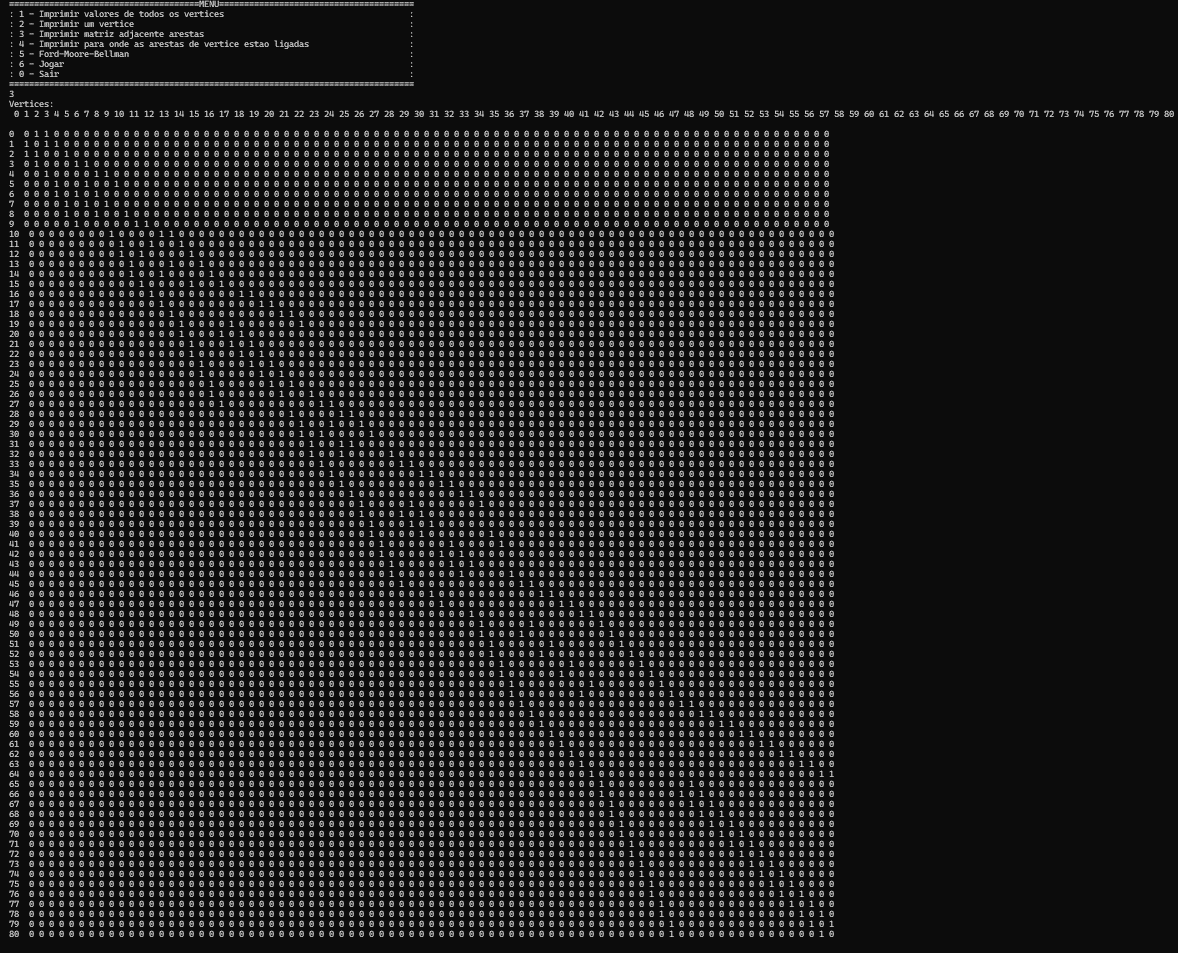
* Utiliza free para liberar a memória alocada após a conclusão do programa.

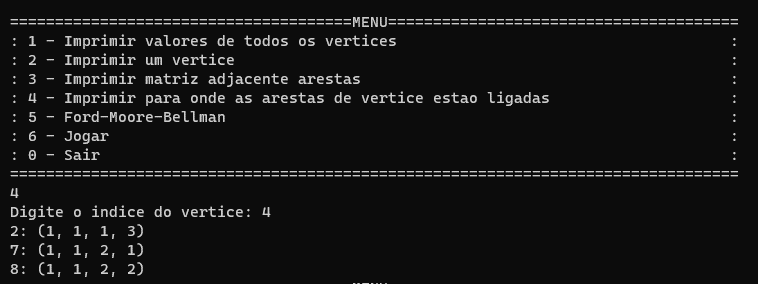
**Resultados da execução do programa:**

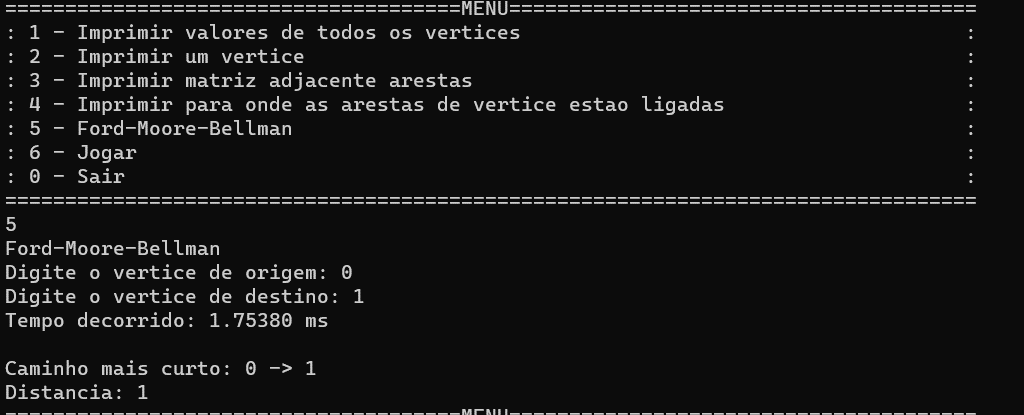
**Questão 1:**

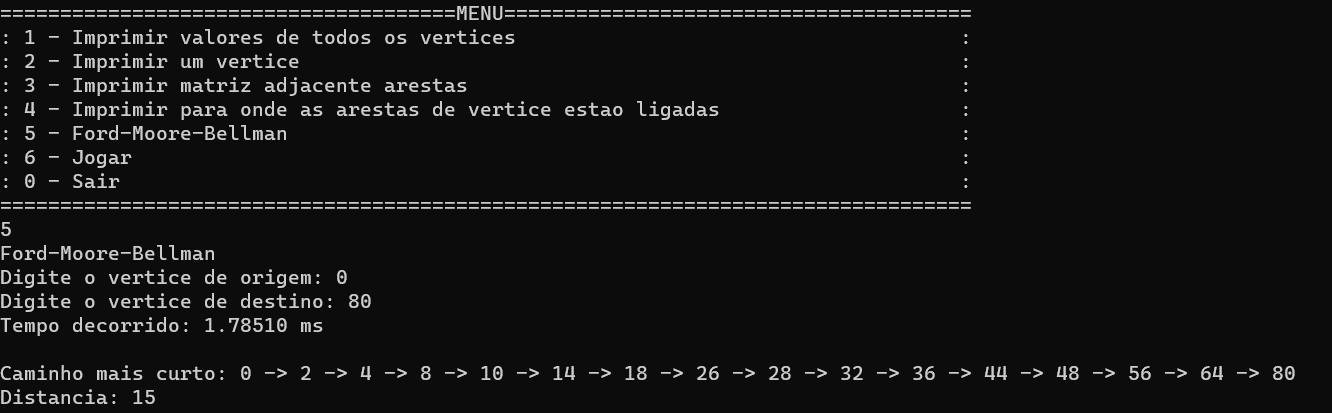


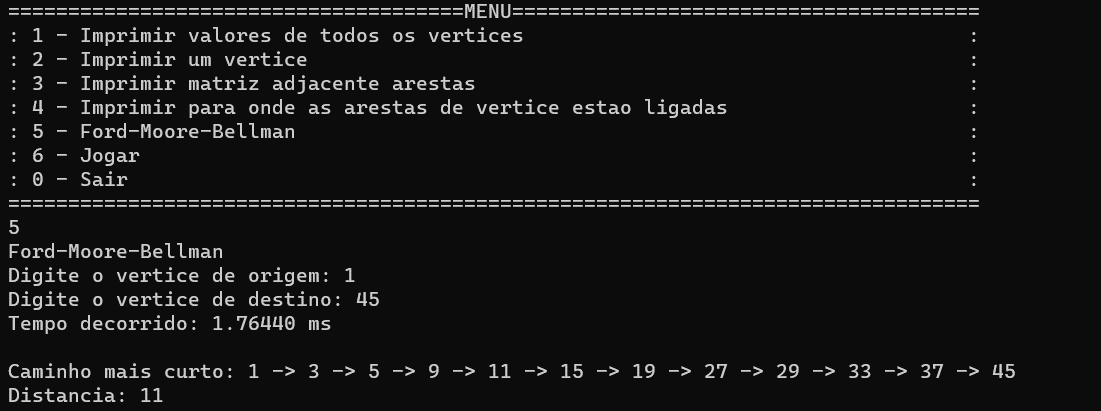


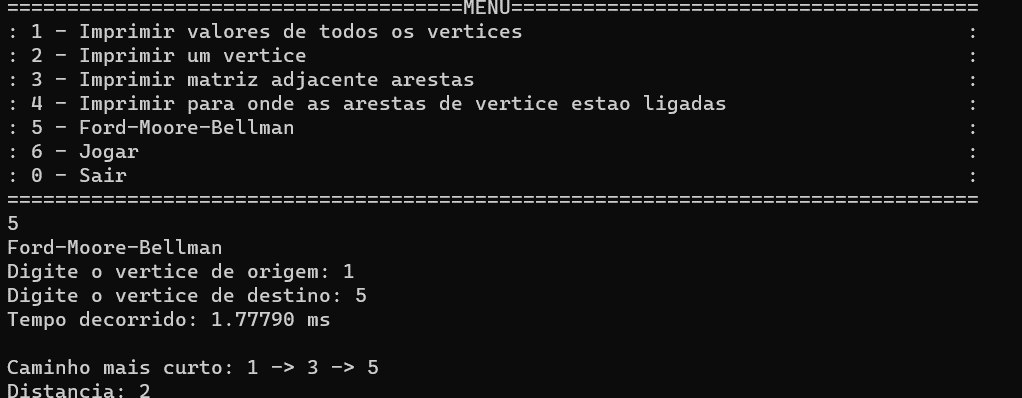


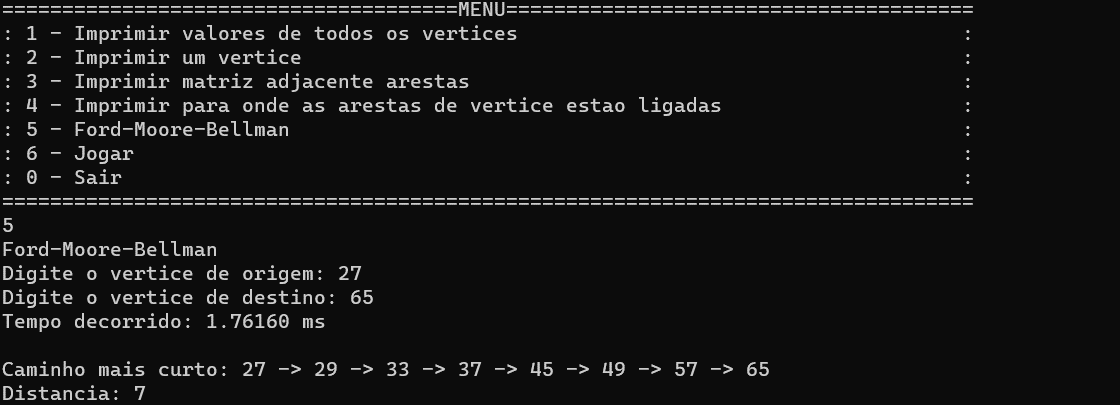


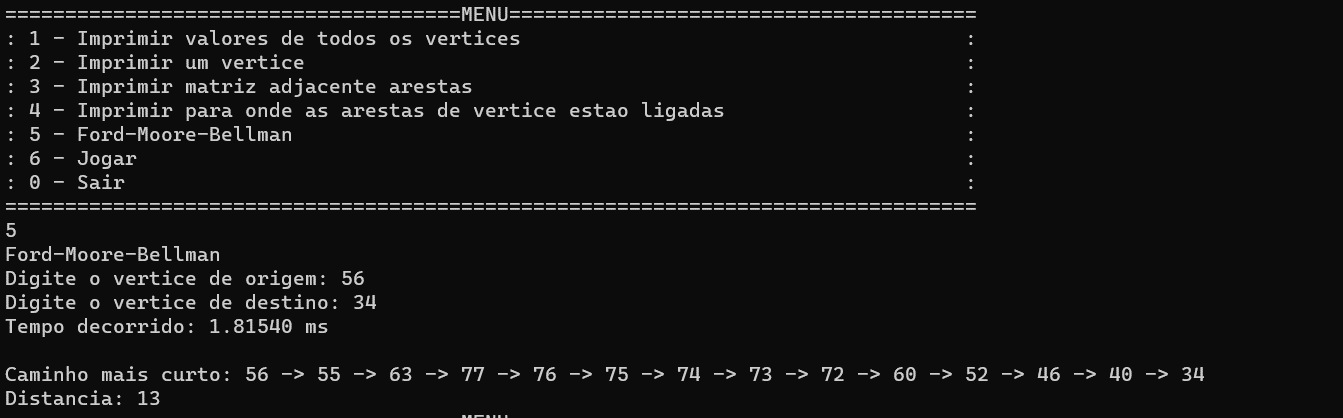


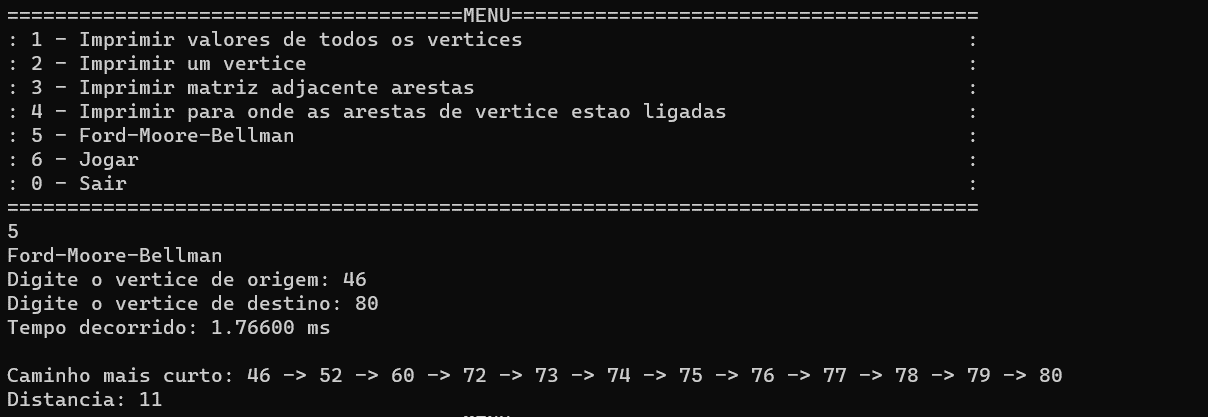


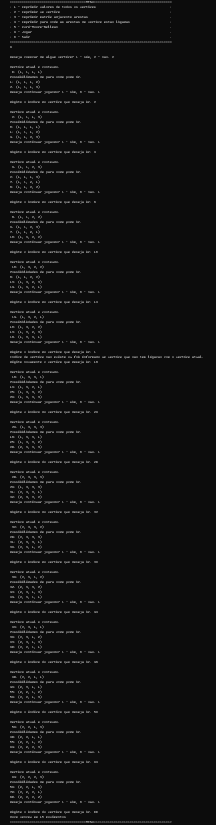




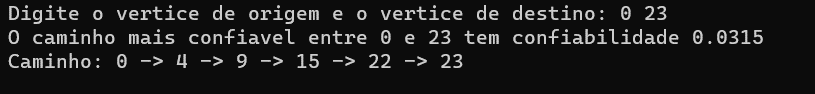


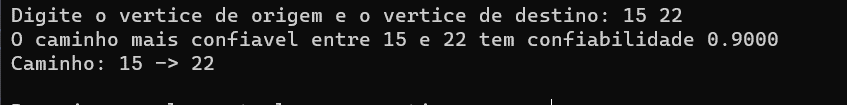


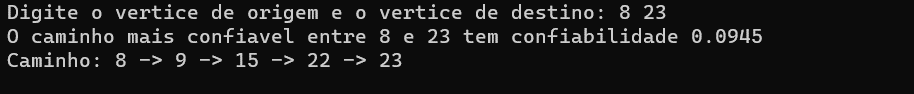


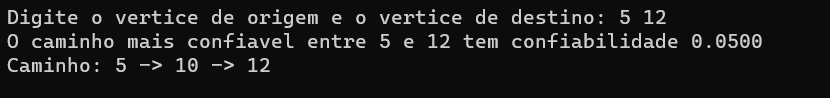


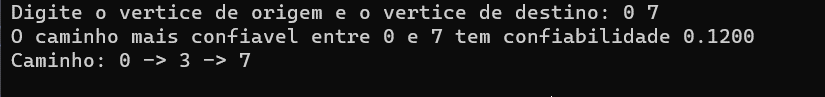
**Questão 2:**

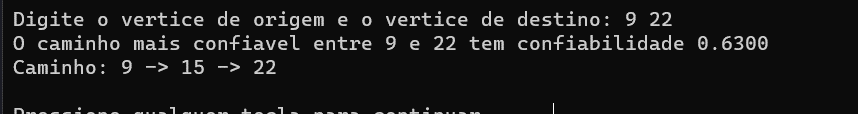


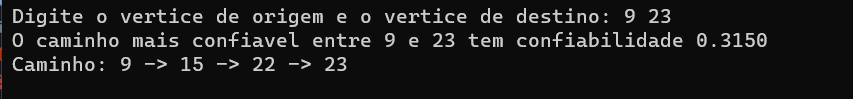




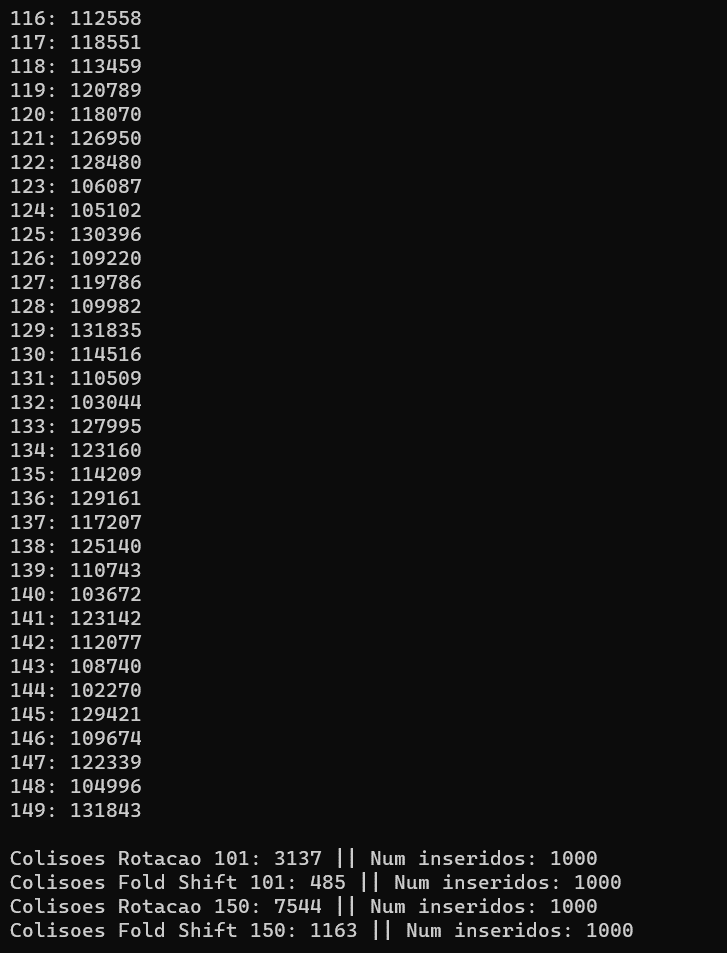


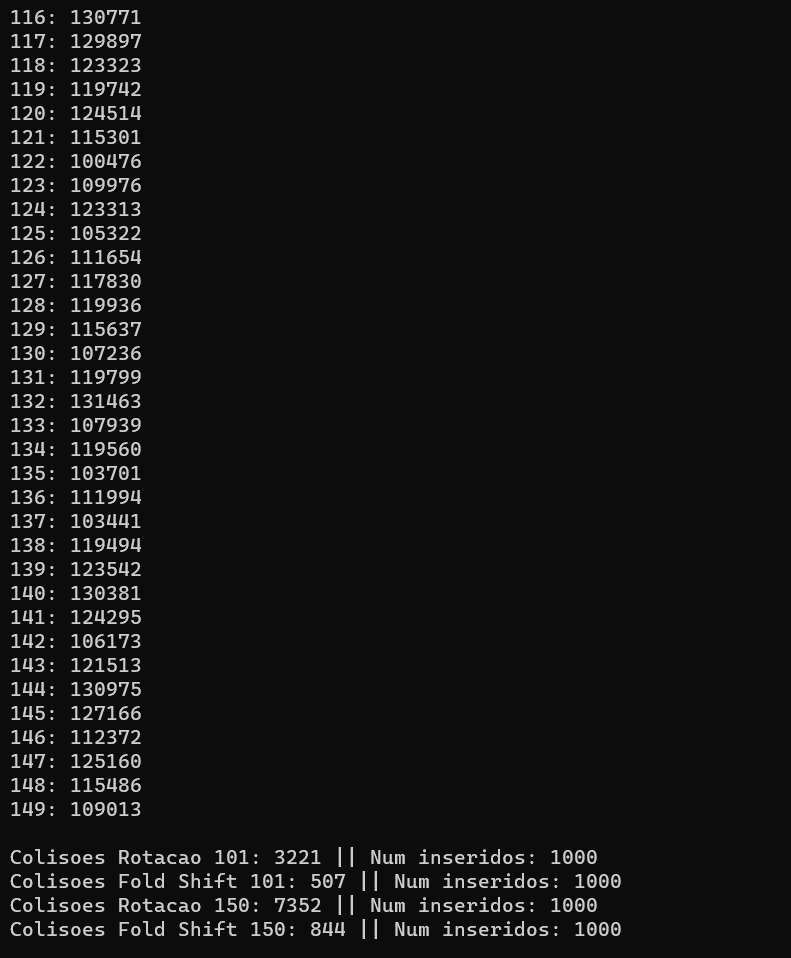


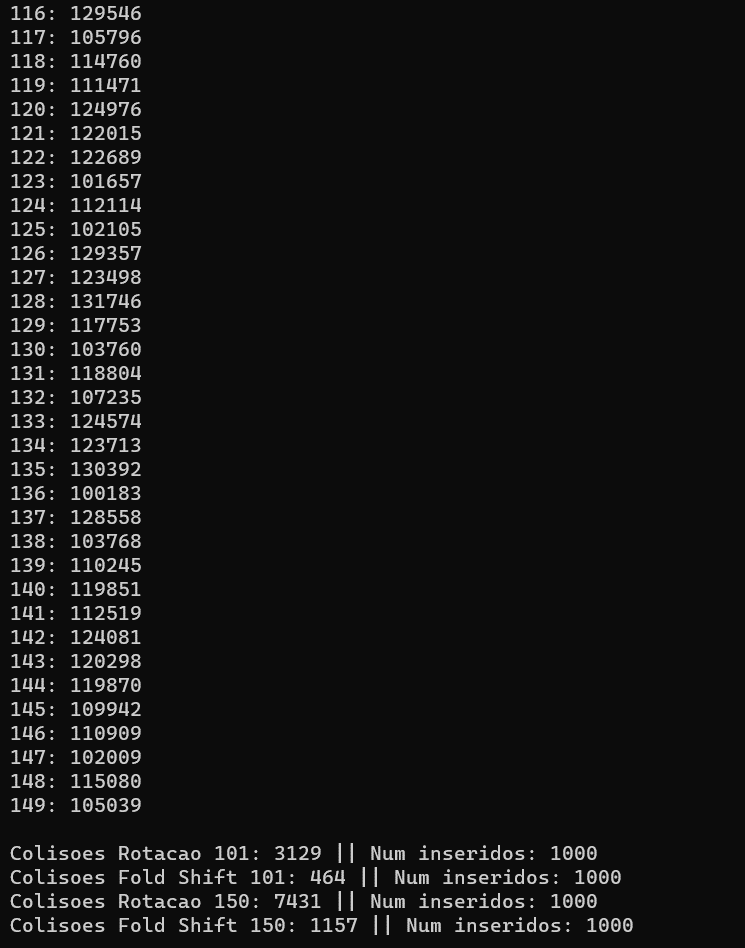


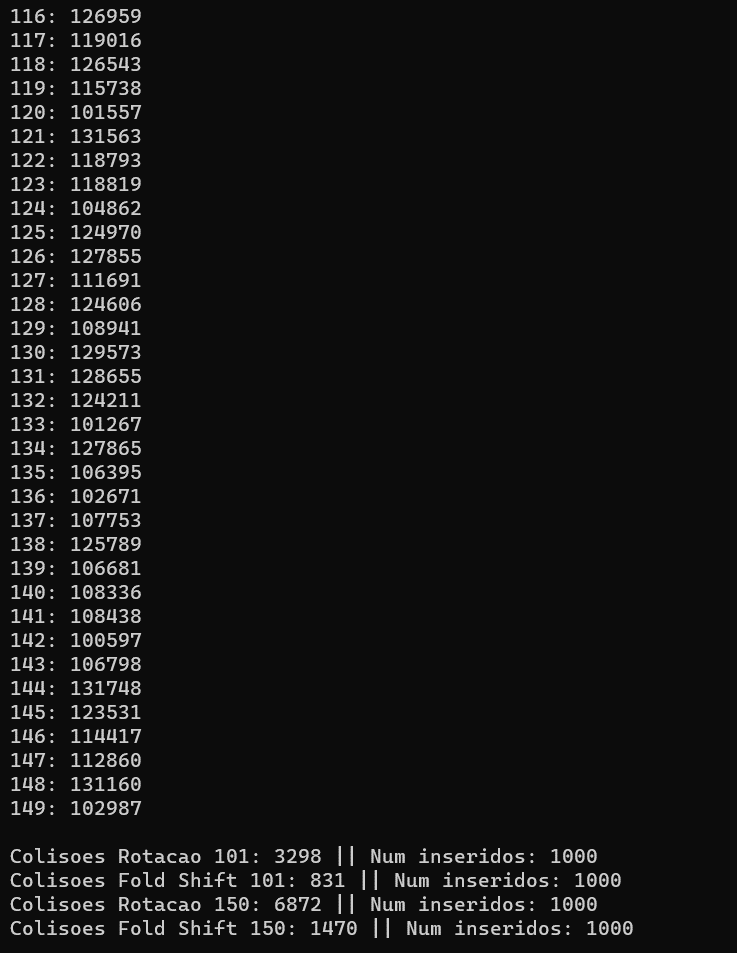


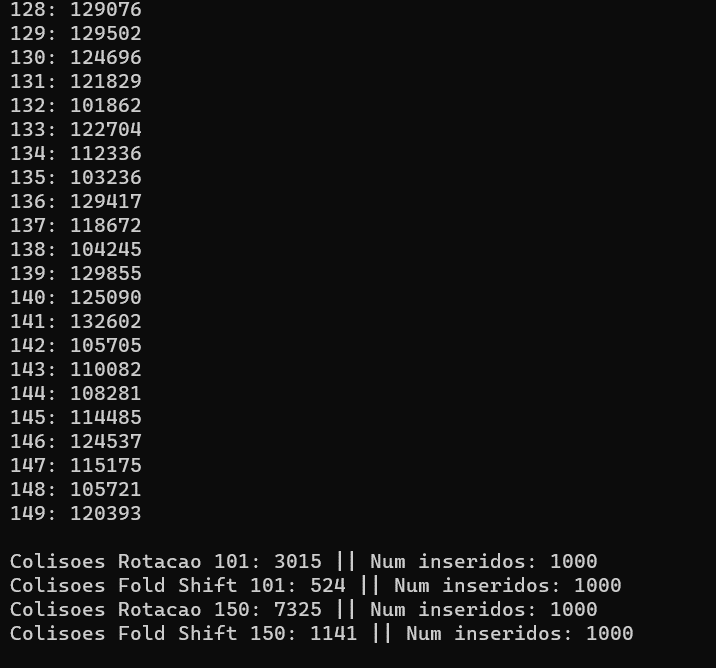
**Questão 3.1:**

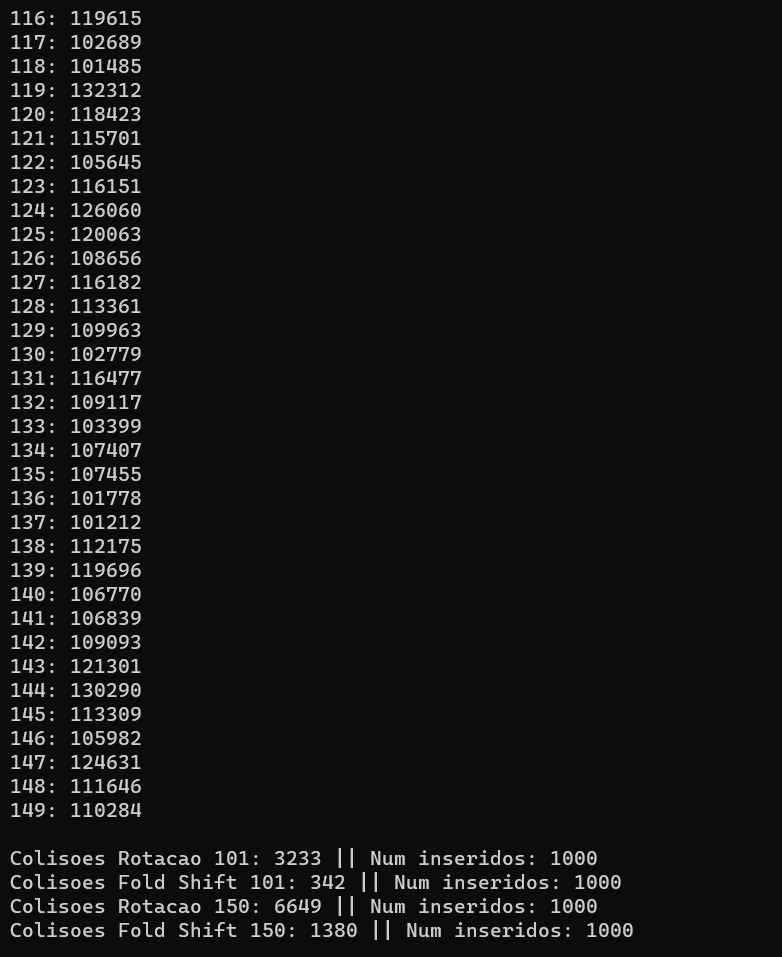












A partir da questão 1, foi observado que o número de movimentos mínimos para vencer o jogo da torre de hanoi com 4 discos (começando no vértice 0 até o vértice 65 ou 80) são de 15 movimentos. A questão 2 foi observado que o nível de confiabilidade pode variar bastante de um vértice até outro, e que o nível de confiabilidade de uma extremidade a outra do grafo (vértice 0 até vértice 23) foi de 0.0315 de acordo com o caminho percorrido, que pode ser analisada como de baixa confiabilidade, valores mais próximos de 1 são mais confiáveis. Por fim, a questão 3.1 (variação da questão 3 só que inserindo somente números de matriculas aleatórias) foi perceptível analisar que o método fold shift foi mais eficiente para tratar as colisões, pois a partir dele, outras colisões foram evitadas, aumentando seu desempenho em relação ao de rotação (Tanto no vetor de 100 posições quanto no de 150 posições).

**Conclusão:** Os programas forneceram uma estrutura básica para manipulação de grafos e de tabela hash. Além do mais, os experimentos de verificação de tempo, se mostraram ser eficientes perante a máquina utilizada para testar os mesmos. Com tudo que já foi dito, podemos concluir que os programas se mostram eficiente nos resultados de saída exibidos. Todas demais informações já foram citadas, explanadas e detalhadas nos tópicos acima. Sendo assim, os programas conseguiram responder as problemáticas dos projetos e se mostram sem erros ou insuficiências. Com estes experimentos, foi possível desenvolver e manipular grafos e tabela hash através da Linguagem C, apesar de ser um projeto simples, a boa interpretação do problema é crucial para uma ótima aplicação.

**Apêndice:**

**Questão 1:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <windows.h>

#define TAM 81

#define INF 999999

/\*Prototipos\*/

typedef struct *vertice* *Vertice*;

typedef struct *grafo* *Grafo*;

/\*Structs\*/

typedef struct *vertice*{

    int discos[4];

}*Vertice*;

typedef struct *grafo* {

*Vertice* \*vertices;

    int \*\*arestas;

}*Grafo*;

/\*Declaração das funções\*/

*Grafo* \*Criar\_Grafo(int *n*);

void menu();

/\*Impressões\*/

void Imprimir\_Matriz\_adjacente(*Grafo* \**G*, int *n*);

void Imprimir\_Menor\_Caminho\_Bellman(int *vertices*, int *destino*, int *verticeAnterior*[], int *distancia*[]);

void Imprimir\_conteudo\_do\_vertice(*Grafo* \**G*, int *linha*);

void Inserir\_vertices\_e\_arestas(*Grafo* \**G*);

void bellmanFord(int \*\**grafo*, int *vertices*, int *origem*, int *destino*);

void Liberar\_Grafo(*Grafo* \*\**G*, int *qtdVertice*);

/\*Jogo\*/

void Vertices\_adjacentes(int \**vetorVertice*, int \**vetorResposta*, int \**cont*);

int Verificar\_Vitoria(*Grafo* \**G*, int *vertice*, int *pinoDesejado*);

int Verificar\_Movimento\_Valido(int \**vetorResposta*, int *vertice*);

void jogar(*Grafo* \**G*, int *vertice*);

*Grafo* \*Criar\_Grafo(int *n*){

    // n é o numero de vertices ou configuracoes possiveis

*Grafo* \*G;

    G = (*Grafo* \*) malloc(sizeof(*Grafo*));

    // Alocação de memoria para o vetor de vertices

    G->vertices = (*Vertice* \*) malloc(*n* \* sizeof(*Vertice*));

    // Alocação de memoria para a matriz de arestas

    G->arestas = (int \*\*) malloc(*n* \* sizeof(int \*));

    for(int i = 0; i < *n*; i++){

        G->arestas[i] = (int \*) calloc(*n*, sizeof(int));

    }

    return G;

}

void Imprimir\_Matriz\_adjacente(*Grafo* \**G*, int *n*){

    printf("Vertices: \n");

    for(int i = 0; i < *n*; i++){

        printf(" %d", i);

    }

    printf("\n\n");

    for(int i = 0; i < *n*; i++){

        printf("%d  ", i);

        for(int j = 0; j < *n*; j++){

            printf("%d ", *G*->arestas[i][j]);

        }

        printf("\n");

    }

    printf("\n");

}

void Inserir\_vertices\_e\_arestas(*Grafo* \**G*){

*FILE* \*verticesEarestas;

    verticesEarestas = fopen("../torre\_hanoi.txt", "r");

    if(verticesEarestas == NULL){

        printf("Error ao localizar o arquivo desejado!!");

    }else{

        int verticeAtual = 0, vermelho, amarelo, verde, rosa, aresta1, aresta2, aresta3;

        // Inserir os valores dos vertices no grafo a partir do arquivo ../

        while (fscanf(verticesEarestas, "%d %d %d %d %d %d %d %d", &verticeAtual, &vermelho, &amarelo, &verde, &rosa, &aresta1, &aresta2, &aresta3) != EOF){

        //preenche os pinos dos vertices

*G*->vertices[verticeAtual].discos[0] = vermelho;

*G*->vertices[verticeAtual].discos[1] = amarelo;

*G*->vertices[verticeAtual].discos[2] = verde;

*G*->vertices[verticeAtual].discos[3] = rosa;

        // marca a ligação das arestas na matriz adjacente

*G*->arestas[verticeAtual][aresta1] = 1;

*G*->arestas[verticeAtual][aresta2] = 1;

        if(aresta3 != 9999)

*G*->arestas[verticeAtual][aresta3] = 1;

    }

    }

}

void Imprimir\_conteudo\_do\_vertice(*Grafo* \**G*, int *linha*){

    printf("%d: (%d, %d, %d, %d)\n", *linha*, *G*->vertices[*linha*].discos[0], *G*->vertices[*linha*].discos[1], *G*->vertices[*linha*].discos[2], *G*->vertices[*linha*].discos[3]);

}

void bellmanFord(int \*\**grafo*, int *vertices*, int *origem*, int *destino*) {

*LARGE\_INTEGER* start\_time, end\_time, frequency;

    double elapsed\_time\_ms;

    int possuiCicloNegativo = 0;

    int \*distancia, \*verticeAnterior;

    distancia = (int\*)malloc(*vertices* \* sizeof(int));

    verticeAnterior = (int\*)malloc(*vertices* \* sizeof(int));

    // Os arrays de distância e vértices anteriores são inicializados com valores que indicam a

    // ausência de ligação(infinito) ou que a posição do vetor ainda não foi utilizada(-1)

    for (int i = 0; i < *vertices*; i++) {

        distancia[i] = INF;

        verticeAnterior[i] = -1;

    }

    // A distância do vértice de origem para ele mesmo é zero

    distancia[*origem*] = 0;

    QueryPerformanceFrequency(&frequency);

    QueryPerformanceCounter(&start\_time);

    // Encontra o caminho mais curto partindo do vértice de origem para todos os demais vértices

    for (int qtdRelaxamentos = 0; qtdRelaxamentos < *vertices* - 1; qtdRelaxamentos++) {

        for (int i = 0; i < *vertices*; i++) {

            for (int j = 0; j < *vertices*; j++) {

                if (*grafo*[i][j] && distancia[i] != INF && distancia[i] + *grafo*[i][j] < distancia[j]) {

                    distancia[j] = distancia[i] + *grafo*[i][j];

                    verticeAnterior[j] = i;

                }

            }

        }

    }

    // Checa se existe um ciclo negativo no grafo,

    // caso exista não é possível encontrar o menor caminho no grafo devido ás limitações do algoritmo

    for (int i = 0; i < *vertices*; i++) {

        for (int j = 0; j < *vertices*; j++) {

            if (*grafo*[i][j] && distancia[i] != INF && distancia[i] + *grafo*[i][j] < distancia[j]) {

                printf("O grafo contem um ciclo negativo.\n");

                possuiCicloNegativo = 1;

                j = *vertices*;

                i = j;

            }

        }

    }

    QueryPerformanceCounter(&end\_time);

    elapsed\_time\_ms = ((double)(end\_time.QuadPart - start\_time.QuadPart) \* 1000.0) / frequency.QuadPart;

    printf("Tempo decorrido: %.5f ms\n", elapsed\_time\_ms);

    // caso não exista um ciclo negativo, imprime o menor caminho

    if(!possuiCicloNegativo)

        Imprimir\_Menor\_Caminho\_Bellman(*vertices*,*destino*,verticeAnterior,distancia);

    free(distancia);

    free(verticeAnterior);

}

void Imprimir\_Menor\_Caminho\_Bellman(int *vertices*, int *destino*, int *verticeAnterior*[], int *distancia*[]){

    // Mostrar o caminho mais curto para o vértice de destino como também a distância

    printf("\nCaminho mais curto: ");

    int comprimentocaminho = 0;

    int \*caminhovertices; caminhovertices = (int\*)malloc(*vertices* \* sizeof(int));

    int verticeatual = *destino*;

    while (verticeatual != -1) {

        caminhovertices[comprimentocaminho++] = verticeatual;

        verticeatual = *verticeAnterior*[verticeatual];

    }

    for (int i = comprimentocaminho - 1; i >= 0; i--) {

        printf("%d ", caminhovertices[i]);

        if (i > 0)

            printf("-> ");

    }

    printf("\nDistancia: %d\n", *distancia*[*destino*]);

    free(caminhovertices);

}

void Liberar\_Grafo(*Grafo* \*\**G*, int *qtdVertice*){

    for (int i = 0; i < *qtdVertice*; i++){

        free((\**G*)->arestas[i]);

        (\**G*)->arestas[i] = NULL;

    }

    free((\**G*)->vertices);

    (\**G*)->vertices = NULL;

    free(\**G*);

    \**G* = NULL;

}

// jogo

// verifica resposta informa para quais arestas um vertice pode ir

void Vertices\_adjacentes(int \**vetorVertice*, int \**vetorResposta*, int \**cont*){

    \**cont* = 0;

    for(int i = 0; i < 81; i++){

        if(*vetorVertice*[i] == 1){

            if(\**cont* == 0)

*vetorResposta*[0] = i;

            if(\**cont* == 1)

*vetorResposta*[1] = i;

            if(\**cont* == 2)

*vetorResposta*[2] = i;

            \**cont* += 1;

        }

    }

}

// verificar se todos os pinos estão no 3 pino, retorna 4 caso estejam.

int Verificar\_Vitoria(*Grafo* \**G*, int *vertice*, int *pinoDesejado*){

    int incremento = 0;

    for (int i = 0; i < 4; i++){

        if(*G*->vertices[*vertice*].discos[i] == *pinoDesejado*)

        {

            incremento += 1;

        }

    }

    return incremento; // Retorna 1 se todos os discos estiverem no pino desejado, 0 caso contrário.

}

// verifica se o vertice informado é uma opção para onde ir

int Verificar\_Movimento\_Valido(int \**vetorResposta*, int *vertice*){

    int cont = 1;

    if (*vertice* == *vetorResposta*[0])

        cont = 0;

    if (*vertice* == *vetorResposta*[1])

        cont = 0;

    if (*vertice* == *vetorResposta*[2])

        cont = 0;

    return cont;

}

// Essa função permite que o usuário jogue o jogo

void jogar(*Grafo* \**G*, int *vertice*){

    int numMovimentos = 0;

    int vetorResposta[3], cont, flagParada = 1;

    do{

         // Esse while continuar até que o usuário ganhe ou desista

        printf("\nVertice atual e conteudo.\n ");

        Imprimir\_conteudo\_do\_vertice(*G*, *vertice*);

        printf("Possibilidades de para onde pode ir: \n");

        // zera o vetor de resposta

        memset(vetorResposta, 0, sizeof(vetorResposta));

        // verifica resposta informa para quais arestas um vertice pode ir

        Vertices\_adjacentes(*G*->arestas[*vertice*], vetorResposta, &cont);

        Imprimir\_conteudo\_do\_vertice(*G*, vetorResposta[0]);

        Imprimir\_conteudo\_do\_vertice(*G*, vetorResposta[1]);

        if(cont == 3){

            Imprimir\_conteudo\_do\_vertice(*G*, vetorResposta[2]);

        }

        printf("Deseja continuar jogando? 1 - sim, 0 - nao: ");

        scanf("%d", &flagParada);

        // entra nesse if se o usuário deseja continuar jogando, se não entrar isso significa que ele desistiu.

        if(flagParada == 1){

            printf("\nDigite o indice do vertice que deseja ir: ");

            scanf("%d", &*vertice*);

            while ((*vertice* < 0 || *vertice* > 80) || (Verificar\_Movimento\_Valido(vetorResposta, *vertice*))){

                printf("Indice de vertice nao existe ou foi informado um vertice que nao tem ligacao com o vertice atual.\n");

                printf("Digite novamente o vertice que deseja ir: ");

                scanf("%d", &*vertice*);

            }

            numMovimentos++;

        }else if(flagParada == 0){

            printf("Voce desistiu em %d movimentos\n", numMovimentos);

        }else{

            printf("Opcao invalida\n");

        }

    }while((Verificar\_Vitoria(*G*, *vertice*, 1) != 4) && (Verificar\_Vitoria(*G*, *vertice*, 2) != 4) && (Verificar\_Vitoria(*G*, *vertice*, 3) != 4) && (flagParada != 0));

    if(flagParada == 1)

        printf("Voce venceu em %d movimentos\n", numMovimentos);

}

void menu()

{

    printf("======================================MENU=======================================\n");

    printf(": 1 - Imprimir valores de todos os vertices                                     :\n");

    printf(": 2 - Imprimir um vertice                                                       :\n");

    printf(": 3 - Imprimir matriz adjacente arestas                                         :\n");

    printf(": 4 - Imprimir para onde as arestas de vertice estao ligadas                    :\n");

    printf(": 5 - Ford-Moore-Bellman                                                        :\n");

    printf(": 6 - Jogar                                                                     :\n");

    printf(": 0 - Sair                                                                      :\n");

    printf("=================================================================================\n");

}

int main(){

*Grafo* \*G;

    int op, vertice, opJogo, cont, vetorResposta[3], origem, destino;

    G = Criar\_Grafo(81);

    Inserir\_vertices\_e\_arestas(G);

    do{

        menu();

        scanf("%d", &op);

        switch (op){

        case 0:

            printf("Saindo...\n");

            break;

        case 1:

            for(int i = 0; i < TAM; i++)

                Imprimir\_conteudo\_do\_vertice(G, i); // imprimir valores dos vertices

            break;

        case 2:

            printf("Digite o indice do vertice: ");

            scanf("%d", &*vertice*);

            if(vertice >= 0 && vertice <= 80)

                Imprimir\_conteudo\_do\_vertice(G, vertice);

            else

                printf("Indice de vertice nao existe.\n");

            break;

        case 3:

            Imprimir\_Matriz\_adjacente(G, TAM); //imprimir arestas

            break;

        case 4:

            cont = 0;

            printf("Digite o indice do vertice: ");

            scanf("%d", &*vertice*);

            if(vertice >= 0 && vertice <= 80){

                Vertices\_adjacentes(G->arestas[vertice], vetorResposta, &cont);

                Imprimir\_conteudo\_do\_vertice(G, vetorResposta[0]);

                Imprimir\_conteudo\_do\_vertice(G, vetorResposta[1]);

                if(cont == 3){

                    Imprimir\_conteudo\_do\_vertice(G, vetorResposta[2]);

                }

                memset(vetorResposta, 0, sizeof(vetorResposta));

            }else

                printf("Indice de vertice nao existe.\n");

            break;

        case 5:

            printf("Ford-Moore-Bellman\n");

            printf("Digite o vertice de origem: ");

            scanf("%d", &*origem*);

            printf("Digite o vertice de destino: ");

            scanf("%d", &*destino*);

            if((origem >= 0 && origem <= 80) && (destino >= 0 && destino <= 80) )

                bellmanFord(G->*arestas*, TAM, origem, destino);

            else

                printf("Verifique se os vertices informados existem\n");

            break;

        case 6:

            printf("\nDeseja comecar de algum vertice? 1 - sim, 2 - nao: ");

            scanf("%d", &*opJogo*);

            if(opJogo == 1){

                printf("\nDigite o indice do vertice: ");

                scanf("%d", &vertice);

                if(vertice >= 0 && vertice <= 80)

                    jogar(G, vertice);

                else

                    printf("\nIndice de vertice nao existe.\n");

            }else if(opJogo == 2)

                jogar(G, 0);

            else

                printf("\nOpcao invalida\n");

            break;

        default:

            printf("Opcao invalida.\n");

            break;

        }

    }while (op != 0);

    Liberar\_Grafo(&G, 81);

    return 0;

}

**Questão 2:**

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

#include <limits.h>

#define MAX\_VERTICES 1000

/\*Prototipos\*/

typedef struct *aresta* *Aresta*;

typedef struct *grafo* *Grafo*;

typedef struct *filaprioridade* *FilaPrioridade*;

/\*Structs\*/

typedef struct *aresta*{

    int vertice;

    double confiabilidade;

} *Aresta*;

typedef struct *grafo*{

*Aresta* \*arestas;

    int num\_arestas;

} *Grafo*;

typedef struct *filaprioridade*{

    double confiabilidade;

    int vertice;

} *FilaPrioridade*;

/\*Declaração das funções\*/

void Liberar\_Grafo(*Grafo* \**grafo*, int *num\_vertices*);

*Grafo*\* Criar\_Grafo(int *num\_vertices*);

void Imprimir\_Caminho\_Mais\_Confiavel(const int \**predecessores*, int *origem*, int *destino*);

void Liberar\_Dijkstra\_Memoria(bool \**visitados*, double \**confiabilidade*, *FilaPrioridade* \**fp*, int \**predecessores*);

void dijkstra(const *Grafo* \**grafo*, int *origem*, int *destino*, int *num\_vertices*);

void Inserir\_Arestas(*Grafo* \**grafo*, int *u*, int *v*, double *confiabilidade*);

void Ler\_Arestas\_Do\_Arquivo(*FILE* \**arquivo*, *Grafo* \**grafo*, int *num\_arestas*);

// Função para liberar a memória alocada para o grafo

void Liberar\_Grafo(*Grafo* \**grafo*, int *num\_vertices*) {

    for (int i = 0; i < *num\_vertices*; ++i) {

        free(*grafo*[i].arestas);

    }

    free(*grafo*);

}

// Função para preencher o grafo a partir de um arquivo

*Grafo*\* Criar\_Grafo(int *num\_vertices*) {

*Grafo* \*grafo = (*Grafo* \*)malloc(*num\_vertices* \* sizeof(*Grafo*));

    for (int i = 0; i < *num\_vertices*; ++i) {

        grafo[i].num\_arestas = 0;

        grafo[i].arestas = NULL;

    }

    return grafo;

}

void Ler\_Arestas\_Do\_Arquivo(*FILE* \**arquivo*, *Grafo* \**grafo*, int *num\_arestas*) {

    for (int i = 0; i < *num\_arestas*; ++i) {

        int u, v;

        double confiabilidade;

        fscanf(*arquivo*, "%d %d %lf", &u, &v, &confiabilidade);

        Inserir\_Arestas(*grafo*, u, v, confiabilidade);

    }

}

void Inserir\_Arestas(*Grafo* \**grafo*, int *u*, int *v*, double *confiabilidade*) {

*grafo*[*u*].arestas = (*Aresta* \*)realloc(*grafo*[*u*].arestas, (*grafo*[*u*].num\_arestas + 1) \* sizeof(*Aresta*));

*grafo*[*u*].arestas[*grafo*[*u*].num\_arestas++] = (*Aresta*){*v*, *confiabilidade*};

}

// Função para imprimir o caminho mais confiável

void Imprimir\_Caminho\_Mais\_Confiavel(const int \**predecessores*, int *origem*, int *destino*) {

    if (*destino* == *origem*) {

        printf("%d", *origem*);

    } else if (*predecessores*[*destino*] == -1) {

        printf("Caminho inexistente entre %d e %d\n", *origem*, *destino*);

    } else {

        Imprimir\_Caminho\_Mais\_Confiavel(*predecessores*, *origem*, *predecessores*[*destino*]);

        printf(" -> %d", *destino*);

    }

}

// Função para liberar a memória alocada durante o algoritmo de Dijkstra

void Liberar\_Dijkstra\_Memoria(bool \**visitados*, double \**confiabilidade*, *FilaPrioridade* \**fp*, int \**predecessores*) {

    free(*visitados*);

    free(*confiabilidade*);

    free(*fp*);

    free(*predecessores*);

}

// Função principal do algoritmo de Dijkstra

void dijkstra(const *Grafo* \**grafo*, int *origem*, int *destino*, int *num\_vertices*) {

    bool \*visitados = (bool \*)malloc(*num\_vertices* \* sizeof(bool));

    double \*confiabilidade = (double \*)malloc(*num\_vertices* \* sizeof(double));

    int \*predecessores = (int \*)malloc(*num\_vertices* \* sizeof(int));

    for (int i = 0; i < *num\_vertices*; ++i) {

        visitados[i] = false;

        confiabilidade[i] = 0.0;

        predecessores[i] = -1;

    }

    confiabilidade[*origem*] = 1.0;

*FilaPrioridade* \*fp = (*FilaPrioridade* \*)malloc(*num\_vertices* \* sizeof(*FilaPrioridade*));

    int tamanho\_fp = 0;

    fp[tamanho\_fp++] = (*FilaPrioridade*){1.0, *origem*};

    while (tamanho\_fp > 0) {

*FilaPrioridade* min\_confiab = fp[0];

        int min\_idx = 0;

        for (int i = 1; i < tamanho\_fp; ++i) {

            if (fp[i].confiabilidade > min\_confiab.confiabilidade) {

                min\_confiab = fp[i];

                min\_idx = i;

            }

        }

        fp[min\_idx] = fp[--tamanho\_fp];

        visitados[min\_confiab.vertice] = true;

        if (min\_confiab.vertice == *destino*) {

            printf("O caminho mais confiavel entre %d e %d tem confiabilidade %.4lf\n", *origem*, *destino*, min\_confiab.confiabilidade);

            printf("Caminho: ");

            Imprimir\_Caminho\_Mais\_Confiavel(predecessores, *origem*, *destino*);

            printf("\n");

            Liberar\_Dijkstra\_Memoria(visitados, confiabilidade, fp, predecessores);

            return;

        }

        for (int i = 0; i < *grafo*[min\_confiab.vertice].num\_arestas; ++i) {

*Aresta* aresta = *grafo*[min\_confiab.vertice].arestas[i];

            if (!visitados[aresta.vertice]) {

                double nova\_confiab = min\_confiab.confiabilidade \* aresta.confiabilidade;

                if (nova\_confiab > confiabilidade[aresta.vertice]) {

                    confiabilidade[aresta.vertice] = nova\_confiab;

                    predecessores[aresta.vertice] = min\_confiab.vertice;

                    fp[tamanho\_fp++] = (*FilaPrioridade*){nova\_confiab, aresta.vertice};

                }

            }

        }

    }

    printf("Nao ha caminho confiavel entre %d e %d\n", *origem*, *destino*);

    Liberar\_Dijkstra\_Memoria(visitados, confiabilidade, fp, predecessores);

}

int main() {

*FILE* \*arquivo = fopen("../Grafo.txt", "r");

    if (arquivo == NULL) {

        printf("Erro ao abrir o arquivo.\n");

        return 1;

    }

    int num\_vertices, num\_arestas;

    fscanf(arquivo, "%d %d", &num\_vertices, &num\_arestas);

*Grafo* \*grafo = Criar\_Grafo(num\_vertices);

    Ler\_Arestas\_Do\_Arquivo(arquivo, grafo, num\_arestas \* 2);

    int origem, destino;

    printf("Digite o vertice de origem e o vertice de destino: ");

    scanf("%d %d", &origem, &destino);

    if (origem < 0 || origem >= num\_vertices || destino < 0 || destino >= num\_vertices) {

        printf("Vertices invalidos. Certifique-se de que os vertices inseridos existem no grafo.\n");

    } else {

        dijkstra(grafo, origem, destino, num\_vertices);

    }

    Liberar\_Grafo(grafo, num\_vertices);

    fclose(arquivo);

    return 0;

}

**Questão 3:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX101 101

#define MAX150 150

/\* Prototypes \*/

typedef struct *func* *Func*;

void imprimirFunc(*Func* \**tabela*, int *MAX*);

void inicializarTabela(*Func* \**f*, int *MAX*);

int colisaoRotacao(int *chave*, char *primeiroDigito*, int *MAX*);

int funcaoHashStringRotacao(char \**str*, int *MAX*);

void inserirTabelaRotacao(*Func* \**tabela*, int \**numColisoes*, int \**countTabela*, int *MAX*);

int colisaoFoldShift(int *chave*, int *MAX*);

int funcaoHashStringFoldShift(char \**str*, int *MAX*);

void inserirTabelaFoldShift(*Func* \**tabela*, int \**numColisoes*, int \**countTabela*, int *MAX*);

/\* Struct \*/

typedef struct *func*

{

    char matricula[7];

    char nome[50];

    char funcao[20];

    float salario;

} *Func*;

/\* Imprimir \*/

void imprimirFunc(*Func* \**tabela*, int *MAX*)

{

    for (int i = 0; i < *MAX*; i++)

    {

        if (*tabela*[i].matricula[0] != '\0')

        {

            printf("Matricula: %s\n", *tabela*[i].matricula);

            printf("Nome: %s\n", *tabela*[i].nome);

            printf("Funcao: %s\n", *tabela*[i].funcao);

            printf("Salario: %.2f\n", *tabela*[i].salario);

            printf("\n");

        }

    }

}

/\* Leitura dos dados \*/

*Func* lerFunc()

{

*Func* f;

    while (strlen(f.matricula) < 6 || strlen(f.matricula) > 6)

    {

        printf("Informe a matricula do funcionario: ");

        scanf(" *%*[^\n]", f.matricula);

    }

    printf("Informe o nome do funcionario: ");

    scanf(" *%*[^\n]", f.nome);

    printf("Informe a funcao do funcionario: ");

    scanf(" *%*[^\n]", f.funcao);

    printf("Informe o salario do funcionario: ");

    scanf("%f", &f.salario);

    return f;

}

/\* Funcoes Hash \*/

void inicializarTabela(*Func* \**f*, int *MAX*)

{

    for (int i = 0; i < *MAX*; i++)

    {

        strcpy(*f*[i].matricula, "\0");

    }

}

/\*Função hash rotação\*/

int colisaoRotacao(int *chave*, char *primeiroDigito*, int *MAX*)

{

    return (*chave* + (*primeiroDigito* - '0'))% *MAX*;

}

int funcaoHashStringRotacao(char \**str*, int *MAX*)

{

    char chave[7];

    sprintf(chave, "%c%c%s", *str*[4], *str*[5], *str*);

    int d2 = chave[1] - '0';

    int d4 = chave[3] - '0';

    int d6 = chave[5] - '0';

    int valor\_hash = d2 \* 100 + d4 \* 10 + d6;

    int posicao = valor\_hash % *MAX*;

    return posicao;

}

void inserirTabelaRotacao(*Func* \**tabela*, int \**numColisoes*, int \**countTabela*, int *MAX*)

{

*Func* f = lerFunc();

    int indice = funcaoHashStringRotacao(f.matricula, *MAX*);

    if ((\**countTabela*) < *MAX*)

    {

        if (*tabela*[indice].matricula[0] != '\0')

        {

            (\**numColisoes*)++;

            int novo = colisaoRotacao(indice, f.matricula[0], *MAX*);

            printf("\n\n\nnovo:%d\n\n\n", novo);

            while (*tabela*[novo].matricula[0] != '\0')

            {

                (\**numColisoes*)++;

                novo = colisaoRotacao(novo, f.matricula[0], *MAX*);

            }

*tabela*[novo] = f;

            (\**countTabela*)++;

        }

        else

        {

            (\**countTabela*)++;

*tabela*[indice] = f;

        }

    }else

    {

        (\**countTabela*)++;

*tabela*[indice] = f;

    }

}

/\*Função hash fold shift\*/

int colisaoFoldShift(int *chave*, int *MAX*)

{

    return (*chave* + 7) % *MAX*;

}

int funcaoHashStringFoldShift(char \**str*, int *MAX*)

{

    int tam = strlen(*str*);

    unsigned int hash = 0;

    unsigned int aux1 = 0;

    unsigned int aux2 = 0;

    for (int i = 0; i < tam; i++)

    {

        if (i == 0 || i == 2 || i == 5)

            aux1 = aux1 \* 10 + (*str*[i] - '0');

        else

            aux2 = aux2 \* 10 + (*str*[i] - '0');

    }

    hash = (aux1 + aux2) % *MAX*;

    return hash;

}

void inserirTabelaFoldShift(*Func* \**tabela*, int \**numColisoes*, int \**countTabela*, int *MAX*)

{

*Func* f = lerFunc();

    int indice = funcaoHashStringFoldShift(f.matricula, *MAX*);

    if ((\**countTabela*) < *MAX*)

    {

        if (*tabela*[indice].matricula[0] != '\0')

        {

            (\**numColisoes*)++;

            int novo = colisaoFoldShift(indice, *MAX*);

            while (*tabela*[novo].matricula[0] != '\0')

            {

                (\**numColisoes*)++;

                novo = colisaoFoldShift(novo, *MAX*);

            }

*tabela*[novo] = f;

            (\**countTabela*)++;

        }

        else

        {

            (\**countTabela*)++;

*tabela*[indice] = f;

        }

    }else

    {

        (\**countTabela*)++;

*tabela*[indice] = f;

    }

}

void menu\_tabela()

{

    printf("========== MENU ==========\n");

    printf("1 - Tabela de 101 posicoes\n");

    printf("2 - Tabela de 150 posicoes\n");

    printf("0 - Sair                  \n");

    printf("==========================\n");

}

void menu\_hash()

{

    printf("========== MENU ==========\n");

    printf("1 - Hash rotacao          \n");

    printf("2 - Hash fold shift       \n");

    printf("0 - Sair                  \n");

    printf("==========================\n");

}

void menu\_opcoes()

{

    printf("========== MENU ==========\n");

    printf("1 - Inserir funcionarios  \n");

    printf("2 - Imprimir funcionarios \n");

    printf("3 - Colisoes totais       \n");

    printf("0 - Voltar                \n");

    printf("==========================\n");

}

int main()

{

*Func* \*tabela101 = (*Func* \*)malloc(MAX101 \* sizeof(*Func*));

*Func* \*tabela150 = (*Func* \*)malloc(MAX150 \* sizeof(*Func*));

    int numColisoes101 = 0;

    int numColisoes150 = 0;

    int countTabela101 = 0;

    int countTabela150 = 0;

    int op = -1;

    while(op != 0)

    {

        menu\_tabela();

        scanf("%d", &op);

        switch (op)

        {

        case 1:

            {

                menu\_hash();

                scanf("%d", &op);

                switch (op)

                {

                case 1:

                    inicializarTabela(tabela101, MAX101);

                    op = -1;

                    while (op != 0)

                    {

                        op = -1;

                        menu\_opcoes();

                        scanf("%d", &op);

                        switch (op)

                        {

                        case 1:

                            inserirTabelaRotacao(tabela101, &*numColisoes101*, &*countTabela101*, MAX101);

                            break;

                        case 2:

                            imprimirFunc(tabela101, MAX101);

                            break;

                        case 3:

                            printf("Numero total de colisoes: %d\n", numColisoes101);

                            break;

                        default:

                            break;

                        }

                    }

                    numColisoes101 = 0;

                    countTabela101 = 0;

                    op = -1;

                    break;

                case 2:

                    inicializarTabela(tabela101, MAX101);

                    while (op != 0)

                    {

                        op = -1;

                        menu\_opcoes();

                        scanf("%d", &op);

                        switch (op)

                        {

                        case 1:

                            inserirTabelaFoldShift(tabela101, &*numColisoes101*, &*countTabela101*, MAX101);

                            break;

                        case 2:

                            imprimirFunc(tabela101, MAX101);

                            break;

                        case 3:

                            printf("Numero total de colisoes: %d\n", numColisoes101);

                            break;

                        default:

                            break;

                        }

                    }

                    numColisoes101 = 0;

                    countTabela101 = 0;

                    op = -1;

                    break;

                default:

                    break;

                }

            }

            op = -1;

            break;

        case 2:

            {

                menu\_hash();

                scanf("%d", &op);

                switch (op)

                {

                case 1:

                    inicializarTabela(tabela150, MAX150);

                    op = -1;

                    while (op != 0)

                    {

                        op = -1;

                        menu\_opcoes();

                        scanf("%d", &op);

                        switch (op)

                        {

                        case 1:

                            inserirTabelaRotacao(tabela150, &*numColisoes150*, &*countTabela150*, MAX150);

                            break;

                        case 2:

                            imprimirFunc(tabela150, MAX150);

                            break;

                        case 3:

                            printf("Numero total de colisoes: %d\n", numColisoes150);

                            break;

                        default:

                            break;

                        }

                    }

                    numColisoes150 = 0;

                    countTabela150 = 0;

                    op = -1;

                    break;

                case 2:

                    inicializarTabela(tabela150, MAX150);

                    while (op != 0)

                    {

                        op = -1;

                        menu\_opcoes();

                        scanf("%d", &op);

                        switch (op)

                        {

                        case 1:

                            inserirTabelaFoldShift(tabela150, &*numColisoes150*, &*countTabela150*, MAX150);

                            break;

                        case 2:

                            imprimirFunc(tabela150, MAX150);

                            break;

                        case 3:

                            printf("Numero total de colisoes: %d\n", numColisoes150);

                            break;

                        default:

                            break;

                        }

                    }

                    numColisoes150 = 0;

                    countTabela150 = 0;

                    op = -1;

                    break;

                default:

                    break;

                }

            }

        default:

            break;

        }

    }

    free(tabela101);

    free(tabela150);

    return 0;

}

**Questão 3.1:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

#define MAX101 101

#define MAX150 150

typedef struct *func* {

    char matricula[7];

} *Func*;

char\* matricula\_aleatoria() {

    char\* matricula = malloc(7 \* sizeof(char));

    for (int i = 0; i < 6; i++) {

        matricula[i] = '0' + rand() % 10;

    }

    matricula[6] = '\0';

    return matricula;

}

void inicializarTabela(*Func* \**f*, int *MAX*) {

    for (int i = 0; i < *MAX*; i++) {

        strcpy(*f*[i].matricula, "\0");

    }

}

int colisaoRotacao(int *chave*, char *primeiroDigito*, int *MAX*) {

    return (*chave* + (*primeiroDigito* - '0')) % *MAX*;

}

int funcaoHashStringRotacao(char \**str*) {

    char chave[7];

    sprintf(chave, "%c%c%s", *str*[4], *str*[5], *str*);

    int d2 = chave[1] - '0';

    int d4 = chave[3] - '0';

    int d6 = chave[5] - '0';

    int valor\_hash = d2 \* 100 + d4 \* 10 + d6;

    return valor\_hash % MAX101;

}

void inserirTabelaRotacao(*Func* \**tabela*, int \**numColisoes*, int \**countTabela*, int *MAX*, int *num*) {

*Func* f;

    snprintf(f.matricula, sizeof(f.matricula), "%d", *num*);

    int indice = funcaoHashStringRotacao(f.matricula);

    if ((\**countTabela*) < *MAX*) {

        if (*tabela*[indice].matricula[0] != '\0') {

            (\**numColisoes*)++;

            int novo = colisaoRotacao(indice, f.matricula[0], *MAX*);

            while (*tabela*[novo].matricula[0] != '\0') {

                (\**numColisoes*)++;

                novo = colisaoRotacao(novo, f.matricula[0], *MAX*);

            }

*tabela*[novo] = f;

            (\**countTabela*)++;

        } else {

            (\**countTabela*)++;

*tabela*[indice] = f;

        }

    } else {

        (\**countTabela*)++;

*tabela*[indice] = f;

    }

}

int colisaoFoldShift(int *chave*, int *MAX*) {

    return (*chave* + 7) % *MAX*;

}

int funcaoHashStringFoldShift(char \**str*) {

    int tam = strlen(*str*);

    unsigned int hash = 0;

    unsigned int aux1 = 0;

    unsigned int aux2 = 0;

    for (int i = 0; i < tam; i++) {

        if (i == 0 || i == 2 || i == 5)

            aux1 = aux1 \* 10 + (*str*[i] - '0');

        else

            aux2 = aux2 \* 10 + (*str*[i] - '0');

    }

    hash = (aux1 + aux2) % MAX101;

    return hash;

}

void inserirTabelaFoldShift(*Func* \**tabela*, int \**numColisoes*, int \**countTabela*, int *MAX*, int *num*) {

*Func* f;

    snprintf(f.matricula, sizeof(f.matricula), "%d", *num*);

    int indice = funcaoHashStringFoldShift(f.matricula);

    if ((\**countTabela*) < *MAX*) {

        if (*tabela*[indice].matricula[0] != '\0') {

            (\**numColisoes*)++;

            int novo = colisaoFoldShift(indice, *MAX*);

            while (*tabela*[novo].matricula[0] != '\0') {

                (\**numColisoes*)++;

                novo = colisaoFoldShift(novo, *MAX*);

            }

*tabela*[novo] = f;

            (\**countTabela*)++;

        } else {

            (\**countTabela*)++;

*tabela*[indice] = f;

        }

    } else {

        (\**countTabela*)++;

*tabela*[indice] = f;

    }

}

void imprimirTabela(*Func* \**tabela*, int *MAX*, const char \**tableLabel*) {

    printf("\n%s:\n", *tableLabel*);

    for (int i = 0; i < *MAX*; i++) {

        printf("%d: %s\n", i, *tabela*[i].matricula);

    }

}

int main() {

*Func* \*tabela101 = (*Func* \*)malloc(MAX101 \* sizeof(*Func*));

*Func* \*tabela150 = (*Func* \*)malloc(MAX150 \* sizeof(*Func*));

    int numColisoesRotacao101 = 0;

    int numColisoesFoldShift101 = 0;

    int numColisoesRotacao150 = 0;

    int numColisoesFoldShift150 = 0;

    int countTabelaRotacao101 = 0;

    int countTabelaRotacao150 = 0;

    int countTabelaFoldShift101 = 0;

    int countTabelaFoldShift150 = 0;

    srand(time(NULL));

    inicializarTabela(tabela101, MAX101);

    inicializarTabela(tabela150, MAX150);

    int i = 0;

    while (i < 1000) {

        int num = rand() % 900000 + 100000;

        inserirTabelaRotacao(tabela101, &numColisoesRotacao101, &countTabelaRotacao101, MAX101, num);

        inserirTabelaRotacao(tabela150, &numColisoesRotacao150, &countTabelaRotacao150, MAX150, num);

        i++;

        // printf("\n\n\ni : %d\n\n\n", i);

    }

    imprimirTabela(tabela101, MAX101, "Table 101 (Rotacao)");

    imprimirTabela(tabela150, MAX150, "Table 150 (Rotacao)");

    inicializarTabela(tabela101, MAX101);

    inicializarTabela(tabela150, MAX150);

    i = 0;

    while (i < 1000) {

        int num = rand() % 900000 + 100000;

        inserirTabelaFoldShift(tabela101, &numColisoesFoldShift101, &countTabelaFoldShift101, MAX101, num);

        inserirTabelaFoldShift(tabela150, &numColisoesFoldShift150, &countTabelaFoldShift150, MAX150, num);

        i++;

        // printf("\n\n\naaaa : %d\n\n\n", i);

    }

    imprimirTabela(tabela101, MAX101, "Table 101 (FoldShift)");

    imprimirTabela(tabela150, MAX150, "Table 150 (FoldShift)");

    printf("\nColisoes Rotacao 101: %d || Num inseridos: %d\n", numColisoesRotacao101, countTabelaRotacao101);

    printf("Colisoes Fold Shift 101: %d || Num inseridos: %d\n", numColisoesFoldShift101, countTabelaFoldShift101);

    printf("Colisoes Rotacao 150: %d || Num inseridos: %d\n", numColisoesRotacao150, countTabelaRotacao150);

    printf("Colisoes Fold Shift 150: %d || Num inseridos: %d\n\n", numColisoesFoldShift150, countTabelaFoldShift150);

    free(tabela101);

    free(tabela150);

    return 0;

}