Documentação do primeiro trabalho prático da disciplina de Estrutura de Dados 1

Bruno Soares Veríssimo
Universidade Federal de Ouro Preto - UFOP

1 Introdução

O Problema do Caixeiro Viajante é um dos desafios mais conhecidos na área da computação. Ele envolve um caixeiro viajante que precisa visitar um conjunto de cidades distintas e retornar à cidade de origem, buscando encontrar a trajetória que resulta na menor distância total de viagem. O desafio é encontrar a ordem de visita das cidades que minimize a distância percorrida.
O problema é relevante em diversas aplicações do mundo real, como planejamento de rotas para entregas, logística, circuitos de computadores, entre outros. A sua complexidade aumenta exponencialmente com o número de cidades envolvidas, o que torna o desenvolvimento de algoritmos eficientes para resolvê-lo uma tarefa desafiadora.
Neste programa, apresentamos uma solução que utiliza a abordagem de força bruta e recursividade para encontrar o caminho mais curto. Isso implica na geração de todas as permutações possíveis das cidades e na avaliação de todas elas para determinar a solução.
A documentação a seguir descreve a implementação, os testes realizados, a análise dos resultados e as conclusões do programa. Além disso, apresenta detalhes sobre as principais funções e estruturas utilizadas na resolução do Problema do Caixeiro Viajante.
Compreender como essa abordagem funciona e suas limitações é fundamental para entender o funcionamento do programa e sua eficácia em diferentes cenários.

2 Implementação

2.1 Uso de Tipos Abstratos de Dados (TAD)

Para manter o código organizado e modular, foi adotada a abordagem de Tipos Abstratos de Dados (TAD). Isso ajuda a encapsular os detalhes da estrutura de dados usada para representar o grafo ponderado e as operações que podem ser executadas sobre ele. O TAD ”GrafoPonderado” é definido no arquivo grafo.h da seguinte maneira:

```
#ifndef GRAFO_H
#define GRAFO_H
typedef struct GrafoPonderado GrafoPonderado;
GrafoPonderado* alocarGrafo(int numCidades);
void desalocarGrafo(GrafoPonderado* grafo);
void leGrafo(GrafoPonderado* grafo);
void encontraCaminho(GrafoPonderado* grafo, int* melhorCaminho);
void imprimeCaminho(GrafoPonderado* grafo, int* caminho, int distancia);
int obterDistancia(GrafoPonderado* grafo, int cidadeOrigem, int cidadeDestin
#endif
```

Aqui, o TAD GrafoPonderado é uma estrutura opaca que contém detalhes internos ocultos da implementação do grafo ponderado. Isso permite que o programa principal (main.c) interaja com o grafo apenas por meio das funções definidas no TAD, garantindo a modularidade e a encapsulação de dados.

2.2 Recursividade

O programa utiliza recursão para encontrar o caminho mais curto em um grafo ponderado. A função principal encontraCaminho chama uma função auxiliar encontraCaminhoRec, que é responsável por explorar todas as possíveis permutações das cidades para encontrar o caminho mais curto. Aqui estão os principais aspectos do uso da recursividade na implementação:
A função encontraCaminhoRec é definida para explorar todas as permutações de cidades. Ela recebe a posição atual, a distância percorrida até o momento, o contador de cidades visitadas, arrays para rastrear as cidades visitadas e o caminho atual, o array para armazenar o melhor caminho encontrado, o número total de cidades e o grafo.
A base da recursão ocorre quando o contador de cidades visitadas atinge o número total de cidades menos um, o que significa que todas as cidades foram visitadas, e é hora de verificar se há um ciclo completo de volta à cidade de origem. Nesse ponto, a distância total é calculada e comparada com a menor distância registrada até o momento. Se um caminho mais curto for encontrado, ele é armazenado no array melhorCaminho.
Em seguida, a função percorre todas as cidades adjacentes à cidade atual, verifica se a cidade adjacente ainda não foi visitada e, se for o caso, marca-a como visitada, atualiza o caminho atual e a distância e faz uma chamada recursiva para explorar a próxima cidade. Após a recursão, a cidade adjacente é desmarcada como não visitada para permitir a exploração de outros caminhos.
Esse processo de recursão e exploração de todas as permutações continua até que todas as possibilidades tenham sido exploradas.
No final, a função principal encontraCaminho configura as estruturas necessárias, como o array visitado, chama a função encontraCaminhoRec, encontra o melhor caminho e, finalmente, libera a memória alocada para os arrays de controle.
Essa abordagem de recursividade e a divisão das tarefas entre as funções do TAD garantem que o programa seja modular, fácil de entender e que a busca pelo caminho mais curto seja realizada de maneira sistemática e eficaz. Assim, segue a implementação de grafo.c:

```
#include "grafo.h" // Inclui o cabe alho "grafo.h" que cont m as declara
#include <stdio.h> // Inclui a biblioteca padr o de entrada/sa da.
#include <stdlib.h> // Inclui a biblioteca padr o de aloca  o de mem ria
#include <limits.h> // Inclui a biblioteca que fornece constantes de limites

struct GrafoPonderado { // Define uma estrutura chamada GrafoPonderado para
    int numCidades; // N mero de cidades no grafo.
    int** matrizAdj; // Matriz de adjac ncia para representar as dist ncia
};

GrafoPonderado* alocarGrafo(int numCidades) { // Fun  o para alocar mem
    GrafoPonderado* grafo = (GrafoPonderado*)malloc(sizeof(GrafoPonderado))
    if (grafo == NULL) { // Verifica se a aloca  o foi bem-sucedida.
```

```
        // Tratamento de erro de aloca  o de mem ria.
        exit(1); // Encerra o programa com c digo de erro 1 ou toma alguma
    }
    grafo->numCidades = numCidades; // Inicializa o n mero de cidades no g
    grafo->matrizAdj = (int**)malloc(numCidades * sizeof(int*)); // Aloca m
    if (grafo->matrizAdj == NULL) { // Verifica se a aloca  o da matriz f
        // Tratamento de erro de aloca  o de mem ria.
        free(grafo); // Libera a mem ria alocada para a estrutura GrafoPon
        exit(1); // Encerra o programa com c digo de erro 1 ou toma alguma
    }
    for (int i = 0; i < numCidades; i++) {
        grafo->matrizAdj[i] = (int*)malloc(numCidades * sizeof(int)); // Al

        if (grafo->matrizAdj[i] == NULL) { // Verifica se a aloca  o das
            // Tratamento de erro de aloca  o de mem ria.
            free(grafo->matrizAdj); // Libera a mem ria alocada para a mat
            free(grafo); // Libera a mem ria alocada para a estrutura Grafo
            exit(1); // Encerra o programa com c digo de erro 1 ou toma alg
        }
    }
    // Inicializa a matriz de adjac ncia com dist ncias iniciais.
    for (int i = 0; i < numCidades; i++) {
        for (int j = 0; j < numCidades; j++) {
            grafo->matrizAdj[i][j] = 0; // Inicializa as dist ncias com 0.
        }
    }
    return grafo; // Retorna a estrutura GrafoPonderado alocada.
}

void desalocarGrafo(GrafoPonderado* grafo) {
    if (grafo != NULL) { // Verifica se o ponteiro para a estrutura GrafoPo
        if (grafo->matrizAdj != NULL) { // Verifica se o ponteiro para a ma
            for (int i = 0; i < grafo->numCidades; i++) { // Loop para perc
                free(grafo->matrizAdj[i]); // Libera a mem ria alocada para
            }
            free(grafo->matrizAdj); // Libera a mem ria alocada para a mat
        }
        free(grafo); // Libera a mem ria alocada para a estrutura GrafoPon
    }
}

void leGrafo(GrafoPonderado* grafo) {
    int numCidades = grafo->numCidades; // Obt m o n mero de cidades do g
    for (int i = 0; i < numCidades; i++) { // Loop externo para percorrer a
        for (int j = 0; j < numCidades; j++) { // Loop interno para percorr
```

```
            int origem, destino, distancia; // Declara vari veis para arma
            scanf("%d■%d■%d", &origem, &destino, &distancia); // L   tr s
            grafo->matrizAdj[origem][destino] = distancia; // Atualiza a ma
        }
    }
}

// Fun  o para obter a dist ncia entre duas cidades no grafo.
int obterDistancia(GrafoPonderado* grafo, int cidadeOrigem, int cidadeDestin
    // Verifica se as cidades fornecidas s o v lidas (dentro dos limites
    if (cidadeOrigem >= 0 && cidadeOrigem < grafo->numCidades && cidadeDesti
        // Retorna a dist ncia armazenada na matriz de adjac ncia para as
        return grafo->matrizAdj[cidadeOrigem][cidadeDestino];
    } else {
        // Se as cidades n o forem v lidas, retorna -1 para indicar um er
        return -1;
    }
}

// Fun  o recursiva para encontrar o melhor caminho em um grafo ponderado
void encontraCaminhoRec(int posicao, int distancia, int contador, int* visit
    // Verifica se o contador alcan ou o n mero total de cidades no grafo
    if (contador == numCidades - 1) {
        // Se o contador atingir o n mero de cidades - 1, significa que vis
        // Verifica se h  uma aresta de retorno para a cidade de origem (c
        if (grafo->matrizAdj[posicao][0] != 0 && distancia + grafo->matrizA
            // Se encontrarmos um caminho mais curto, atualizamos o menorDi
            *menorDistancia = distancia + grafo->matrizAdj[posicao][0];
            for (int i = 0; i < numCidades; i++) {
                melhorCaminho[i] = caminhoAtual[i];
            }
        }
        return; // Retorna da recurs o.
    }
    // Percorre as pr ximas cidades a serem visitadas a partir da posi  o
    for (int proximaCidade = 0; proximaCidade < numCidades; proximaCidade++
        if (grafo->matrizAdj[posicao][proximaCidade] != 0 && visitado[proxim
            // Verifica se existe uma aresta para a pr xima cidade e se ess
            // Marcamos a pr xima cidade como visitada, atualizamos o camin
            visitado[proximaCidade] = 1;
            caminhoAtual[contador] = proximaCidade;
            // Chamamos a fun  o recursivamente para explorar o pr ximo
            encontraCaminhoRec(proximaCidade, distancia + grafo->matrizAdj[
            // Desmarcamos a pr xima cidade como visitada ap s a recurs o
            visitado[proximaCidade] = 0;
```

```
        }
    }
}

// Fun  o para encontrar o caminho mais curto em um grafo ponderado usand
void encontraCaminho(GrafoPonderado* grafo, int* melhorCaminho) {
    int numCidades = grafo->numCidades;
    // Aloca um array para controlar as cidades visitadas.
    int* visitado = (int*)malloc(numCidades * sizeof(int));
    // Aloca um array para rastrear o caminho atual.
    int* caminhoAtual = (int*)malloc(numCidades * sizeof(int));
    // Inicializa a menorDistancia com um valor m ximo.
    int menorDistancia = INT_MAX;
    // Marca a cidade de origem (0) como visitada e inicia o caminho atual.
    for (int i = 0; i < numCidades; i++) {
        visitado[i] = 0;
    }
    visitado[0] = 1;
    caminhoAtual[0] = 0;
    // Chama a fun  o encontraCaminhoRec para encontrar o caminho mais cu
    encontraCaminhoRec(0, 0, 0, visitado, caminhoAtual, melhorCaminho, numC
    // Libera a mem ria alocada para os arrays de controle.
    free(visitado);
    free(caminhoAtual);
}

// Fun  o para imprimir o caminho mais curto encontrado e a dist ncia to
void imprimeCaminho(GrafoPonderado* grafo, int* caminho, int distancia) {
    // Imprime a cidade de origem (0) para iniciar o caminho.
    printf("0 ");
    // Percorre as cidades do caminho, exceto a  ltima , e as imprime.
    for (int i = 0; i < grafo->numCidades - 1; i++) {
        printf("%d ", caminho[i]);
    }
    // Imprime a cidade de origem (0) novamente para completar o ciclo.
    printf("0\n");
    // Imprime a dist ncia total do caminho encontrado.
    printf("%d\n", distancia);
}
```

2.3 main.c

A função main() é o ponto de entrada do programa, onde a execução é iniciada. Esta função é responsável por coordenar a execução geral do programa e interagir com o usuário. Aqui estão os principais elementos da função main():

```c
#include "grafo.h" // Inclui a defini  o das estruturas e fun  es do gr
#include <stdio.h> // Inclui a biblioteca padr o para entrada/sa da.
#include <stdlib.h> // Inclui a biblioteca padr o de aloca  o de mem ri
int main() {
    int numCidades;
    scanf("%d", &numCidades); // L  o n mero de cidades a partir da entra
    // Aloca espa o na mem ria para um grafo ponderado com 'numCidades'.
    GrafoPonderado* grafo = alocarGrafo(numCidades);
    // L  as informa  es das dist ncias entre as cidades e as armazena
    leGrafo(grafo);
    // Aloca mem ria para armazenar o melhor caminho encontrado, exceto a
    int* melhorCaminho = (int*)malloc((numCidades - 1) * sizeof(int));
    // Encontra o caminho mais curto atrav s do grafo.
    encontraCaminho(grafo, melhorCaminho);
    int distancia = 0; // Inicializa a vari vel que armazenar  a dist nci
    // Calcula a dist ncia total percorrida ao somar as dist ncias entre
    for (int i = 0; i < numCidades - 1; i++) {
        int cidadeOrigem = melhorCaminho[i];
        int cidadeDestino = melhorCaminho[i + 1];
        distancia += obterDistancia(grafo, cidadeOrigem, cidadeDestino);
    }
    // Adiciona a dist ncia entre a  ltima  cidade e a cidade de origem pa
    distancia += obterDistancia(grafo, melhorCaminho[numCidades - 1], melho
    // Imprime o caminho mais curto encontrado e a dist ncia total.
    imprimeCaminho(grafo, melhorCaminho, distancia);
    // Libera a mem ria alocada para o vetor do melhor caminho.
    free(melhorCaminho);
    // Libera a mem ria alocada para o grafo.
    desalocarGrafo(grafo);
    return 0; // Retorna 0 para indicar que o programa foi executado com su
}
```

int numCidades;: Declaração de uma variável numCidades do tipo inteiro. Esta variável será usada para armazenar o número de cidades no grafo.

scanf("

Alocação de Memória para o Grafo: A próxima etapa do código envolve a alocação de memória para representar o grafo ponderado. Isso é feito chamando a função alocarGrafo(numCidades), onde numCidades é o valor lido anteriormente. Essa alocação dinâmica de memória permite que o programa crie uma estrutura de dados para armazenar as informações do grafo.

Leitura das Informações do Grafo: Após a alocação da memória do grafo, o programa lê as informações das distâncias entre as cidades e as armazena na estrutura do grafo. Isso é realizado pela função leGrafo(grafo).

Alocação de Memória para o Caminho: Para armazenar o melhor caminho encontrado, excluindo a cidade de origem, o programa aloca memória para um vetor de inteiros chamado melhorCaminho.

A alocação é feita para o tamanho (numCidades - 1), uma vez que o caminho começa e termina na mesma cidade. Isso é realizado pela linha:

```
int* melhorCaminho = (int*)malloc((numCidades - 1) * sizeof(int));
```

Encontrando o Caminho Mais Curto: O programa chama a função encontraCaminho(grafo, melhorCaminho) para encontrar o caminho mais curto no grafo. Essa função utiliza uma abordagem de força bruta, considerando todas as permutações das cidades para encontrar o caminho mais curto.
Cálculo da Distância Total: Para calcular a distância total percorrida no caminho mais curto, o programa itera sobre as cidades no caminho e soma as distâncias entre elas. O resultado é armazenado na variável distancia.
Impressão do Resultado: Após encontrar o caminho mais curto e calcular a distância total, o programa imprime o caminho e a distância na saída padrão. Isso é realizado pela função imprimeCaminho(grafo, melhorCaminho, distancia).
Liberação de Memória: Para evitar vazamentos de memória, o programa libera a memória alocada dinamicamente para o vetor melhorCaminho e a estrutura do grafo. Isso é feito pelas linhas:

```
free(melhorCaminho);
desalocarGrafo(grafo);
```

Retorno de 0: A função main() retorna 0, o que indica que o programa foi executado com sucesso. Essa função main() é essencial para orquestrar todo o processo de encontrar o caminho mais curto em um grafo ponderado, desde a leitura dos dados de entrada até a impressão do resultado e a liberação de memória. Ela encapsula as etapas fundamentais do programa e permite que o usuário interaja com o software, fornecendo o número de cidades para o cálculo do caminho mais curto.

3 Testes

Os testes foram realizados com os arquivos disponibilizados pelo professor da disciplina. No caso foram disponibilizados 5 arquivos com as entradas e 5 arquivos com as saídas esperadas. Para o primeiro caso, as entradas são

```
4
0 0 0
0 1 10
0 2 15
0 3 20
1 0 10
1 1 0
1 2 35
1 3 25
2 0 15
2 1 35
2 2 0
2 3 30
3 0 20
3 1 25
```

```
3 2 30
3 3 0
```

e a saída esperada deve ser

```
0 1 3 2 0
80
```

Para o segundo caso, as entradas são

```
6
0 0 0
0 1 1
0 2 2
0 3 1
0 4 1
0 5 2
1 0 1
1 1 0
1 2 7
1 3 1
1 4 4
1 5 3
2 0 2
2 1 7
2 2 0
2 3 3
2 4 1
2 5 1
3 0 1
3 1 1
3 2 3
3 3 0
3 4 8
3 5 1
4 0 1
4 1 2
4 2 1
4 3 8
4 4 0
4 5 1
5 0 2
5 1 3
5 2 1
5 3 1
5 4 1
5 5 0
```

e a saída esperada deve ser

```
0 1 3 5 2 4 0
6
```

Para o terceiro caso, as entradas são

```
4
0 0 0
0 1 1
0 2 1
0 3 3
1 0 1
1 1 0
1 2 4
1 3 5
2 0 1
2 1 4
2 2 0
2 3 6
3 0 3
3 1 5
3 2 6
3 3 0
```

e a saída esperada deve ser

```
0 1 3 2 0
13
```

Para o quarto caso, as entradas são

```
5
0 0 0
0 1 2
0 2 0
0 3 3
0 4 6
1 0 2
1 1 0
1 2 4
1 3 3
1 4 0
2 0 0
2 1 4
2 2 0
2 3 7
```

```
2 4 3
3 0 3
3 1 3
3 2 7
3 3 0
3 4 3
4 0 6
4 1 0
4 2 3
4 3 3
4 4 0
```

e a saída esperada deve ser

```
0 1 2 4 3 0
15
```

Para o quinto caso, as entradas são

```
5
0 0 0
0 1 5
0 2 10
0 3 0
0 4 1
1 0 5
1 1 0
1 2 0
1 3 10
1 4 1
2 0 10
2 1 0
2 2 0
2 3 2
2 4 1
3 0 0
3 1 10
3 2 2
3 3 0
3 4 1
4 0 1
4 1 1
4 2 1
4 3 1
4 4 0
```

e a saída esperada deve ser

```
0 1 3 2 4 0
19
```

Felizmente o programa atendeu a todos os testes.

4 Análise

A análise dos resultados obtidos com o programa inclui a avaliação do tempo gasto para resolver o problema em relação ao tamanho do conjunto de cidades. Nesse programa, foi implementada uma abordagem de força bruta utilizando recursividade para encontrar o caminho mais curto, o que significa que ele explora todas as possíveis combinações de cidades para encontrar a solução. É importante notar que a eficiência computacional dessa abordagem pode ser fortemente afetada pelo aumento do número de cidades no conjunto. Conforme o número de cidades aumenta, o número de permutações possíveis cresce exponencialmente. Isso resulta em um alto custo computacional, o que significa que o tempo necessário para encontrar o caminho mais curto aumenta de forma exponencial com o número de cidades. Para conjuntos de cidades pequenos, a abordagem utilizando recursividade pode ser aceitável e fornecer soluções precisas. No entanto, para um grande número de cidades, a execução do programa pode levar um tempo impraticável, tornando-o ineficiente.
Para resolver o problema de forma eficiente em conjuntos de cidades maiores, seriam necessárias abordagens mais sofisticadas, como algoritmos baseados em heurísticas.
Em resumo, a análise dos resultados destaca a importância de considerar o tamanho do conjunto de cidades ao escolher a abordagem para resolver o problema do caixeiro-viajante. Para conjuntos pequenos, a recursividade pode ser adequada, mas para conjuntos maiores, outras abordagens mais eficientes são necessárias para encontrar soluções dentro de um tempo razoável.

5 Conclusão

É importante destacar que o desenvolvimento do programa para resolver o problema do caixeiro viajante foi um desafio significativo e uma experiência enriquecedora. Uma das maiores dificuldades enfrentadas ao longo desse projeto foi a criação da lógica para encontrar o melhor caminho. Buscar a solução ideal utilizando uma abordagem de força bruta exigiu raciocínio cuidadoso e o uso de técnicas de recursividade.
Um obstáculo adicional foi a falta de uma dupla para colaborar no projeto. Trabalhar sozinho significou que tive que lidar com os desafios sem a oportunidade de trocar ideias. No entanto, essa experiência me permitiu desenvolver minhas habilidades de resolução independente de problemas, o que considero valioso.
Apesar dos desafios e das limitações da abordagem de força bruta, considero o trabalho produtivo, resultando em um programa funcional capaz de encontrar o caminho mais curto entre cidades e fornecer uma saída formatada de acordo com os padrões desejados. O desenvolvimento deste projeto fortaleceu minhas habilidades de programação e proporcionou uma introdução prática à resolução de problemas de otimização e à implementação de algoritmos em cenários do mundo real. Estou satisfeito com o resultado e com a superação dos desafios apresentados por este projeto desafiador.

6 Bibliografia

BHARGAVA, Aditya. Y. Entendendo Algoritmos: Um guia ilustrado para programadores e outros curiosos. Novatec, 2017.