Implementação Algoritmo de Kruskal

Avaliação 2 – Teoria dos Grafos

Camila Andrade de Sena

Centro de Ciências e Tecnologia
Universidade Estadual do Ceará (UECE) – Fortaleza, CE – Brazil

andrade.sena@aluno.uece.br

Abstract. *This report aims to describe the decisions and code structures revolving around the implementation of the Kruskal Algorithm in a given set of four graphs. In this regard, it dealt with the description of each structure used, as well as the functions and why they were used at any given time. Therefore, an overview of the developed programming project will be presented.*

Resumo. *O presente relatório tem como objetivo descrever as decisões e estruturas de códigos revolvendo em torno da implementação do Algoritmo de Kruskal em um conjunto pré-definido de quatro grafos. Neste sentido, tratou-se da descrição de cada estrutura utilizada, assim como as funções e o porquê do uso das mesmas em cada dado momento. Portanto, uma visão panorâmica do projeto de programação desenvolvido será apresentada.*

1. Descrição do ambiente de desenvolvimento

O projeto foi desenvolvido no ambiente com a descrição a seguir.

a) Processador: Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz

b) Quantidade de memória: 8,00 GB

c) Sistema operacional: Windows 11 64-bit

d) Linguagem de programação: C / C++

e) Compilador (versão e IDE): Code::Blocks - Versão 20.03-r11983

2. Decisões e Estruturas

Inicialmente, foi criado um projeto de aplicação de console com a linguagem C++, contendo o arquivo principal main. O código foi inicialmente pensado em C, no entanto, devido a possibilidade de orientação a objetos e a facilidade para se lidar com vetores (estrutura usada para armazenar as informações do grafo) do C++, eventualmente as duas linguagens foram utilizadas.

Ademais, como o compilador foi o Code::Blocks, no começo da implementação houve a formação de um arquivo ".cbp", que quando aberto contém todos os arquivos pertencentes ao projeto.

Logo no começo foram adicionadas as seguintes bibliotecas:

```
#include <iostream>
#include <string.h>
#include <fstream>
#include <vector>
#include <algorithm>
```

A primeira é a biblioteca padrão de entrada e saída, a segunda e terceira serão úteis na futura manipulação de arquivos. A quarta, como dito anteriormente, será utilizada na manipulação dos abundantemente presentes vetores. E a última contém o sort, que ordenará as arestas pelo menor peso.

Foram utilizados conceitos de Orientação a Objetos para a representação dos grafos e arestas.

2.1. Classes, Construtores e Funções

Antes da implementação do algoritmo de Kruskal em si, se fazia necessária uma decisão de como os grafos e informações seriam dispostos no código. Foi percebido que seria adequado utilizar Orientação a Objetos, que permite a representação de elementos que estão presentes no mundo real de maneira organizada.

A primeira informação a ser abstraída em uma classe foi a aresta, com seus atributos relevantes: dois inteiros para os vértices origem e destino; um double para o peso.

O peso foi incluido pois era de conhecimento prévio que o grafo era ponderado. Nenhuma inclusão adicional foi necessária, uma vez que apenas grafos simples e não direcionados foram tradados.

```
// Definição da classe aresta
class aresta{
    int vertice;
    int vertice2;
    double peso;

public: // Modificador de acesso
    aresta(int vertice,int vertice2, double peso){ // Construtor
        this->vertice=vertice;
        this->vertice2=vertice2;
        this->peso=peso;
    }


    int retornar_vertice(){
        return vertice;
    }

    int retornar_vertice2(){
        return vertice2;
    }

    double retornar_peso(){
        return peso;
    }
```

O acesso foi modificado para público e o método construtor foi definido, de modo a garantir que todos os atributos da aresta fossem inicializados corretamente. Também foram escritas as funções para obter os valores de cada atributo, como visto acima.

A última das funções referentes as arestas diz respeito ao retorno da aresta com menor peso, que será usada futuramente no algoritmo.

```
// Retorna menor peso quando o operador < for chamado
    bool operator < (const aresta& aresta2) const{
        return (peso<aresta2.peso);
    }

};
```

O operador "<" faz a operação de comparação entre as duas arestas passadas como parâmetro e ordena os pesos de maneira crescente, sem necessidade para futuras comparações.

É chegado ao final da definição das arestas, e se tem início a definição da classe grafo.

Assim como anteriormente, foi criada uma classe grafo com os atributos pertencentes a ele: um inteiro para o número de vértices e um vetor de arestas do tipo já definido "aresta". O acesso foi modificado para *public* e o construtor foi escrito, fazendo com que o grafo ao ser criado receba seu número de vértices como parâmetro.

```
// Definição classe grafo
class grafo{
    int numero_vertices;
    vector<aresta> arestas; // Vetor de arestas

public: // Modificador de acesso
    grafo(int numero_vertices){ // Construtor
        this->numero_vertices=numero_vertices;
    }
```

A penúltima função escrita antes do Algoritmo de Kruskal foi a de adicionar arestas, que recebe os vértices origem e destino como parâmetros, assim como o peso referente a elas. Dentro da função, é criada a aresta com os atributos inseridos pelo usuário, que é então adicionada ao vetor de arestas.

```
// Cria a aresta e insere no vetor de arestas
    void adicionar_aresta(int vertice, int vertice2, double peso){
        aresta Aresta(vertice, vertice2, peso);
        arestas.push_back(Aresta);
    }
```

A última função já faz parte do algoritmo, dizendo respeito a busca e união de elementos do grafo. Basicamente, define a qual subconjunto um elemento pertencerá. Para aprofundar o que será feito, vale adiantar algo do que é feito já dentro da função de Kruskal.

Suponha um grafo de quatro vértices. É visto que haverá, inicialmente, a criação de um vetor com o tamanho igual ao número de vértices. Esse vetor será todo inicializado com o valor de -1.

1	2	3	4
-1	-1	-1	-1

O próximo passo é percorrer o vetor de arestas já citado no relatório, recuperando cada aresta. Imagine que em uma primeira iteração das funções “retornar_vertice()“ “retornar_vertice2()“ e recuperamos a aresta (1,2).

```
vertice=1;
vertice2=2;
```

Agora, será necessário o retorno da busca realizada pela última função supracitada, a de achar os subconjuntos.

```
// Acha o subconjunto do elemento fornecido
// VÊ se dois elementos estão no mesmo subconjunto
int achar_subconjunto(int subconjunto[], int i){
    if(subconjunto[i]==-1)
        return i;
    return achar_subconjunto(subconjunto, subconjunto[i]);
}
```

Nesse cenário, a função receberia como parâmetros o vetor inicializado com –1 e o valor de cada um dos vértices retornados, 1 e 2. Como visto, se na posição do vetor referente ao vértice o valor contido for -1, o valor retornado será o do vértice. Caso contrário, a função será chamada recursivamente recebendo novamente o vetor de -1 e o valor dele no índice do vértice.

Na primeira iteração, todos os valores do vetor são -1. Logo, nesse primeiro momento os valores seriam 1 e 2.

Tendo a função sido explicada, faz-se necessário a explicação do Algoritmo de Kruskal para a compreensão do que será feito com os valores obtidos.

3. Algoritmo de Kruskal

O cenário criado deve ser deixado de lado momentaneamente.

No Algoritmo de Kruskal, uma floresta é construída ao longo da execução, as árvores são encontradas e unidas ao final do processo. Na construção desse trabalho foi utilizado o Union&Find para achar as arestas, que no pior caso otimiza a execução para O(m log n), sendo “m” o total de arestas e “n” o total de vértices. A parte do “Find” foi explicada logo acima, por meio da função de busca.

Visto isso, uma explicação será realizada unindo a idéia geral do algoritmo e o código em si.

Primeiro, cada vértice do grafo é tornado em uma árvore independente. Ou seja, são formados subconjuntos. O subconjunto foi representado como um vetor de tamanho igual ao número de vértices mais uma unidade. A memória foi alocada com a função “malloc”. A função “memset()” foi utilizada para preencher todas as posições alocadas com “-1”.

O resto do processo envolve encontrar a aresta de menor peso e incorporar ela a Árvore Geradora Mínima, que também foi representada por um vetor. O algoritmo “sort” da biblioteca algorithm foi utilizado recebendo como parâmetro o vetor de arestas, de forma a ordená-lo. E claro, foi criado um vetor do tipo “aresta” para a Árvore Geradora Mínima.

```
    void algoritmo_kruskal(){
        // Cria memória para as árvores independentes
        int * subconjunto = (int*)malloc((numero_vertices+1)
        * sizeof(int));

        // Inicializa todas as árvores como -1
        memset(subconjunto, -1, sizeof(int)*(numero_vertices+1));

        // Percorre o vetor de arestas e ordena
        sort(arestas.begin(), arestas.end());

        // Vetor que vai receber a árvore geradora mínima
        vector<aresta> arvore_geradora_minima;

        // Percorre as arestas
        int i=0;
        while(i!=arestas.size()){

        int vertice=achar_subconjunto
        (subconjunto,arestas[i].retornar_vertice());
        int vertice2=achar_subconjunto
        (subconjunto,arestas[i].retornar_vertice2());

        // Checa se não tem ciclo, ou seja, os dois vértices
        precisam ser diferentes
        if(vertice!=vertice2){
            // Insere os vértices na árvore geradora mínima
            arvore_geradora_minima.push_back(arestas[i]);
            // Une os vértices
            int conjunto_vertice=achar_subconjunto
            (subconjunto,vertice);
            int conjunto_vertice2=achar_subconjunto
            (subconjunto,vertice2);
            subconjunto[conjunto_vertice]=conjunto_vertice2;
            }
            i++;
        }

        // Imprime a árvore gerada
        int j=0;
        // double peso=0;
        printf("%d \n",numero_vertices);
        while(j!=arvore_geradora_minima.size()){
            // peso=peso+arvore_geradora_minima[j].retornar_peso();
            printf("%d %d %.2f \n",
            arvore_geradora_minima[j].retornar_vertice(),
            arvore_geradora_minima[j].retornar_vertice2(),
            arvore_geradora_minima[j].retornar_peso());
            j++;
        }
       // printf("PESO: %f \n",peso);
    }
};
```

Haverá reiteração até que todos os vértices do grafo façam parte da mesma árvore, e não pode haver formação de ciclos.

Retornando ao cenário criado. Uma vez que os vértices 1 e 2 foram recuperados, é preciso testar se eles são diferentes. Em um ciclo, o vértice de origem e destino são diferentes, então uma comparação “if” é realizada. Se eles são diferentes, então pertencem a Árvore Geradora Mínima. O “push_back” é chamado para realizar a inserção no vetor correspondente da Árvore. Então a aresta (1,2) está dentro.

Até o presente momento apenas o “Find” do “Union&Find” foi esclarecido.

Reafirmando, a função de busca não é capaz apenas de buscar e devolver os elementos, mas também de achar o subconjunto de um determinado elemento. Portanto, no “Union”, ela será chamada novamente, devolvendo para duas variáveis criadas os subconjuntos dos vértices passados como parâmetro juntamente com o vetor subconjunto. No cenário apresentado, quando houver o retorno, o 2 será incluído na posição 1 do vetor subconjunto.

1	2	3	4
2	-1	-1	-1

Haverá repetição da operação descrita e, portanto, o preenchimento do vetor supracitado enquanto o inteiro “i”, que foi inicializado como 0, seja diferente do tamanho do vetor de arestas.

Quando todas as iterações forem feitas, será impressa a Árvore Geradora Mínima, contendo seu número de vértices na primeira linha, e nas linhas seguintes todas as arestas com as informações correspondentes. Novamente, uma estrutura “while” será utilizada, com um inteiro variando de 0 até o tamanho do vetor da Árvore.

As linhas de código que foram escritas para apresentar a soma dos pesos das arestas estão comentadas no código fonte. A cada iteração do loop anterior, o peso é somado a uma variável peso.

4. O Programa Principal

Logo nas primeiras linhas do *main* é chamada a função “freopen()”, que recebe como parâmetros o nome do arquivo de saída, o modo de abertura e o fluxo padrão. Ela funciona direcionando o fluxo para o arquivo especificado, e foi usada para produzir os quatro “.txt” contendo a saída do console. Para a entrega final, foi feita a escolha de comentar essas linhas, pois elas já haviam servido seu propósito.

```
int main()
{
    /* escrever o output do console em um arquivo
    freopen("out1.txt","w",stdout); *
```

Ocorre a declaração das variáveis já conhecidas, dois inteiros para os vértices e um double para os pesos. Ademais, há a instância de um inteiro para o total de vértices que será passado como parâmetro para a função de criação do grafo, assim como uma string para receber o número de vértices da primeira linha do “.txt” que será aberto com a função seguinte.

O “ifstream()” serve para ler ficheiros, e deve adotar entre os parênteses o nome do arquivo contendo o grafo a ser utilizado. O grafo “grafo_W_1_1.txt” está como exemplo.

```
    // Declaração de variáveis
    int total_vertices=0,v1,v2;
    double peso;
    string total_verticesc;

    // Abre o arquivo fornecido como parâmetro
    ifstream file("grafo_W_1_1.txt");
```

A função “getline()” é então utilizada para ler a primeira linha do ficheiro aberto, passar para a string “total_verticesc”, e atualizar o ponteiro para a próxima linha. O “stoi” transforma o conteúdo da string em um inteiro, que é passado como o número de vértices para a criação do grafo “g”.

```
    // Captura o total de vértices na primeira linha e cria o grafo
    getline(file,total_verticesc);
    total_vertices=stoi(total_verticesc);
    grafo g(total_vertices);
```

Para transformar o restante das linhas em arestas, foi chamado um *loop* que roda até o final do arquivo, passando o conteúdo das linhas de forma individual, e separado por colunas, para as variáveis “v1”, “v2”, e “peso”. O processo é interrompido se houver um self- loop, comportamento que não foi observado. Se não houver interrupções, os números são informados para a função de adicionar arestas no grafo já instanciado.

```
     // Se não houver self loop, adiciona as arestas ao grafo criado
     while(file >> v1 >> v2 >> peso){
        if(v1==v2){
            printf("Há um self-loop nesse grafo! \n");
            return 0;
        }
        else{
            g.adicionar_aresta(v1,v2,peso);
        }
    }
```

Desse modo, resta apenas chamar a função do Algoritmo de Kruskal para o grafo, fechar o arquivo e obter os resultados.

```
    // Chama o algoritmo de kruskal
    g.algoritmo_kruskal();

    // Fecha o arquivo
    file.close();
    return 0;
}
```

5. Resultados

Como solicitado, foram gerados quatro arquivos no formato “.txt”, que se encontram junto com a entrega do presente relatório:

- resultado_grafo_W_1_1;
- resultado_grafo_W_2_1;
- resultado_grafo_W_3_1;
- resultado_grafo_W_4_0.

Para cada um dos grafos foram logrados os seguintes resultados:

- **grafo_W_1_1**

  Peso da Árvore Geradora Mínima: 220.110000

  Tempo de Execução: 0.901 s

- **grafo_W_2_1**

  Peso da Árvore Geradora Mínima: 2247.070000

  Tempo de Execução: 13.157 s

- **grafo_W_3_1**

  Peso da Árvore Geradora Mínima: 22218.710000

  Tempo de Execução: 938 s

- **grafo_W_4_0**

  Peso da Árvore Geradora Mínima: -43990.450000

  Tempo de Execução: 20.898 s

Referências

Rocha, L. (2022) “Grafos para a Computação: Fundamentos, Algoritmos e Aplicações”, Edited by Leonardo Sampaio Rocha, Fortaleza.

Backes, A. (2016) “Estrutura de dados descomplicada: em linguagem C”, Edited by Elsevier, Rio de Janeiro.