Relatório do Primeiro Trabalho Prático - T1

Alunos: Karla Sancio e Hugo Lima
Matrículas: 2019108953, 2019107917
Disciplina: Técnicas de Busca e Ordenação
Curso: Engenharia da Computação

1- Introdução

Este trabalho prático tem como objetivo elaborar um algoritmo que resolva o *Travelling Salesman Problem* com um tempo de execução aprimorado, e que essa solução do problema seja ótima para todos os possíveis casos. A ideia do TSP é fazer com que o caixeiro viajante consiga realizar uma *tour* com o menor custo possível, visitando todas as cidades e depois retornando ao início.

Para resolver o problema foi necessário implementar uma *Minimum Spanning Tree* (MST) que consiste em um subconjunto de arestas de um grafo, formando uma árvore de vértices.

Neste documento, explicamos como se procedeu a implementação do trabalho, apresentando cada ferramenta utilizada e justificando a sua escolha. Também detalhamos cada resultado obtido durante a execução do programa, envolvendo análises de complexidade e de tempo de execução.

2 - Metodologia

A nossa implementação se inicia com a criação da estrutura de dados responsável por armazenar as informações do arquivo de entrada, para logo em seguida ler o arquivo e alocar cada informação em seu devido lugar. Cada linha de informação da entrada foi lida e armazenada em uma variável, utilizando a função ***fgets()*** da biblioteca stdio.h, com exceção da linha de comentários que pôde ser ignorada. Para a leitura e representação das coordenadas aos vértices, implementamos a estrutura ***Node**** presente nos arquivos Node.h/Node.c. Armazenamos essas informações na estrutura e criamos um array de nodes para armazenar o ID de cada node.

Com esse setup inicial concluído, agora é criado uma matriz de distâncias para armazenar as distâncias entre cada nó. Essa matriz utiliza a função ***calculaDist()***, presente também no arquivo Node.c. O cálculo de complexidade dos acessos da matriz é equivalente a **N(N-1)**.

```
double** matrizDist = (double**)malloc(numVertices * sizeof(double*));

for(int i = 0; i < numVertices; i++){
    matrizDist[i] = (double*)malloc(numVertices*sizeof(double));
    for(int j = 0; j < i; j++){
        matrizDist[i][j] = calculaDist(i,j,arrayNode,numVertices);
    }
}
```

A próxima etapa é a construção dos arcos da MST. Para tal, foram criados os arquivos ***Arco.c/Arco.h***. A estrutura arco é a seguinte:

```
typedef struct arco{
    double peso;
    Node* leftNode;
    Node* rightNode;
}Arco;
```

Escolhemos essa estrutura para que fosse possível localizar os nós da esquerda e da direita do arco e também atribuir o seu devido peso, pois mais à frente será necessário ordená-los de acordo com seu peso. Para ordenar os arcos, optamos pelo método de ordenação *quicksort mediana de três*. Para justificar essa escolha precisamos pontuar que são muitos arcos formados. Inicialmente optamos pelo quicksort normal, mas notamos que existiam casos com muitas repetições. Dessa forma, ao implementar o segundo algoritmo vimos uma melhora significativa no tempo de execução do programa, o que é justificado quando analisamos que para o caso médio e pior caso, o número de comparações é o mesmo, mas para o melhor varia de **N logN** (quicksort normal) para apenas **N** (3-way quicksort).

Com os arcos já ordenados, está tudo pronto para a criação da MST em si. Nesse momento, aplicamos o algoritmo de Kruskal disponível em: https://en.wikipedia.org/wiki/Kruskal%27s_algorithm, para encontrar a árvore geradora mínima a partir de um grafo completo. O algoritmo funciona da seguinte maneira:

- Cria um conjunto de árvores, onde cada vértice do grafo é uma árvore diferente e cria um outro conjunto com todas as arestas do grafo .
- Se o conjunto de arestas é não-nulo e a árvore ainda não está expandindo, então ele remove uma aresta com peso mínimo do conjunto de arestas. Se a aresta removida conecta outras duas árvores diferentes, então ela é adicionada ao conjunto de árvores.
- O esperado do algoritmo, se implementado corretamente, é ter todos os arcos do grafo conectados, formando uma árvore geradora mínima.

A implementação em C então fica:

```
int* mst = MST_init(numVertices);
Node* nA1;
Node* nA2;
Arco** arcosMST = (Arco**)malloc((numVertices-1)*sizeof(Arco*)); // Array de arcos utilizados na MST
int posAtual = 0;

for(int i = 0 ; i< nArcos;i++){
    nA1 = grafo[i]->leftNode;
    nA2 = grafo[i]->rightNode;


    if(MST_find(nA1->id-1,mst) != MST_find(nA2->id-1,mst)){
        MST_union(nA1->id-1,nA2->id-1,mst);
        arcosMST[posAtual] = grafo[i];
        posAtual++;
    }
}
```

Para um grafo com ***E*** arestas (arcos do grafo) e ***V*** vértices, o cálculo de complexidade do Algoritmo de Kruskal é dado por *O(E log V)* para tempo de execução. Pela especificação do trabalho, e assumindo G como um grafo completo dado por $|E| = N(N-1)/2$, ao passarmos a complexidade do algoritmo para função de *N* ela resulta em $\boldsymbol{O(N^2 \log N)}$. Todas as funções referentes a esse processo se encontram nos arquivos **MST.c/MST.h**.

Tentamos implementar uma estrutura de dados Árvore para gerar o *tour* da MST, mas como não obtivemos sucesso, optamos então por ir criando cada nó da árvore e já imprimir o vértice em que a função percorreu. Basicamente, visitamos um nó, verificamos se ele é nulo, caso seja, será criado um nó com o id (valor) do

vértice encontrado. Para encontrar o vértice utilizamos a função **searchNext**, que no qual verifica no array de arcos da MST se há algum arco em que o id do último vértice visitado está presente, retornando o id do outro vértice pertencente ao arco e setando esse arco para *NULL*, a fim de não repetir o mesmo caminho.

```
int searchNext(int idNode,Arco** array,int size){
    for(int i = 0;i < size; i++){ //  Roda no array de arcos da MST
        if(array[i] != NULL){ // verifica se nulo
            if(array[i]->leftNode->id == idNode){ // se o idNode faz parte do arco
                idNode = array[i]->rightNode->id;
                array[i] = NULL; // seta NULL para não repetirmos o mesmo caminho
                return idNode; // retorna o idNode do outro vértice do arco
            }
            if(array[i]->rightNode->id == idNode){  // se o idNode faz parte do arco
                idNode = array[i]->leftNode->id;
                array[i] = NULL;
                return idNode; // retorna o idNode do outro vértice do arco
            }
        }

    }
    return 0; // caso nao encontre mais arcos com esse id retorna 0
}
```

3 - Tabela comparativa dos Cálculos de Complexidade

Parte do Código	**Cálculo de Complexidade**
Criação da Matriz de Distâncias	N(N-1)
Quicksort	N log N
3-way Quicksort	N
Algoritmo de Kruskal	O(N² log N)

4- Análise Empírica

Para a construção da tabela a seguir a fim de obter resultados mais coesos, escolhemos o caso **pr1002.tsp** para a montagem da tabela.

Medida	Leitura dos Dados	Cálculo das Distâncias	Ordenação das Distâncias	Obtenção da MST	Obtenção do Tour	Escrita da saída	Total
Tempo (segundos)	**0.0000s**	**0.031s**	**0.109s**	**0.078s**	**0.000s**	**0.0000s**	**0.2344s**
Porcentagem	**0%**	**13.22%**	**46.50%**	**33.27%**	**0%**	**0%**	**100%**