CEFET/RJ

Árvore Geradora Mínima: Itinerário Papai Noel - 1764

Felipe Hiroshi Correia Katsumoto
CEFET/RJ - Engenharia de Computação
AEDS II

RESUMO

Este documento apresenta a soluçao proposta (e aceita pelo juíz online Beecrowd) para o problema de Árvores Geradoras Mínimas (AGM) denominado como Itinerário Papai Noel - 1764.
A solução consiste na aplicação direta do algoritmo de Kruskal com pequenas alterações para adequações ao problema, juntamente com a utilização de estruturas de dados auxiliares, como o Union Find e um algoritmo de ordenação com a complexidade *O(n log n)*, o MergeSort foi escolhido neste caso.

PALAVRAS CHAVE. Kruskal, Árvore Geradora Mínima, Itinerário Papai Noel, Grafos, Estruturas de dados.

ABSTRACT

This document presents the proposed solution (and accepted by the online judge Beecrowd) for the Minimum Spanning Tree (MST) known as 1764 - Itinerary of Santa Claus.
The solution consists on the direct application of the Kruskal algorithm with some small changes to adapt to the problem, along with some auxiliary data structures such as Union Find and a sort algorithm with the complexity *O(n log n)*, in this case, MergeSort was chosen.

KEYWORDS. Kruskal, Minimum Spanning Tree, Santa Claus Itinerary, Graphs, Data structures.

CEFET/RJ

1. O problema

O problema é descrito da seguinte maneira:

Antes de Papai Noel começar a fazer as suas viagens de trenó pelo Brasil para entregar os presentes de Natal, ele solicitou que você o ajudasse a desenhar um mapa com todas as cidades que deverá visitar. A regra para desenhar este mapa é a seguinte: a soma de todas rotas (distâncias entre duas cidades) existentes no mapa deve ser a menor possível e deve-se poder chegar em qualquer cidade, independente de onde se esteja partindo. Noel não se importa de passar por uma determinada cidade mais de uma vez, contanto que ele utilize apenas as rotas desenhadas no mapa.

Isto quer dizer que será fornecido um grafo ponderado e não direcionado, onde os vértices serão as cidades pela qual queremos percorrer, as arestas os caminhos entre as cidades e os pesos a distância entre uma cidade e outra.

O enunciado do problema exige que deve-se passar por todos os vértices somando o menor custo possível, uma Árvore Geradora Mínima (AGM) em sua definição.

Além disto é apresentada no enunciado a limitação de timelimit 1.

2. Algoritmo proposto - Kruskal

O algoritmo proposto para solucionar este problema é o Algoritmo de Kruskal, um algoritmo para achar árvores geradoras mínimas em grafos conexos.

Pseudocódigo

```
1:  AGM - Kruskal (G, w)
2:  A ← ∅
3:  for all v ∈ V do
4:      MAKESET(v)
5:  end for
6:  Ordene as arestas F em ordem não-decrescente de w
7:  for all (u, v) ∈ F nessa ordem do
8:      if FINDSET(u) ≠ FINDSET(v) then
9:          A ← A ∪ {(u, v)}
10:         UNION(u, v)
11:     end if
12: end for
13: return A
```

O algoritmo segue os seguintes passos:

- O conjunto de **Arestas A**, é inicialmente vazio;
- As arestas são ordenadas a partir do peso em forma crescente;
- No algoritmo, o conjunto de **Vértices V** forma uma floresta **F** (inicialmente de grafos unitários);
- A cada iteração o algoritmo escolhe uma aresta de menor peso entre vértices distintos e que não forma ciclo, adiciona esta aresta a **A** e une os grafos em F.
- Repete-se até formar uma única árvore em F.

Por utilizar sempre as arestas de menor peso e que não formem ciclos, no final do algoritmo a árvore formada em F será uma AGM.

3. Resolução

Visando o desempenho e eficiência foram utilizados algoritmos de ordenação e estruturas de dados.

3.1. MergeSort

O algoritmo de MergeSort foi escolhido por ter complexidade de O(n log n) em todos os seus casos.
O algoritmo funciona da seguinte forma:
A ordenação ocorre com a técnica de dividir para conquistar. Os itens vão sendo divididos até que torna-se trivial a comparação e volta concatenando em ordem crescente, organizando o vetor.

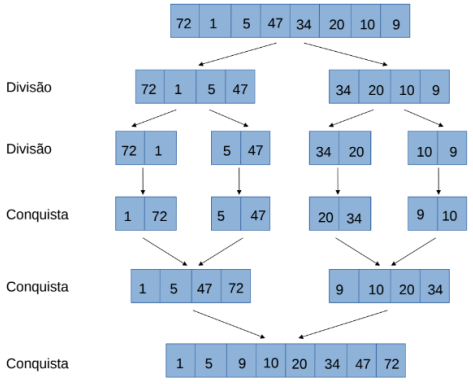

Figura 1: MergeSort

3.2. Union Find

Nota-se que o algoritmo de Kruskal utiliza-se da estrutura de Disjoint-set para lidar com os vértices desconexos com a árvore principal em F, portanto torna-se necessária a implementação da estrutura de dados ou a utilização de alguma biblioteca que a implemente, como é utilizado apenas o MakeSet, o Union e o Find, optei pela implementação própria. A implementação utiliza duas técnicas notáveis: Union by Rank e a técnica de path compression.

Union by Rank

Para minimizar a altura das árvores formadas durante a operação de Union, é utilizado o rank de cada árvore para decidir qual árvore é anexada em qual, a menor é anexada a maior. Isto garante que a árvore mantenha-se mais ou menos balanceada sem grandes descrepâncias de altura.

Path Compression

A compressão do caminho é a técnica de anexar todos os nós filhos a raiz da árvore, achatando a altura da árvore e tornando muito rápido o acesso aos nós. O path compression é aplicado toda vez que o find é chamado.

3.3. Mudanças em Kruskal

Os parâmetros da função original são passados apenas os pesos e o grafo. Na função utilizada são passados:

- **n - Número de vértices no grafo**: Lido da entrada, determina a condição de parada da função pois sabe-se que a AGM deverá ter n-1 arestas.
- **m - Número de arestas no grafo**: Lido da entrada, controla o número de iterações do for.
- **A - Arestas do grafo**: Lista de arestas do grafo que contém os vértices que a aresta liga e o peso da aresta. Usada para ordenar as arestas e para as operações do Union Find.

Além dos parâmetros utilizados foram adicionadas duas variáveis:

- **arestasInc**: conta quantas arestas foram adicionadas e é usada para controlar a parada da função assim que atinge n-1.
- **somaArestas**: na literatura o algoritmo de Kruskal retorna o conjunto de arestas, mas como o enunciado do problema pede para imprimir apenas a soma das arestas da AGM, Kruskal retorna o inteiro que é a soma das arestas que formam a AGM.

Algortimo final

```
#include <stdio.h>
#include <stdlib.h>

typedef struct vert {
    int dado;
    int rank;
    struct vert* pai;
} Vert;

typedef struct aresta {
    int peso;
    Vert* u;
    Vert* v;
} Aresta;

void merge(Aresta* A, int inicio, int meio, int fim) {
    int n1 = meio - inicio + 1;
    int n2 = fim - meio;
    Aresta* L = (Aresta*)malloc(n1 * sizeof(Aresta));
    Aresta* R = (Aresta*)malloc(n2 * sizeof(Aresta));
    for (int i = 0; i < n1; i++) L[i] = A[inicio + i];
    for (int j = 0; j < n2; j++) R[j] = A[meio + 1 + j];
    int i = 0, j = 0, k = inicio;
    while (i < n1 && j < n2) {
        if (L[i].peso <= R[j].peso)
            A[k++] = L[i++];
        else
            A[k++] = R[j++];
    }
    while (i < n1) A[k++] = L[i++];
```

```
    while (j < n2) A[k++] = R[j++];
    free(L);
    free(R);
}

void mergeSort(Aresta* A, int inicio, int fim) {
    if (inicio >= fim) return;
    int meio = (inicio + fim) / 2;
    mergeSort(A, inicio, meio);
    mergeSort(A, meio + 1, fim);
    merge(A, inicio, meio, fim);
}

Vert* criaVert(int dado) {
    Vert* novo = (Vert*)malloc(sizeof(Vert));
    novo->dado = dado;
    novo->pai = novo;
    novo->rank = 0;
    return novo;
}

Vert* find(Vert* i) {
    if (i != i->pai) {
        i->pai = find(i->pai);
    }
    return i->pai;
}

void unionSets(Vert* x, Vert* y) {
    Vert* sx = find(x);
    Vert* sy = find(y);
    if (sx->rank > sy->rank) {
        sy->pai = sx;
    } else {
        sx->pai = sy;
        if (sx->rank == sy->rank) {
            sy->rank++;
        }
    }
}

int kruskal(int n, int m, Aresta* A) {
    mergeSort(A, 0, m - 1);
    int somaArestas = 0, arestasInc = 0;
    for (int i = 0; i < m; i++) {
        if (find(A[i].u) != find(A[i].v)) {
```

```
            somaArestas += A[i].peso;
            unionSets(A[i].u, A[i].v);
            arestasInc++;
            if (arestasInc == n - 1) break;
        }
    }
    return somaArestas;
}

int main() {
    int m, n;
    while (1) {
        scanf("%d %d", &m, &n);
        if (m == 0 && n == 0) break;
        Aresta* A = (Aresta*)malloc(n * sizeof(Aresta));

        Vert** vertices = (Vert**)malloc(m * sizeof(Vert*));
        for (int i = 0; i < m; i++) {
            vertices[i] = criaVert(i);
        }

        for (int i = 0; i < n; i++) {
            int u, v, peso;
            scanf("%d %d %d", &u, &v, &peso);
            A[i].u = vertices[u];
            A[i].v = vertices[v];
            A[i].peso = peso;
        }

        printf("%d\n", kruskal(m, n, A));

        free(A);
        free(vertices);
    }
    return 0;
}
```

4. Beecrowd

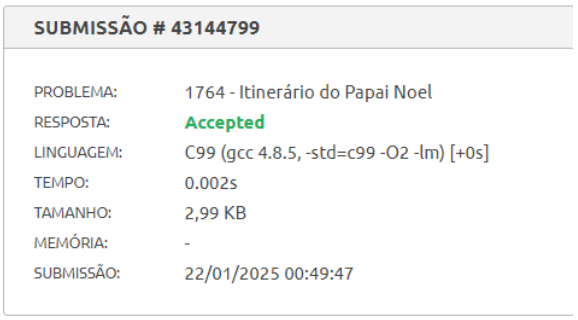

Figura 2: Problema aceito no Beecrowd e especificações de consumo

5. Referências

ASSIS, Laura. Aula29_AEDsII_ArvoreGeradoraMinAlgoritmos. 2-24. [Apresentação de slides]. CEFET/RJ UnED Petrópolis, 2024.