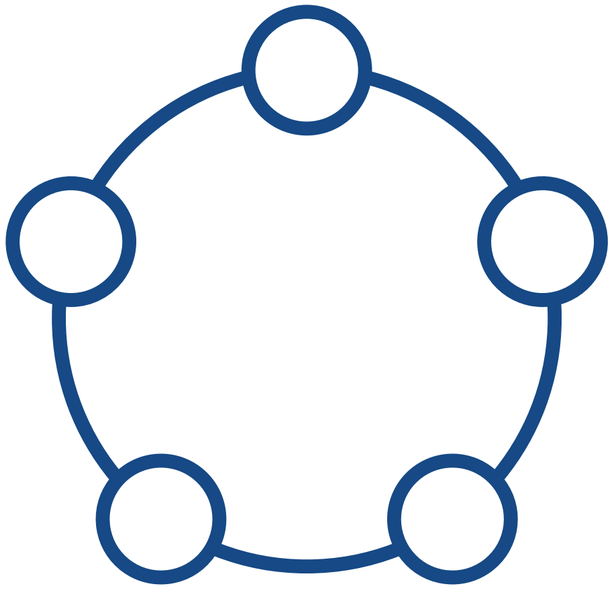


Semana 5

Teoria de Grafos

Ana Luiza Sticca, Gabriel Antônio, Guilherme Cabral e João Pedro Marques

Problemas

04 Re-Connecting Computer Sites

10 Driving Range

04 Re-Connecting Computer Sites

Re-Connecting Computer Sites

Considere o problema de selecionar um conjunto T de linhas de alta velocidade para conectar N locais de computadores, a partir de um universo de M linhas de alta velocidade, cada uma conectando um par de sites de computadores. Cada linha de alta velocidade tem um dado custo mensal, e o objetivo é minimizar o custo total de conexão dos N sites de computadores, onde o custo total é a soma do custo de cada linha incluída no conjunto T. Considere ainda que este problema foi resolvido anteriormente para o conjunto de N sites de computadores e M linhas de alta velocidade, mas alguns K novas linhas de alta velocidade foram recentemente disponibilizadas. Seu objetivo é calcular o novo conjunto T' que pode render um custo inferior ao conjunto original T, devido às K novas linhas de alta velocidade adicionais e quando as linhas de alta velocidade M + K estiverem disponíveis.

Re-Connecting Computer Sites

Entrada A entrada conterá vários casos de teste, cada um deles conforme descrito abaixo. Consecutivo os casos de teste são separados por uma única linha em branco. A entrada é organizada da seguinte forma:

- Uma linha contendo o número N de sites de computador, com $1 \leq N \leq 1000000$, e onde cada site de computador é referido por um número i, $1 \leq i \leq N$
- O conjunto T de linhas de alta velocidade previamente escolhidas, consistindo de N – 1 linhas, cada uma descrevendo uma linha de alta velocidade e contendo os números dos dois locais de computador que a linha conecta e o custo mensal de utilização desta linha. Todos os custos são inteiros.
- K linhas, cada uma descrevendo uma nova linha de alta velocidade e contendo os números dos dois computadores sites aos quais a linha se conecta e o custo mensal de utilização desta linha. Todos os custos são inteiros.

Re-Connecting Computer Sites

- Uma linha contendo o número M de linhas de alta velocidade originalmente disponíveis, com $N - 1 \leq M \leq N(N - 1)/2$.
- M linhas, cada uma descrevendo uma das linhas de alta velocidade originalmente disponíveis e contendo os números dos dois locais de computadores que a linha conecta e o custo mensal de uso dessa linha. Todos os custos são inteiros.

Saida: Para cada caso de teste, a saída deve seguir a descrição abaixo. As saídas de dois casos consecutivos serão separados por uma linha em branco. O arquivo de saída deve ter uma linha contendo o custo original de conexão dos N sites de computadores com M linhas de alta velocidade e outra linha contendo o novo custo de conexão dos N locais de informática com linhas de alta velocidade M + K. Se o novo custo for igual ao custo original, o mesmo valor será escrito duas vezes.

Sample Input

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

Sample Output

```
20
17
```

Re-Connecting Computer Sites

Estrutura de dados: Listas de Adjacência

Vértices: Locais de computadores

Arestas: Linhas de alta velocidade

Re-Connecting Computer Sites

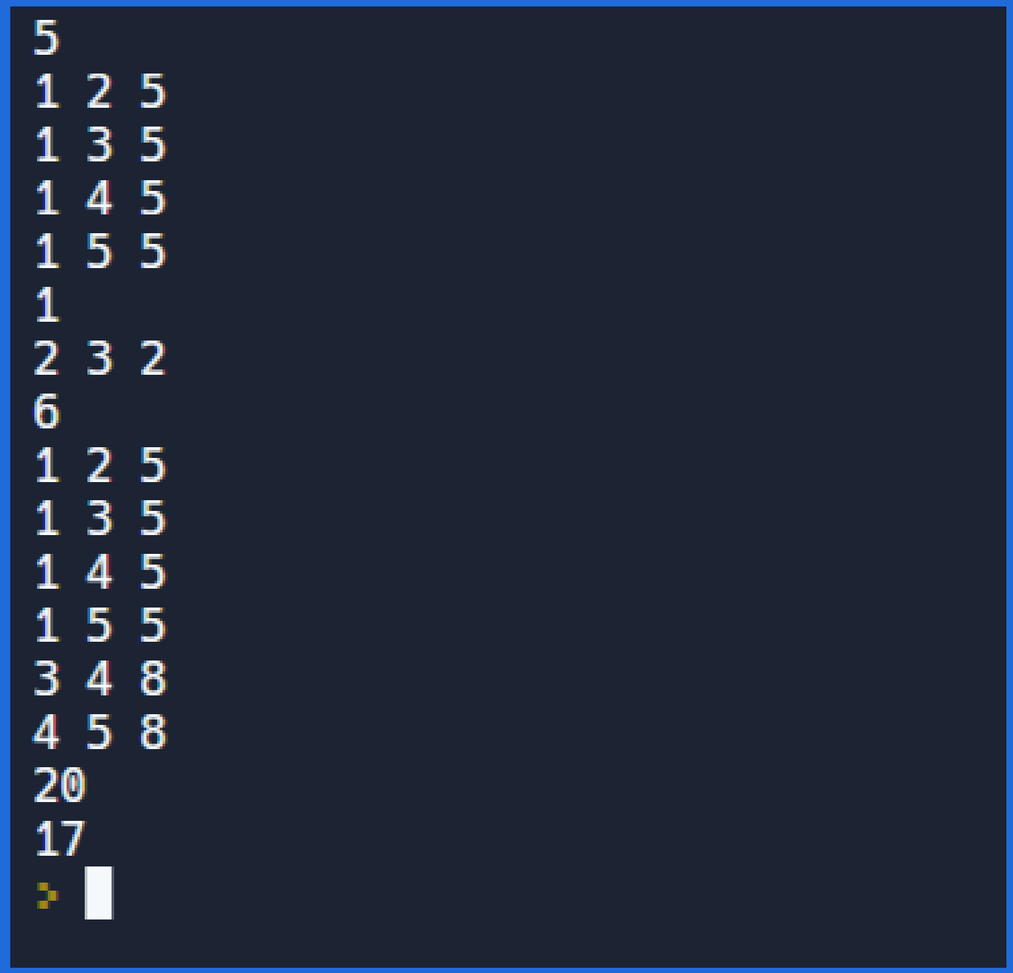
```java
public class Dupla implements Comparable<Dupla>{
    int first;
    int second;

    public Dupla(int a, int b){
        first = a;
        second = b;
    }

    @Override
    public int compareTo(Dupla duplaAlt) {
        return Integer.compare(first, duplaAlt.first);
    }
}
```

```
//iniciando grafo
adj = new ArrayList[n];
visited = new boolean[n];
for(int i = 0; i<n; i++){
    adj[i] = new ArrayList<>();
    visited[i] = false;
}

//arestas extras
m = sc.nextInt();
for(int i = 0; i < m; i++){
    p = sc.nextInt()-1;
    s = sc.nextInt()-1;
    val = sc.nextInt();
    adj[p].add(new Dupla(val, s));//adicionando aresta s em p com valor val
    adj[s].add(new Dupla(val, p));//adicionando aresta p em s com valor val
}
```

```java
//arestas originais
m = sc.nextInt();
for(int i = 0; i < m; i++){
    p = sc.nextInt()-1;
    s = sc.nextInt()-1;
    val = sc.nextInt();
    adj[p].add(new Dupla(val, s));//adicionando aresta s em p com valor val
    adj[s].add(new Dupla(val, p));//adicionando aresta p em s com valor val
}

//Algoritmo de Prim a partir da raiz = 0
visited[0] = true; //marco como visitado
//adicionando todos os vértices a fila de prioridade
for(int i = 0; i < adj[0].size(); i++){
    lPrior.add(new Dupla(adj[0].get(i).first, adj[0].get(i).second));
}
//enquanto a fila de prioridade não for vazia:
/*Colocar em dupla auxiliar "a" o menor elemento(topo da fila de prioridade)
* se vértice ligado a aresta "a"(segundo elemento da dupla) não tiver sido visitado
* somamos a carga da aresta "a" aresta ao total, marcamos como visitado e aplicamos Prim a ele,
* adicionando a fila de prioridade todos os seus adjacentes, a partir daí laço while garante a repetição
* em todos os elementos do grafo*/
```

```java
while(!lPrior.isEmpty()){
    //cria dupla auxiliar que guarda primeiro elemento da fila e retira ele da fila(poll)
    Dupla a = new Dupla(lPrior.peek().first, lPrior.poll().second);
    //se não visitado
    if(!visited[a.second]){
        s2 = s2 + a.first;//soma ao total o valor da aresta
        visited[a.second] = true; //marca vértice como visitado
        //adiciona todos os adjacentes ao vértice na fila
        for(int i = 0; i < adj[a.second].size(); i++){
            lPrior.add(new Dupla((adj[a.second].get(i).first), adj[a.second].get(i).second));
        }
        //a fila é reordenada ao adicionar um novo item de acordo com a ordem de prioridades crescente
        //(primeiro elemento sempre o menor)
    }
}
//printa primeira soma e segunda soma
System.out.println(s1+"\n"+s2);
```

Exemplos rodados



10 Driving Range

Driving Range

Hoje em dia, muitas montadoras estão desenvolvendo carros que funcionam com eletricidade em vez de gasolina. As baterias usadas nestes carros são geralmente muito pesadas e caras, por isso os projetistas devem fazer escolhas importantes ao determinar a capacidade da bateria e, portanto, o alcance desses veículos. Sua tarefa é ajudar a determinar o alcance mínimo necessário para que o carro possa viajar entre quaisquer duas cidades do continente.

A rede rodoviária do continente consiste em cidades ligadas por estradas bidirecionais de diferentes comprimentos. Cada cidade contém uma estação de carregamento. Ao longo de uma rota entre duas cidades, o carro pode passar por qualquer número de cidades, mas a distância entre cada par de cidades consecutivas ao longo da rota não deve ser maior que a autonomia do carro. Qual é a autonomia mínima do carro para que haja uma rota que satisfaça esta restrição entre cada par de cidades do continente?

Driving Range

Entrada: A primeira linha da entrada contém dois inteiros não negativos n e m, o número de cidades e estradas. Cada um desses números inteiros não é maior que um milhão. As cidades são numeradas de 0 a n-1. A primeira linha é seguida por m mais linhas, cada uma descrevendo uma estrada. Cada linha contém três inteiros não negativos. Os primeiros dois números inteiros são os números das duas cidades ligadas pela estrada. O terceiro inteiro é o comprimento da estrada, um número inteiro positivo não maior que 10^9.

Driving Range

Saída: Para cada rede rodoviária, produza uma linha contendo um número inteiro, o alcance mínimo do carro que lhe permite dirigir de qualquer cidade para qualquer outra cidade. Se não for possível dirigir de uma cidade para outra, independentemente da autonomia do carro, em vez disso, imprima uma linha contendo a palavra IMPOSSÍVEL.

Sample Input 1

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

Sample Output 1

```
4
```

Sample Input 2

```
2 0
```

Sample Output 2

```
IMPOSSIBLE
```

Driving Range

Estrutura de dados: Listas de Adjacência

Vértices: Cidades

Arestas: Rotas entre as cidades

Driving Range

```java
import java.util.ArrayList;

public class Aresta implements Comparable<Aresta> {
    int origem, destino, peso;

    public Aresta(int origem, int destino, int peso) {
        this.origem = origem;
        this.destino = destino;
        this.peso = peso;
    }

    //Método que compara arestas pelo peso
    @Override
    public int compareTo(Aresta outra) {
        return Integer.compare(this.peso, outra.peso);
    }
}
```

```java
class Grafo {
    ArrayList<Aresta>[] adj; //lista de adjacências
    boolean[] visited;
    PriorityQueue<Aresta> Prioridade = new PriorityQueue<>();
    int custoMin = 0;

    public Grafo(int v) {
        adj = new ArrayList[v];
        visited = new boolean[v];

        for (int i = 0; i < v; i++) {
            adj[i] = new ArrayList<>();
            visited[i] = false;
        }
    }

    public void adicionarAresta(int u, int v, int peso) {
        adj[u].add(new Aresta(u, v, peso));
        adj[v].add(new Aresta(v, u, peso));
    }
```

```java
//Encontrar a árvore geradora miníma
public void prim() {
    //marca o primeiro vértice como visitado
    visited[0] = true;
    //adiciona as arestas conectadas a
    //a fila de prioridade
    for (int i = 0; i < adj[0].size(); i++) {
        Prioridade.add(adj[0].get(i));
    }
    //Construindo a árvore geradora miníma
    //(processa todas as arestas)
    while (!Prioridade.isEmpty()) {
        Aresta a = Prioridade.poll();
        //verifica se o vértice adjacente
        //já foi visitado
        if (!visited[a.destino]) {
            visited[a.destino] = true;
            custoMin = a.peso; //guarda o custo
            custoMin = a.peso; //guarda o custo
            //adiciona todas as arestas conectadas
            //ao vértice destino na fila de
            //prioridade
            for (int i = 0; i < adj[a.destino].size(); i++)
                Prioridade.add(adj[a.destino].get(i));
            }
        }
    }
    if(custoMin <= 0){
        System.out.println(x:"IMPOSSÍVEL");
    }else{
        System.out.println(custoMin);
    }
```

Exemplos rodados