Exercícios-Problemas (EP)

Semana 6:

https://www.facom.ufu.br/~albertini/grafos/

EP11: UVa 00544 - Heavy Cargo

Big Johnsson Trucks Inc. is a company specialized in manufacturing big trucks. Their latest model, the Godzilla V12, is so big that the amount of cargo you can transport with it is never limited by the truck itself. It is only limited by the weight restrictions that apply for the roads along the path you want to drive. Given start and destination city, your job is to determine the maximum load of the Godzilla V12 so that there still exists a path between the two specified cities.

Input

The input file will contain one or more test cases. The first line of each test case will contain two integers: the number of cities n ($2 \leq n \leq 200$) and the number of road segments r ($1 \leq r \leq 19900$) making up the street network. Then r lines will follow, each one describing one road segment by naming the two cities connected by the segment and giving the weight limit for trucks that use this segment. Names are not longer than 30 characters and do not contain white-space characters. Weight limits are integers in the range 0..10000. Roads can always be travelled in both directions. The last line of the test case contains two city names: start and destination. Input will be terminated by two values of 0 for n and r.

EP11: UVa 00544 - Heavy Cargo

Output

For each test case, print three lines:

- **a line saying ‘Scenario #x’ where x is the number of the test case**
- **a line saying ‘y tons” where y is the maximum possible load**
- **a blank line**

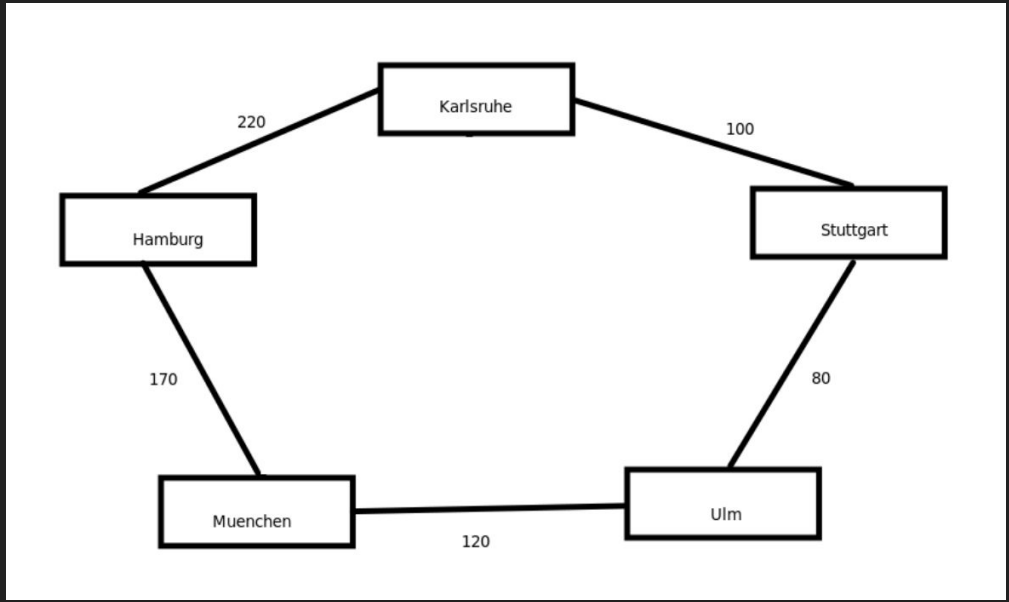
Sample Input

```
4 3
Karlsruhe Stuttgart 100
Stuttgart Ulm 80
Ulm Muenchen 120
Karlsruhe Muenchen
5 5
Karlsruhe Stuttgart 100
Stuttgart Ulm 80
Ulm Muenchen 120
Karlsruhe Hamburg 220
Hamburg Muenchen 170
Muenchen Karlsruhe
0 0
```

Sample Output

```
Scenario #1 80 tons

Scenario #2 170 tons
```



EP11: UVa 00544 - Heavy Cargo

```python
def main():
    X = 0  # Contador de cenarios
    results = [] # Array para armazenar resultados

    while True:
        N, R = map(int, input().split()) # Leitura do numero de cidades e numero de estradas
        X += 1 # Incrementar o numero do cenario

        if N == 0 or R == 0: # Condição de parada
            break

        grafo = []
        for _ in range(R):
            c1, c2, peso = input().split() # Leitura das estradas e pesos
            grafo.append((c1, c2, int(peso)))

        partida, destino = input().split() # Leitura das cidades de partida e destino

        max_load = get_maximum_load(grafo, partida, destino) # Chamada da função para obter peso máximo

        results.append(("Scenario #" + str(X), str(max_load) + " tons")) # Formatação da resposta

    for cenario, peso in results: # Print das repostas
        print(cenario)
        print(peso)
        print()

if __name__ == "__main__":
    main()
```

- Leitura de inputs
- Chamada das funções
- Print resultados

EP11: UVa 00544 - Heavy Cargo

- Kruskal para árvore geradora máxima
- Ordenação decrescente para maximizar os valores de peso
- Retornar MST

```
MST:  [('Karlsruhe', 'Hamburg', 220), ('Hamburg', 'Muenchen', 170), ('Ulm', 'Muenchen', 120), ('Karlsruhe', 'Stuttgart', 100)]
```

```python
def kruskal(grafo):
    def find(parent, cidade): # Verifica conjunto de um vértice encontrando sua raiz
        if cidade not in parent:
            parent[cidade] = cidade
        elif parent[cidade] != cidade:
            parent[cidade] = find(parent, parent[cidade])
        return parent[cidade]

    def union(parent, cidade1, cidade2): # União dos dois conjuntos
        raiz_cidade1 = find(parent, cidade1)
        raiz_cidade2 = find(parent, cidade2)
        parent[raiz_cidade1] = raiz_cidade2

    mst = [] # Arestas da árvore geradora maxima
    parent = {} # Armazenar os conjuntos distuntos para verificar ciclos
    grafo.sort(key=lambda x: x[2], reverse=True) # Arestas do grafo em ordem decrescente de peso

    for cidade1, cidade2, peso in grafo: # Itera sob as arestas do grafo
        raiz_cidade1 = find(parent, cidade1)
        raiz_cidade2 = find(parent, cidade2)

        if raiz_cidade1 != raiz_cidade2: # Verifica se a raiz da cidade 1 e a raiz da cidade 2 estão em conjuntos diferentes
            mst.append((cidade1, cidade2, peso)) # Adiciona a aresta que faz parte da árvore geradora mínima
            union(parent, raiz_cidade1, raiz_cidade2) # Une os conjuntos da cidade1 com a cidade 2
    return mst
```

EP11: UVa 00544 - Heavy Cargo

- Criar lista de adjacência a partir da MST
- DFS para encontrar menor caminho
- Retorna lista de vértices do caminho

```
Lista de Adjacência:  {'Karlsruhe': [('Hamburg', 220), ('Stuttgart', 100)], 'Hamburg': [('Karlsruhe', 220), ('Muenchen', 170)], 'Muenchen':
[('Hamburg', 170), ('Ulm', 120)], 'Ulm': [('Muenchen', 120)], 'Stuttgart': [('Karlsruhe', 100)]}
```

```
Menor caminho:  ['Muenchen', 'Hamburg', 'Karlsruhe']
```

```python
def criar_lista_adjacencia(grafo):  # Cria lista de adjacencia com os vertices sendo as cidades
    lista_adjacencia = {}
    for cidade1, cidade2, peso in grafo:
        if cidade1 not in lista_adjacencia: # Cria nó para cidade 1
            lista_adjacencia[cidade1] = []
        if cidade2 not in lista_adjacencia: # Cria nó para cidade 2
            lista_adjacencia[cidade2] = []
        lista_adjacencia[cidade1].append((cidade2, peso)) # Criar arestas
        lista_adjacencia[cidade2].append((cidade1, peso))
    return lista_adjacencia

def dfs(lista_adjacencia, partida, destino, visited, menor_caminho): # DFS para achar menor caminho entre 2 cidades
    visited.add(partida)
    menor_caminho.append(partida) # Adiciona cidade no menor caminho

    if partida == destino:  # Condição de parada
        return menor_caminho

    for vizinho, peso in lista_adjacencia[partida]:
        if vizinho not in visited:
            resultado = dfs(lista_adjacencia, vizinho, destino, visited, menor_caminho.copy())
            if resultado:
                return resultado

    return None
```

EP11: UVa 00544 - Heavy Cargo

- Kruskal para árvore geradora máxima
- Criar estrutura de lista de adjacência a partir da MST
- DFS para achar menor caminho
- Filtrar somente as arestas do menor caminho
- Retornar menor valor de peso entre as arestas

```
Arestas filtradas:  [('Karlsruhe', 'Hamburg', 220), ('Hamburg', 'Muenchen', 170)]
```

```python
def get_maximum_load(grafo, partida, destino):
    mst = kruskal(grafo) # Arvore geradora maxima
    visited = set() # Set de visitados
    lista_adjacencia = criar_lista_adjacencia(mst) # Criar estrutura de lista de adjacencia a partir da mst gerada no kurskal
    menor_caminho = dfs(lista_adjacencia, partida, destino, visited, []) # Chamada da dfs para achar menor caminho dentro da lista de adjacencia

    filtered_arestas = [] # Array de arestas que são contem as cidade do menor caminho

    for cidade1, cidade2, peso in grafo:  # Filtra o grafo apenas com as arestas do menor caminho
        if cidade1 in menor_caminho and cidade2 in menor_caminho:
            filtered_arestas.append((cidade1, cidade2, peso))

    menor_peso = float('inf')  # Inicializa com infinito para encontrar o menor valor

    for cidade1, cidade2, peso in filtered_arestas:
        if peso < menor_peso: # Menor valor é o peso máximo
            menor_peso = peso

    return menor_peso
```