Lista Avaliativa 2

Henrique Parede de Souza - 260497

Problema 1

Minha Camiseta Me Serve

Dificuldade: 6

Entrada: Número de instâncias de teste, e para cada instância o número total de camisetas (**N**), o número de pessoas (**M**) e para cada pessoa dois tamanhos de camiseta que a sirvam (de XS a XXL).

Saída esperada: Para cada instância, **YES** se existe pelo menos uma maneira de distribuir as camisetas tal que todos tenham uma camiseta que lhes sirvam, ou **NO** caso contrário.

Algoritmo utilizado: Fluxo Máximo (Ford-Fulkerson)

Podemos modelar o problema representando cada tamanho de camisetas e cada pessoa como um vértice de um grafo ponderado direcionado, onde todas as camisetas estarão ligadas à uma fonte **i** por meio de arestas de capacidade **N/6** e todas as pessoas estarão ligadas a um sorvedouro **f** por meio de arestas de capacidade 1. Além disso, se um tamanho de camisa serve a uma determinada pessoa, então a aresta dessa camisa até essa pessoa também terá capacidade 1.

Aplicando o algoritmo de Ford-Fulkerson, podemos obter o fluxo máximo do grafo. Se esse fluxo for maior ou igual ao número de pessoas, significa que existe pelo menos uma maneira de distribuir as camisetas de forma que todas as pessoas tenham uma camisa que lhes sirva.

Complexidade: Para cada instância, executamos uma vez o Algoritmo de Ford-Fulkerson, que é $O(|f| \cdot E)$. Como existem **p** instâncias na mesma entrada, executaremos p vezes Ford-Fulkerson.

Motivo para funcionar: A solução proposta utiliza apenas o Algoritmo de Ford-Fulkerson, que como provado em aula é correto e de fato obtêm o fluxo máximo do grafo. Além disso, realiza-se uma BFS para verificar se existe caminho aumentador na rede, e como visto em aula, a BFS também é correta e consegue retornar se o vértice f é alcançável a partir do vértice i.

```python
from collections import defaultdict
from dataclasses import dataclass, field
```

```
sizes = ["XS", "S", "M", "L", "XL", "XXL"]

@dataclass
class Queue:
        Q: list = field(default_factory=list)

        def dequeue(self):
                return self.Q.pop(0)
        def enqueue(self, num):
                self.Q.append(num)
        def size(self):
                return len(self.Q)

class Graph:
        capacidade: defaultdict
        vertices: list

        def __init__(self, n_camisas:int, n_pessoas:int, preferencias:dict):
                self.capacidade = defaultdict(lambda: defaultdict(str))
                self.vertices = ["i"] +  sizes + [str(i) for i in range(1, n_pessoas + 1)] +
["f"]

                for size in sizes:
                        self.capacidade["i"][size] = n_camisas // 6
                        self.capacidade[size]["i"] = 0
                        for j in range(1, n_pessoas + 1):
                                self.capacidade[size][str(j)] = 0
                                self.capacidade[str(j)][size] = 0
                                self.capacidade[str(j)]["f"] = 1
                                self.capacidade["f"][str(j)] = 0

                for i in range(1, n_pessoas + 1):
                        self.capacidade[preferencias[i][0]][str(i)] = 1
                        self.capacidade[str(i)][preferencias[i][0]] = 0
                        self.capacidade[preferencias[i][1]][str(i)] = 1
                        self.capacidade[str(i)][preferencias[i][1]] = 0

        def __bfs(self):
                visitado = {}
                pai = {}

                for u in self.vertices:
                        visitado[u] = False
                        pai[u] = None

                visitado["i"] = True
                Q = Queue()
                Q.enqueue("i")

                while Q.size() != 0:
                        u = Q.dequeue()
                        for v in self.capacidade[u]:
                                if self.capacidade[u][v] > 0 and not visitado[v]:
                                        visitado[v] = True
                                        pai[v] = u
                                        Q.enqueue(v)

                                        if v == "f":
                                                return True, pai

                return False, None
```

```
        def FordFulkerson(self) -> int:
                fluxo_maximo = 0

                while True:
                        existe_caminho_aumentador, pai = self.__bfs()
                        if not existe_caminho_aumentador:
                                break
                        capacidade_residual = float("inf")
                        atual = "f"
                        while atual != "i":
                                capacidade_residual = min(capacidade_residual,
self.capacidade[pai[atual]][atual])
                                atual = pai[atual]

                        fluxo_maximo += capacidade_residual

                        atual = "f"
                        while atual != "i":
                                self.capacidade[pai[atual]][atual] -= capacidade_residual
                                self.capacidade[atual][pai[atual]] += capacidade_residual
                                atual = pai[atual]

                return fluxo_maximo


instancias = int(input())
for _ in range(instancias):
        n_camisas, n_pessoas = map(int, input().split())
        preferencias = {}
        for i in range(1, n_pessoas + 1):
                preferencias[i] = tuple(input().split())
        graph = Graph(n_camisas, n_pessoas, preferencias)
        fluxo_maximo = graph.FordFulkerson()
        print("YES" if fluxo_maximo >= n_pessoas else "NO")
```

Problema 2

Reduzindo Detalhes em um Mapa

Dificuldade: 3

Entrada: Na primeira linha temos o número de cidades **N** e o número de rodovias **M**. Nas próximas M linhas temos 3 inteiros, **u**, **v** e **c**, indicando que existe uma aresta de u para v com peso c.

Saída esperada: Comprimento do subconjunto de arestas com menor peso total.

Algoritmo utilizado: Árvore Geradora Mínima (Algoritmo de Kruskal)

O problema apresentado pode ser resolvido através de uma implementação simples do algoritmo de Kruskal, onde ordenamos as arestas do grafo em ordem não-decrescente de peso e escolhemos a aresta de menor peso desta ordenação que una dois conjuntos

disjuntos para compor a árvore geradora mínima. A saída do programa começará como zero, sendo incrementado o valor da aresta escolhida a cada iteração.

Complexidade: $O(E \cdot log\ E)$, pois como visto em aula a operação mais custosa do Algoritmo de Kruskal é a ordenação.

Motivo para funcionar: Por se tratar de uma aplicação simples de Kruskal, sabemos que o algoritmo é correto e devolve o tamanho da AGM, como visto em aula.

Python

```python
def makeset(pontos):
	conjuntos = []
	for ponto in pontos:
	conjuntos.append({ponto})
	return conjuntos

def findset(v):
	for conjunto in conjuntos:
	if v in conjunto:
		return conjunto
	return None

def union(vertices):
	conjuntos.append(findset(vertices[0]) | findset(vertices[1]))
	conjuntos.remove(findset(vertices[0]))
	conjuntos.remove(findset(vertices[1]))
	return conjuntos

def kruskal(arestas):
	comprimento = 0

	w = dict(sorted(arestas.items(), key=lambda item: item[1]))

	for vertices, e in w.items():
		if findset(vertices[0]) != findset(vertices[1]):
			conjuntos = union(vertices)
			comprimento += e

	return comprimento

cidades, rodovias = map(int, input().split())
vertices = [i for i in range(1, cidades + 1)]
arestas = {(i, j): c for i, j, c in [map(int, input().split()) for _ in
range(rodovias)]}

conjuntos = makeset(vertices)
print(f'{kruskal(arestas)}')
```

Problema 3

106 Milhas para Chicago

Dificuldade: 6

Entrada: Na primeira linha temos o número de intersecções **n** e o número de ruas **m**. Nas próximas m linhas teremos uma descrição da rua através de três inteiros **a**, **b** e **p**, sendo a e b duas cidades distintas e p a probabilidade de passar em segurança pela aresta (a, b).

Saída esperada: A probabilidade associada ao caminho mais seguro entre a origem e o destino.

Algoritmo utilizado: Caminhos Mínimos (Floyd-Warshall)

O problema apresentado pode ser resolvido por uma variação do algoritmo de Floyd-Warshall que calcula a máxima probabilidade de atravessar em segurança para todos os vértices, imprimindo apenas a probabilidade do caminho entre a origem e o destino.

Complexidade: $O(V^3)$

Motivo para funcionar: Como visto em aula, o algoritmo de Floyd-Warshall é capaz de resolver os problemas de caminho mínimo e de caminho máximo entre quaisquer pares de vértices. Portanto o código é correto e retorna a maior probabilidade entre a origem e o destino.

C/C++

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

int main() {
	while (true) {
		int intercecoes, ruas;
		scanf("%d", &intercecoes);
		if (intercecoes == 0)
			break;
		scanf("%d", &ruas);

		double **w = (double **)malloc((intercecoes + 1) * sizeof(double *));
		for (int i = 0; i <= intercecoes; i++) {
			w[i] = (double *)malloc((intercecoes + 1) * sizeof(double));
			for (int j = 0; j <= intercecoes; j++)
				w[i][j] = 0.0;
		}
```

```
                for (int r = 0; r < ruas; r++) {
                        int i, j;
                        double p;
                        scanf("%d %d %lf", &i, &j, &p);
                        w[i][j] = p * 1e-2;
                        w[j][i] = p * 1e-2;
                }

                for (int k = 1; k <= intercecoes; k++)
                        for (int i = 1; i <= intercecoes; i++)
                                for (int j = 1; j <= intercecoes; j++)
                                        if (w[i][j] < w[i][k] * w[k][j])
                                                w[i][j] = w[i][k] * w[k][j];

                printf("%.6f percent\n", w[1][intercecoes] * 1e2);

                for (int i = 0; i <= intercecoes; i++)
                        free(w[i]);
                free(w);
        }

        return 0;
}
```