Lista Avaliativa 1

Henrique Parede de Souza - 260497

Exercício 1a

Seja G = (V , E) um grafo conexo. Primeiro iremos provar o seguinte lema:

Se G possui $d(v) \geq 2 \ \forall v \in V$, então G contém pelo menos um ciclo.

Escolha qualquer vértice v_0 de G e comece a montar um passeio a partir de v_0, escolhendo arestas ainda não visitadas. Pela propriedade da paridade, é possível obter um passeio P = {v_0 , a_0 , v_1 , … }, onde o próximo vértice será qualquer vértice adjacente ao atual exceto o anterior. Sendo assim, após algum tempo, necessariamente haverá um vértice repetido em P. O caminho fechado delimitado por esse vértice repetido necessariamente será um ciclo em G. ❒

Com o lema anterior em mãos, agora podemos provar o enunciado:

Se G é conexo e $\forall v \in V(d(v) = 2p, p \in \mathbb{N})$, então G é euleriano.

Seja n = |E|. Note que para n = 0, ou o grafo é vazio (e por vacuidade é euleriano) ou ele possui apenas 1 vértice (d(v) = 0.p, portanto é euleriano), e para n = 1 e 2, o grafo não pode satisfazer a propriedade da paridade. Sabemos que se G possui um passeio fechado que passa exatamente única vez por cada aresta, então G é euleriano. Com isso em mente, podemos realizar uma indução forte em n≥2:

Base: Para n = 3, a única configuração possível é um triângulo, que é trivialmente cíclico. ❒

Hipótese de Indução: $\forall j \in [3, n-1]$, se G = (V, E) (com |E| = j) é conexo e todo $v \in V$ obedece a propriedade da paridade, então G é euleriano.

Passo Indutivo: Considere um grafo G = (V, E) com n + 1 arestas. Como G é conexo, todos os vértices de G obedecem o critério de paridade, então pelo lema anterior, G possui pelo menos um ciclo A = (V', E'). Se |E'| = n+1, todas as arestas de G também pertencem a A, provando assim o enunciado. Caso contrário tome B = (V, E \ E'). Como B pode ser desconexo, ao remover A cada vértice em B pode ficar com grau 0 ou ter seu grau reduzido em duas unidades por causa do ciclo (ou seja, este vértice ainda é par). Aplicando a HI, constatamos que cada componente com mais de dois vértices de B é euleriano, possuindo portanto um ciclo. Ao juntarmos A com B, temos como resultado um grafo euleriano que é o próprio G. **qed**

Exercício 2a

O algoritmo apresentado realiza uma DFS em G a fim de descobrir as componentes fortemente conexas. Podemos dividir a nossa análise de complexidade em algumas etapas:

- Inicializar a lista de cores: O(V)
- Percorrer todos os vértices de G (laço da linha 8 da *Main*): O(V)
- DFS: O(V + E)
- Pop da pilha (laço linha 23 da *Find_CC):* O(V) no pior caso

Portanto o algoritmo é O(V + E). Para provar a sua corretude, podemos usar o resultado visto em aula que garante a corretude da DFS. Portanto, precisamos provar que (1) min(menor[u], menor[v]) retorna o vértice com menor tempo de descoberta alcançável a partir de u e (2) os vértices da mesma componente são desempilhados juntos.

(1) Como a DFS visita todos os vértices até no caso onde existem vértices sem adjacência, pelo Princípio da Boa Ordenação temos que qualquer subconjunto de vértices de G possui um vértice com o menor tempo de descoberta. Fazemos uma indução na distância (d) entre um vértice qualquer (v) e o menor vértice:

Base: Para d = 0, é trivial que o vértice com menor tempo de descoberta alcança a si próprio.❒

Hipótese de Indução: se d = n-1, v alcança o vértice com menor tempo de descoberta.

Passo Indutivo: Considere d = n e sejam u o vértice atual e v o próximo vértice adjacente a u chamado na recursão. Como provamos a existência de um mínimo pelo PBO, podemos aplicar a hipótese de indução em v, provando que ele alcança o mínimo. Como u é vizinho de v, u também alcança o mínimo. **qed**

(2) Suponha, por absurdo, que existe um vértice v pertencente a uma componente fortemente conexa qualquer que não é desempilhado junto com os outros. Temos que o vértice u que entra no if da linha 18 é o primeiro vértice da componente, e que todos os vértices acima dele serão desempilhados na mesma componente. Como v não é desempilhado, d[v] < d[u], porém como eles estão na mesma componente, u não pode ser o primeiro vértice da componente. Logo chegamos em uma contradição, provando portanto que todos os vértices da mesma componente são desempilhados juntos. **qed**

Exercício 2b

Python

```python
def dfs_visit(u:int, adj:dict, cor:dict, caminho:list, eh_euleriano:bool):
	cor[u] = "c"
	caminho.append(u)

	if len(adj[u]) % 2 != 0:
		eh_euleriano = False
		return

	for v in adj[u]:
		if u != v and cor[v] == "b":
			caminho.append((u, v))
			dfs_visit(v, adj, cor, caminho, eh_euleriano)
	cor[u] = "p"


def dfs(adj):
	cor = {}
	caminho = []
	segunda_vez = False
	eh_euleriano = True

	for u in adj.keys():
		cor[u] = "b"

	for u in adj.keys():
		if cor[u] == "b" and segunda_vez:
			eh_euleriano = False
		if cor[u] == "b" and not segunda_vez:
			dfs_visit(u, adj, cor, caminho, eh_euleriano)
			caminho.append([caminho[-1], caminho[0]])
			caminho.append(caminho[0])
			segunda_vez = True

	return (True, caminho) if eh_euleriano else (False, None)
```

O algoritmo apresentado utiliza uma DFS para determinar um caminho euleriano. Pelo teorema provado no Exercício 1, se G é conexo e possui todos os vértices com grau par, então G é euleriano. Baseando-se nisso, foi proposto uma implementação de *dfs_visit* que verificasse a paridade do grau de cada vértice *da* recursão, e retornasse *False* caso exista algum vértice com grau ímpar. Além disso, na função *dfs* há a verificação do número de componentes conexas de G, pois se houver mais que uma, G não é euleriano.

Analisando a complexidade, temos:

- Inicializar a lista de cores: O(V)
- Varrer a lista de adjacência 1 vez: O(V)
- DFS: O(V + E)

Portanto o algoritmo é O(V + E). Note que se soubermos com antecedência que G é conexo e não precisarmos imprimir o caminho, bastaria uma iteração na lista de adjacência para verificar a paridade de cada vértice, fazendo assim um algoritmo O(V).

Sabendo que o algoritmo de DFS em si é correto, para provar a corretude da nossa variação devemos mostrar que o nosso algoritmo é capaz de determinar se existem componentes desconexas (1) e se o grau de cada vértice é par (2).

(1) Considere o segundo laço da função *dfs*. Se houver mais que uma chamada de *dfs_visit*, significa que há mais de uma componente em G. Suponha por absurdo que a afirmação anterior seja falsa. Após a primeira chamada da função, todos os vértices da árvore de busca terão a cor preta, logo nenhum vértice da segunda chamada poderia ser ligado a um vértice da primeira, formando portanto duas componentes distintas, o que os leva a uma contradição. Sendo assim, a variável *segunda_vez* garante que se existir um vértice branco após a dfs inicial, o retorno será *False.*

(2) A verificação da paridade de cada vértice é feita em cada chamada de *dfs_visit*. Como a verificação da conexidade será realizada pela função casca, podemos considerar sem perda de generalidade que o grafo será conexo (caso não seja, a função casca irá tratar esse caso e retornar *False*). Como é conhecido que o algoritmo de dfs visita todos os vértices de G, temos que a verificação da paridade ocorrerá para todos os vértices.

Exercício 3

Problema 1 - Árvore Genealógica

Dificuldade: 4

Entrada: Quantidade de pessoas(**M)** e de relacionamentos(**N**), além de N relações do tipo (Nome1, relacionamento, Nome2).

Saída esperada: Número de famílias distintas (ou seja, o número de componentes do grafo).

Algoritmo utilizado: DFS

Na solução do problema, o tipo de relação que uma pessoa possui com a outra não é relevante. Sendo assim, podemos modelar a entrada como um grafo onde cada vértice representa uma pessoa e cada aresta um relacionamento. Para isso foi utilizado um dicionário representando uma lista de adjacência:

```
adj = {"João": ["Pedro", "Santiago"],
"Pedro": ["João"], "Santiago": ["Pedro"]}
```

A partir desta lista, foi possível realizar uma DFS para determinar quantas componentes o grafo possui. Para isso, toda vez que a função *dfs_visit* era chamada pela *dfs*, incrementava-se o contador. Com isso temos a resposta final.

Complexidade: A complexidade da DFS é O(V + E).

Motivo para funcionar: Sabemos que a DFS é correta, e que cada chamada de *dfs_visit* em *dfs* começa em um novo componente.

Python

```python
def dfs(adj):
	visitado = {}
	tamanho_componente = 0
	for u in adj.keys():
		visitado[u] = False
	for u in adj.keys():
		if not visitado[u]:
		dfs_visit(u, adj, visitado)
			tamanho_componente += 1
	print(tamanho_componente)


def dfs_visit(u, adj, visitado):
	visitado[u] = True
	for v in adj[u]:
	if not visitado[v]:
		dfs_visit(v, adj, visitado)
	visitado[u] = True

_, relacoes = map(int, input().split())

adj = {}
jafoi = []

for _ in range(relacoes):
	nome1, _, nome2 = input().split()
	if not nome1 in jafoi:
		adj[nome1] = [nome2]
		jafoi.append(nome1)
	else:
		adj[nome1].append(nome2)
```

```
        if not nome2 in jafoi:
                adj[nome2] = [nome1]
                jafoi.append(nome2)
        else:
                adj[nome2].append(nome1)

dfs(adj)
```

Problema 2 - Rodovia

Dificuldade: 6

Entrada: Número de cidades (**N**), seguido de N linhas representando as estradas entre duas cidades (1 2 -> existe uma estrada de 1 para 2).

Saída esperada: "S" se todos os vértices podem ser alcançados a partir de qualquer outro vértice, "N" caso contrário.

Algoritmo utilizado: DFS

Na prática, o problema consiste em determinar se existe uma única componente fortemente conexa em um grafo. Para isso foi utilizado um dicionário representando uma lista de adjacência, onde cada par *u*: [v_1, ..., v_j] significa que existe uma aresta direcionada de u até v_i, $\forall$ i $\in$ [1, j].

```
adj = {1: [2, 3], 2: [3], 3: []}
```

A partir desta lista, foi aplicado um algoritmo para fazer uma cópia transposta de *adj*, chamada de *adj_trans*. São realizadas as chamadas de DFS(*adj*) e de DFS(*adj_trans*), onde cada DFS retorna o número de componentes do grafo (como feito no exercício anterior). Se ambas as DFS retornarem exatamente 1 componente cada, o grafo é fortemente conexo, portanto é possível alcançar qualquer vértice a partir de outro.

Complexidade: São realizadas 2 DFS's. A complexidade da DFS é O(V + E), porém como V = E, temos O(2V) = O(V).

Motivo para funcionar: Como dito no exercício anterior, sabemos que a DFS é correta e que cada chamada de *dfs_visit* em *dfs* começa em um novo componente, retornando portanto o número de componentes. Se só houver uma componente, o grafo será fortemente conexo, e por consequência, haverá caminho entre qualquer par de vértices.

Python

```python
import sys

def dfs_visit(u:int, adj:dict, cor:dict):
	cor[u] = "c"
	for v in adj[u]:
		if u != v and cor[v] == "b":
			dfs_visit(v, adj, cor)
	cor[u] = "p"

def dfs(adj:dict):
	cor = {}
	n_componentes = 0
	for u in adj.keys():
		cor[u] = "b"
	for u in adj.keys():
		if cor[u] == "b":
			dfs_visit(u, adj, cor)
			n_componentes += 1
	if n_componentes == 1:
		return True
	return False

sys.setrecursionlimit(200000)
cidades = int(input())
adj = {}

for i in range(1, cidades + 1):
	adj[i] = []

for _ in range(cidades):
	i, j = map(int, input().split())
	adj[i].append(j)

adj_trans = {v: [] for v in adj.keys()}
for u, vizinhos in adj.items():
	for v in vizinhos:
		adj_trans[v].append(u)


if dfs(adj) and dfs(adj_trans):
	print("S")
else:
	print("N")
```