Atividade Prática A3 – Grafos (INE 5413)

Leonardo Rocha - 19102922
André Luiz - 19150871
Thiago Comelli - 19100546

1. Introdução

Para realização deste trabalho selecionamos aleatoriamente uma pessoa pra ficar responsável por cada algoritmo. Contudo, dúvidas que foram aparecendo damos um apoio mútuo uns aos outros. Para esclarecimento de dúvidas utilizamos o Discord e Whatsapp.

Organização do trabalho:

Na raiz da atividade, deixamos o arquivo de teste usado para cada exemplo e os respectivos códigos:

1. **grafo.py [Arquivo Base]**
2. **edmonds_karp.py [Etapa 1]**
3. **Hopcroft-Karp.py [Etapa 2]**

Para este trabalho precisamos fazer pequenas adaptações no arquivo base **grafo.py,** para que fosse possível fazer a leitura dos arquivo *.gr.

2. Edmons Karp

Como estamos lidando com um novo tipo de arquivo foi necessário adaptações no grafo.py e toda a sua estrutura para melhorar organizar os diferentes tipos de arquivo.

Para realização deste algoritmo foi necessário a utilização de dois principais algoritmos, Ford Fulkerson e Edmonds Karp. O algoritmo de Ford Fulkerson se baseia em três ideias principais: redes residuais, caminhos aumentantes e cortes. Este algoritmo vai aumentando iterativamente o valor do fluxo. Para melhorar o algoritmo de Ford Fulkerson, o Edmonds Karp busca melhorar o desempenho através do algoritmo Busca em Largura adaptado.

```python
def ford_fulkerson():
    global grafo

    fluxo = 0

    fonte = grafo.fonte if (grafo.fonte) else 0
    destino = grafo.destino if (grafo.destino) else (grafo.vertices - 1)

    while(True):
        caminho = edmonds_karp(fonte, destino)

        if (not caminho):
            break

        fluxo_caminho = grafo.inf

        for i in range(len(caminho) - 1):
            fluxo_caminho = min(grafo[caminho[i]][caminho[i + 1]], fluxo_caminho)

        for i in range(len(caminho) - 1):
            grafo[caminho[i]][caminho[i + 1]] -= fluxo_caminho

        fluxo += fluxo_caminho

    print(fluxo)
```

As adaptações que foram necessárias adicionar um controlador se o fluxo residual atual era maior que zero, portanto era um caminho possível para ajustar.

```
while (q):
    u = q.pop()
    vizinhos = grafo.vizinhos(u)
    for vizinho in vizinhos:
        if not visitados[vizinho] and grafo[u][vizinho] > 0.0:
            visitados[vizinho] = True
            a[vizinho] = u

            if (vizinho == destino):
                p = [destino]
                w = destino
                while (w != fonte):
                    w = a[w]
                    p.append(w)

                p.reverse()
                return p

            q.insert(0, vizinho)

return []
```

O algoritmo utilizado apresenta um desempenho satisfatório, porém devido sua complexidade demanda um pequeno tempo. Após sua execução o resultado sera por exemplo:

```
x@x-desktop:~/Downloads/INE5413-Grafos/atividade3$ python3 edmonds_karp.py
128.0
x@x-desktop:~/Downloads/INE5413-Grafos/atividade3$
```

Retornando ao fluxo máximo do problema.

3. Hopcroft-Karp

Este algoritmo é mais eficiente do que utilizar algoritmos de Fluxo Máximo, tendo em vista esse aspecto, foram realizadas certas alterações e adaptações de entradas para o algoritmo:

Primeiramente, temos uma função get_sets, utilizando dois vetores s1 e s2, e um vetor T visando obter os vértices que já foram percorridos:

```python
def get_sets(grafo):
    s1 = [0]
    s2 = []
    t = [False] * grafo.vertices
    t[0] = True

    i = 0
    while(False in t):
        vz = grafo.vizinhos(i)
        if (i in s1):
            for v in vz:
                if (not t[v]):
                    s2.append(v)
                    t[v] = True
        elif (i in s2):
            for v in vz:
                if (not t[v]):
                    s1.append(v)
                    t[v] = True

        i = (i + 1) % grafo.vertices
    return s1, s2
```

Em segundo plano, temos uma busca em largura, sendo realizada no grafo Bipartido, com algumas adaptações, no caso, temos dois novos argumentos de entrada adaptados, com os argumentos mate e d, temos um vetor

```python
def BFS(grafo, x, mate, d):
    q = []
    for v in x:
        if (mate[v] == -1):
            d[v] = 0
            q.insert(0, v)
        else:
            d[v] = grafo.inf

    d[-1] = grafo.inf

    while (q):
        v = q.pop()
        if (d[v] < d[-1]):
            vizinhos = grafo.vizinhos(v)
            for vizinho in vizinhos:
                if (d[mate[vizinho]] == grafo.inf):
                    d[mate[vizinho]] = d[v] + 1
                    q.insert(0, mate[vizinho])

    return d[-1] != grafo.inf, d
```

Também temos uma adaptação do algoritmo de busca em profundidade, adaptando tal algoritmo de forma similar a busca em largura:

```
def DFS(grafo, mate, xx, d):
    if xx != -1:
        vizinhos = grafo.vizinhos(xx)
        for y in vizinhos:
            if (d[mate[y]] == d[xx] + 1):
                validez, d, mate2 = DFS(grafo, mate, mate[y], d)
                if (validez):
                    mate = mate2.copy()
                    mate[y] = xx
                    mate[xx] = y
                    return True, d, mate
        d[xx] = grafo.inf
        return False, d, mate
    return True, d, mate
```

Por fim, temos a construção do algoritmo em si, baseado na quantidade de vértices, criamos o vetor de distâncias entre os grafos, sendo este iniciado com valores em ∞, e temos o vetor mate iniciado com valores em -1, tendo o tamanho a quantidade de vértices.

```
def hopcroft_karp(grafo):
    d = [grafo.inf] * (grafo.vertices + 1)
    mate = [-1] * (grafo.vertices)

    x, y = get_sets(grafo)
    m = 0

    while(True):
        validez, d = BFS(grafo, x, mate, d)
        if (not validez):
            break

        for xx in x:
            if (mate[xx] == -1):
                validez, d, mate2 = DFS(grafo, mate, xx, d)
                if (validez):
                    mate = mate2.copy()
                    m += 1
    return m, mate, x
```

Temos a busca em largura sendo realizada com os vértices de cada lado do grafo bipartido, além disso, temos a busca em profundidade sendo realizada, levando em conta o conjunto x.

Por fim temos a amostragem de dados do grafo bipartido:

```
def hopcroft_karp_resultado(resultado):
    m, mate, x = resultado

    print("valor: %d\n" % m)
    print("arestas: ")
    for v in x:
        print("%d - %d;" % (v + 1, mate[v] + 1), end = " ")
    print()
```