Manipulação de Grafos – Criação de biblioteca em *Python*

Lucas Brazzarola de Lima[1]

[1]Instituto de Ciências Exatas e Aplicadas – Universidade Federal de Ouro Preto (UFOP)
Minas Gerais – MG – Brasil

[2]Departamento de Ciências Exatas e Aplicadas
DECEA

[3]Departamento da Computação e Sistemas
DECSI

lucas.brazza.lb@gmail.com

Abstract. *Graph theory studies the relationships between objects in the same set. To facilitate their representation, structures known as graphs are used, G (V, E) in which V is a non-empty set of objects (vertices) and E is a subset of unordered pairs of V. This article aims to create a library to manipulate and perform operations with graphs. Such library will be developed using the Python programming language.*

Resumo. *A teoria dos grafos estuda as relações entre os objetos de um mesmo conjunto. Para facilitar a representação dos mesmos, são utilizadas estruturas conhecidas como grafos, G (V, E) no qual V é um conjunto não vazio de objetos (vértices) e E é um subconjunto de pares não ordenados de V. O presente trabalho tem como objetivo a elaboração de uma biblioteca para manipular e realizar operações com grafos. Tal biblioteca será desenvolvida utilizando a linguagem de programação Python.*

1. Introdução

Utilizando a Teoria dos Grafos podemos representar, entender e resolver diversos problemas reais de diversas maneiras. A representação de um grafo pode ser feita em um arquivo de texto de forma que na primeira linha de tal arquivo tem-se o número de vértices do grafo juntamente com a quantidade de arestas no mesmo; nas linhas subsequentes tem-se o vértice de origem, o vértice destino e o peso das arestas entre eles. Tal é o formato Dimacs.

O presente artigo busca a criação de uma biblioteca onde será possível a manipulação de um grafo a partir de um arquivo no formato Dimacs. Foram desenvolvidos algoritmos para representar tal grafo em forma de uma lista ou matriz de adjacências, a partir das quais é possível obter algumas informações para melhor análise do mesmo. Utilizando a biblioteca é possível encontrar para um grafo seus vértices de maior e menor graus, o grau médio dos vértices e a frequência relativa dos graus. Ademais, é possível analisar os vértices do grafo e seu nível de acordo com uma árvore de busca, seja ela criada a partir de uma busca em largura ou em profundidade, além de se obter a conexidade do grafo.

Ao final a biblioteca será testada para se averiguar a eficiência da mesma. Os testes serão feitos utilizando dois grafos, um representando as conexões das redes que formam a Internet e o outro representando a colaboração entre pesquisadores na criação de artigos científicos. Para cada grafo será analisado o custo de memória, o tempo de execução e a eficácia dos resultados dos algoritmos desenvolvidos.

2. Algoritmos de Manipulação de Grafos

Nesta seção serão apresentados e explicados os pseudocódigos dos algoritmos implementados.

2.1. Algoritmos de Entrada e Representação

A biblioteca desenvolvida consegue representar um grafo ponderado como uma lista de adjacências ou uma matriz de adjacências. Em uma lista de adjacências ponderada é mantida para cada vértice do grafo, uma lista de todos os outros vértices com os quais ele tem uma aresta e o peso da mesma (em forma de tupla). Já em uma matriz de adjacências ponderada é criada uma matriz quadrada onde o tamanho corresponde ao número de vértices no grafo, caso o vértice da linha i não tenha conexão com o vértice da coluna j, a posição a_{ij} será preenchida com o número zero, e caso tenha será preenchida com o peso da aresta correspondente. Para ser possível a utilização da biblioteca, é necessário inicialmente abrir o arquivo de texto e transforma-lo em uma variável. Para tal foi criado um método simples que recebe o nome do arquivo a ser aberto e retorna uma variável do tipo FILE, que é justamente o tipo recebido pelas funções de representação.

Algoritmo 1: Lista de Adjacências

```
    Entrada: arquivo Dimacs(file)
1   file seek início
2   tam ← lê tamanho na primeira linha
3   lista ← [ [ ] para cada n em tam]
4   linha ← lê próxima linha
5   enquanto linha ≠ ∅ faça
6       vértice ← linha[0]
7       lista[vertice] acrescenta (linha[1] , linha[2])
8       vértice ← linha[1]
9       lista[vertice] acrescenta (linha[0] , linha[2])
10      linha ← lê próxima linha
11  retorna lista
```

O Algoritmo 1 é usado para representar um grafo na forma de lista de adjacências. Ele recebe um FILE e na primeira linha reinicia o ponteiro de leitura do arquivo para a primeira posição do mesmo. Nas linhas (2-4) é feita a inicialização necessária para a execução do algoritmo. Nas linhas (6-9) é feita a atribuição da tupla (destino, peso) para os vértices na posição correspondente na lista, note que é necessário atribuir o vértice destino à posição da origem e o vértice destino à posição origem. Na linha (10) o ponteiro de leitura é deslocado em uma posição. Por fim o método retorna a lista gerada.

Algoritmo 2: Matriz de Adjacências

```
    Entrada: arquivo Dimacs (file)
1   file seek início
2   size ← lê tamanho na primeira linha
3   matriz ← [ [0 para cada i em size] para cada n em size]
4   linha ← lê próxima linha
5   enquanto linha ≠ ∅ faça
6       matriz [linha[0]] [linha[1]] ← linha[2]
7       matriz [linha[1]] [linha[0]] ← linha[2]
8       linha ← lê próxima linha
9   retorna matriz
```

O Algoritmo 2 é usado para representar um grafo na forma de lista de adjacências. Igualmente ao anterior, ele recebe um FILE e na primeira linha reinicia o ponteiro de leitura do arquivo para a primeira posição do mesmo. Nas linhas (2-4) é feita a inicialização necessária para a execução do algoritmo, onde é criada uma matriz quadrada de tamanho igual ao número de vértices e com todos os valores sendo zero. Nas linhas (6-8), as posições da matriz (origem-destino/destino-origem) recebem o valor do peso da aresta. Por fim o método retorna a matriz gerada.

2.2. Algoritmo de Maior e Menor Grau e Grau Médio

Na Teoria dos Grafos, por vezes é interessante saber a quantidade de ligações de um vértice, ou seja, qual o grau do vértice. Sabendo disso pode-se analisar melhor a "anatomia" do grafo em questão. Assim sendo foram desenvolvidos algoritmos para encontrar qual o vértice de maior grau, o de menor e a média dos graus.

Algoritmo 3: Maior Grau

```
    Entrada: lista de adjacências (L)
1   grau ← 0
2   vértice ← 0
3   para cada v em L faça
4       compara ← tamanho de L[v]
5       se compara > grau então
6           vértice ← v
7           grau ← compara
8   retorna (grau, vértice)
```

O Algoritmo 4 recebe uma lista de adjacências e nas linhas (1-2) faz as inicializações necessárias para a execução do algoritmo, sendo zero o menor grau possível. Nas linhas (4-7) a lista é percorrida para encontrar qual vértice possui mais adjacência. Ao final é retornada uma tupla com o vértice encontrado e seu grau.

Algoritmo 4: Menor Grau

```
   Entrada: lista de adjacências (L)
1  grau ← tamanho L
2  vértice ← 0
3  para cada v em L faça
4  |    compara ← tamanho de L[v]
5  |    se compara < grau então
6  |    |    vértice ← v
7  |    |    grau ← compara
8  retorna (grau, vértice)
```

O Algoritmo 5 recebe uma lista de adjacências e nas linhas (1-2) faz as inicializações necessárias para a execução do algoritmo, sendo o maior grau possível o número total de vértices. Nas linhas (4-7) a lista é percorrida para encontrar qual vértice possui menos adjacência. Ao final é retornada uma tupla com o vértice encontrado e seu grau.

Algoritmo 5: Grau Médio

```
   Entrada: lista de adjacências (L)
1  tam ← tamanho L
2  média ← 0
3  para cada v em L faça
4  |    média ← média + tamanho de L[v]
5  média ← média / tam
6  retorna média
```

O Algoritmo 5 recebe uma lista de adjacências e nas linhas (1-2) faz as inicializações necessárias para a execução do algoritmo. Nas linhas (3-4) a lista é percorrida e é somado o grau de todos os vértices, esse valor então é dividido pelo tamanho da lista na linha (5), para se obter a média dos graus. Ao final é retornada a média.

2.3. Algoritmo de Distribuição Empírica

A distribuição empírica do grau dos vértices, ou frequência relativa de graus, é a frequência que cada grau aparece no grafo, ou seja, basicamente é a porcentagem de aparição dos graus no grafo. Para calcula-la é divide-se o número de vértices de grau x pelo total de vértices.

Algoritmo 6: Frequência Relativa

```
     Entrada: lista de adjacências (L)
1    tam ← tamanho L
2    lista ← [ ]
3    indice ← 0
4    para cada k em L faça
5    |    qtd ← 0
6    |    para cada i em L faça
7    |    |    temp ← tamanho L[i]
8    |    |    se temp = k então
9    |    |         qtd ← qtd + 1
10   |    frequência ← qtd / tam
11   |    se frequência ≠ 0 então
12   |    |    lista adiciona (k, frequência)
13   |    |    indice ← indice + 1
14   para cada n em (tamanho lista)-1 começando do fim faça
15   |    se lista [n] [1] = [ ] então
16   |    |    lista remove lista[n]
17   retorna lista
```

O Algoritmo 6 recebe uma lista de adjacências e nas linhas (1-3) faz as inicializações necessárias para a execução do algoritmo. Nas linhas (5-13), para cada elemento da lista é testado quantos vértices de determinado grau existem e então dividido pelo total de vértices, caso o valor obtido não seja zero, ele é então acrescentado na lista de graus na forma de uma tupla (grau, frequência). Nas linhas (14-16) são removidos os elementos nulos da lista. Por fim é retornada a lista de frequências relativas.

2.4. Algoritmo de Busca em Largura e em Profundidade

É comum a necessidade de encontrar um vértice específico em um grafo, uma das maneiras de fazer isso é utilizando árvores de busca, onde cada vértice será o nó de uma árvore e seus adjacentes serão os vizinhos do nó. Duas formas de se fazer isso são utilizando a busca em largura e a busca em profundidade. A primeira analisa primeiro os nós mais próximos da origem para depois explorar os mais distantes. Já a segunda explora os vértices o mais profundo o possível de um ramo para depois passar para o próximo ramo. Na biblioteca desenvolvida, escreverão em um arquivo de saída todos os vértices do grafo e seus respectivos níveis na árvore de busca.

Algoritmo 7: Busca em Largura

```
    Entrada: lista de adjacência (G)
    Entrada: origem init
1   visitados ← [init]
2   fila ← [init]
3   nível ← [-1 para cada i em G ]
4   nível [init] ← 0
5   arq ← cria arquivo
6   enquanto fila faça
7       u ← fila retira primeiro elemento
8       para cada v em G[u]
9           k ← v[0]
10          se k ∉ visitados
11              nível [k] ← nível [u] + 1
12              visitados acrescenta k
13              fila acrescenta k
14      arq escreve u , level [u]
15  fecha arq
```

O Algoritmo 7 recebe como parâmetros uma lista de adjacências e um vértice de origem. Nas linhas (1-2) são feitas as iniciações necessárias. Então é inicializado um vetor que armazenará os níveis dos vértices nas linhas (3-4), caso vértice não esteja na árvore de busca seu nível será (-1). A seguir, linhas (7-14), todos os vértices são percorridos e, caso tão vértice ainda não tenha sido descoberto (visitado) o nível é incrementado, o vértice é adicionado na fila e nos visitados e então as informações necessárias são escritas no arquivo de saída que então é fechado (15).

Algoritmo 8: Busca em Profundidade

```
   Entrada: lista de adjacência (G)
   Entrada: origem init
1  visitados ← [0 para cada i em G]
2  visitados [init] ← 1
3  pilha ← [init]
4  R ← [init]
5  nível ← [-1 para cada _ em G ]
6  nível [init] ← 0
7  arq ← cria arquivo
8  arq escreve init, nivel[init]
9  enquanto pilha faça
10     u ← pilha [-1]
11     desempilha ← True
12     para cada v em G[u] faça
13         k ← v[0]
14         se visitados[k] = 0 então
15             desempilha ← False
16             pilha acrescenta k
17             R acrescenta k
18             visitados[k] ← 1
19             nível [k] ← nível [u] + 1
20             arq escreve u , level [u]
21             break
22     se desempilha = True então
23         remove pilha [-1]
24 fecha arq
```

O Algoritmo 8 recebe como parâmetros uma lista de adjacências e um vértice de origem. Nas primeiras linhas, no vetor visitados é atribuído o valor zero para cada vértice da lista, e outras inicializações necessárias. Nas linhas (5-6), o vetor de níveis é criado da mesma forma que no Algoritmo 7. A seguir, linhas (10-23), enquanto existirem elementos na pilha criada anteriormente, é testado para cada vértice adjacente ao ultimo da pilha se ele já foi visitado. Caso não, o nível é incrementado, o vértice é adicionado na pilha e no vetor e nos visitados e então as informações necessárias são escritas no arquivo de saída. Após, caso todos os vértices já tenham sido visitados, o ultimo elemento da pilha é removido (tal controle é feito pela variável desempilha). Ao final o arquivo é fechado e o algoritmo encerrado.

2.5. Algoritmo de Componentes Conexas

Diz-se que um grafo é conexo quando é possível chegar em qualquer vértice x a partir de qualquer vértice y. Quando um grafo não é conexo, é possível separa-lo em subgrafos de componentes conexas, onde a mesma lógica se aplica a aquele subconjunto. Para encontrar as componentes conexas dos grafos, é necessária a realização de uma busca no grafo, marcando as componentes conexas do mesmo.

Algoritmo 9: Busca Profundidade Recursiva

```
    Entrada: Um grafo (G)
    Entrada: vertice origem (init)
    Entrada: vetor visitados
    Entrada: marca
1   visitados [init] ← marca
2   para cada v em G[init] faça
3   |   se visitados[v] = 0 então
4   |   |   BuscaProfundidadeRecursiva( G, v, visitados, marca)
5   retorna visitados
```

No Algoritmo 9 foi desenvolvida uma busca em profundidade recursiva, a qual tem a função de atribuir uma marca aos vértices conexos, linha (1). Tal marca indica quais são as componentes conexas daquele subgrafo. Ao final é retornado o vetor com as marcas. Como entrada a função necessita de um grafo, o vértice origem, o vetor de marcas, a marca a ser atribuída aos conexos.

Algoritmo 10: Componentes Conexas

```
    Entrada: Um grafo (G)
1   visitados ← [ 0 para cada _ em tamanho G ]
2   marca ← 0
3   temp ← [ ]
4   componentes ← [ ]
5   para cada i em tamanho G faça
6   |   se visitados[ i ] = 0 então
7   |   |   marca ← marca + 1
8   |   |   BuscaProfundidadeRecursiva( G, i, visitados, marca)
9   para cada i em tamanho visitados faça
10  |   conexas ← 0
11  |   se visitados[ i ] ∉ temp então
12  |   |   temp acrescenta visitados[ i ]
13  |   |   para cada j em tamanho visitados faça
14  |   |   |   se visitados[ i ] = visitados[ j ] então
15  |   |   |   |   conexas ← conexas + 1
16  |   se conexas ≠ 0 então
17  |   |   componentes acrescenta conexas
18  retorna componentes
```

No Algoritmo 10, nas linhas (5-8), são definidas as marcas que cada componente conexa receberá e então é feita uma busca em profundidade no grafo para atribui-las. Após, nas linhas (9-17), são contabilizadas quantas componentes conexas foram encontradas no grafo e quais são. Então é retornado um vetor, que contém as componentes conexas encontradas, sendo seu tamanho é o número de componentes conexas encontradas.

3. Aplicação Prática

Os algoritmos descritos acima foram testados em relação ao uso de memória e tempo de execução, além da corretude dos mesmos. Para tal, foram usados dos arquivos de texto: as_graph e collaboration_graph. O primeiro, com 32385 vértices, representa as conexões das redes que formam a Internet. O segundo, com 71998 vértices, representa a colaboração entre pesquisadores na criação de artigos científicos. Para cada teste foram obtidas 3 amostras resultantes, e então feita a média aritmética para se obter um valor adequado. Todos os testes foram realizados em um computados com processador Intel(R) Core (TM) i3-6100 CPU @ 3.70GHz e 8.00 GB de memória RAM.

3.1. Grafo de Conexões da Web

3.1.1. Representações em Lista e em Matriz

Tanto em termos de tempo de execução quanto em uso de memória, o esperado era que a representação por matriz de adjacências fosse mais custosa do que por lista de adjacências, pois a primeira se trata de uma matriz quadrática onde todas as "linhas" possuem 32385 elementos, enquanto na outra forma de representação, o mesmo não é garantido, podendo este ser apenas o número máximo de elementos. Tal suposição provou-se verdadeira, como será demonstrado a seguir.

Na Imagem 1 temos o tempo de execução em segundos das três amostras, obtendo uma média aproximada de 0.11977 segundos. Já na Imagem 2 é mostrado o ápice do uso de memória na execução do algoritmo, que foi em torno de 1072 MB.

```
Adjacency List Execution Time (s)

1° Run
0.12497067451477051

2° Run
0.10934853553771973

3° Run
0.125
```

Imagem 1

Nome	Status	11% CPU	44% Memória	0% Disco	0% Rede
Aplicativos (2)					
PyCharm (5)		7,5%	1.071,9 MB	0,1 MB/s	0 Mbps
Python		0%	12,0 MB	0 MB/s	0 Mbps
Python		0%	0,5 MB	0 MB/s	0 Mbps
PyCharm		7,5%	1.053,5 MB	0,1 MB/s	0 Mbps
Host da Janela do Console		0%	5,7 MB	0 MB/s	0 Mbps
Filesystem events processor		0%	0,2 MB	0 MB/s	0 Mbps

Imagem 2

Na Imagem 3 temos os resultados das amostras para a criação da matriz de adjacências, a média obtida para tal abordagem foi 41.86608 segundos. Seguido, na Imagem 4, tem-se o uso de memória em seu máximo: por volta de 5192 MB.

```
Adjacency Matrix Execution Time (s)

1° Run
38.86564636230469

2° Run
38.77192497253418

3° Run
47.96068215370178

Process finished with exit code 0
```

Imagem 3

Nome	Status	46% CPU	92% Memória	44% Disco	0% Rede
Aplicativos (2)					
PyCharm (4)		35,3%	5.191,8 MB	0 MB/s	0 Mbps
Python		35,3%	4.481,1 MB	0 MB/s	0 Mbps
Python		0%	0,4 MB	0 MB/s	0 Mbps
PyCharm		0%	704,6 MB	0 MB/s	0 Mbps
Host da Janela do Console		0%	5,7 MB	0 MB/s	0 Mbps

Imagem 4

Como visto, a representação matricial é muito mais custosa, chegando a ter um tempo de execução aproximadamente 40 vezes maior e uma utilização de memória quase 5 vezes maior do que a necessária para a representação em lista.

3.1.2. Distribuição Empírica do Grau dos Vértices e Componentes Conexas

No gráfico a seguir, é exposta a distribuição empírica do grau dos vértices, que é a frequência de cada grau no grafo. Nota-se que há uma predominância de vértices de menor grau, enquanto existem poucos vértices com muitas arestas. Utilizando os métodos da biblioteca, foram obtidos também: o vértice de maior grau é o 1, com grau igual a 2161; o vértice de menor grau é o 0, de grau 1; a média aritmética dos graus é 2.89165 (valor aproximado). Nota-se que o maior grau possível que poderia haver em tal grafo é o próprio número de vértices (32385), mas tal não ocorre.

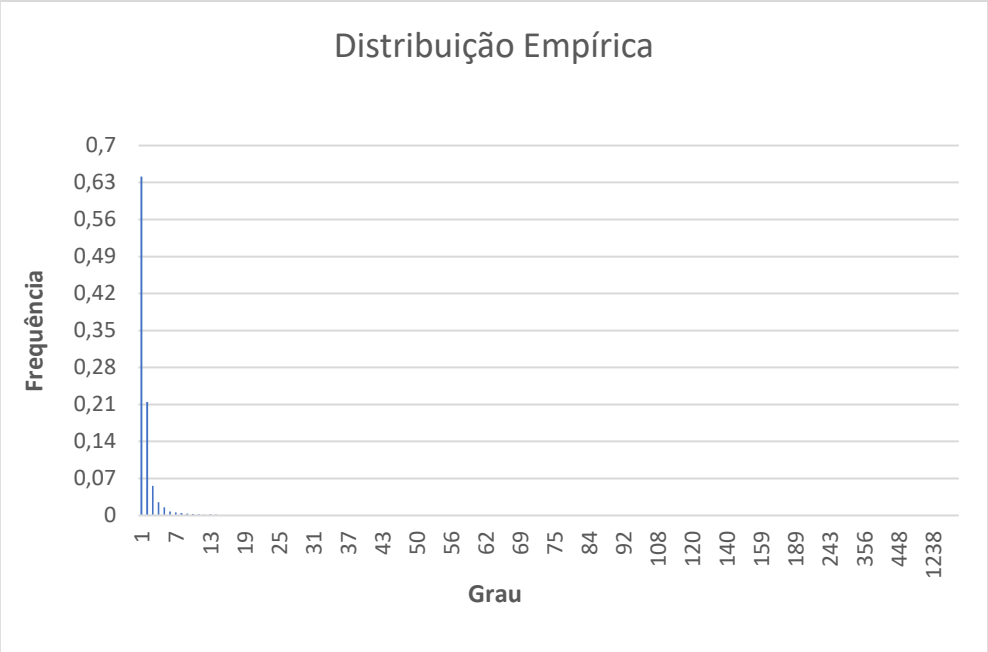


Gráfico 1

Além disso, também é possível reparar que, segundo o algoritmo desenvolvido, não há componentes desconexas. Ou seja, é possível chegar a qualquer vértice x partindo de qualquer vértice y, desde que x e y estejam contidos no grafo. Tal conclusão foi obtida através do retorno do algoritmo, que mostra um vetor de n números que contém a quantidade de vértices em cada componente conexa, sendo n o numero de componentes conexas. Como retorno foi obtido um vetor de tamanho 1 contendo o valor 32385.

3.1.3. Busca em Largura e em Profundidade

Foi medido o tempo de execução de ambos os algoritmos e o que se sobressaiu foi que o algoritmo de busca em profundidade, teve uma média de 0.64074 segundos (feita a partir dos dados da Imagem 5), enquanto o outro demorou em média 12.45765 segundos para ser finalizado (valor obtido a partir dos dados da Imagem 6).

```
Breadth-First Search Execution Time (s)

1° Run
12.450203657150269

2° Run
12.465792655944824

3° Run
12.456966161727905
```

Imagem 5

```
Depth-First Search Execution Time (s)

1° Run
0.6092321872711182

2° Run
0.6560966968536377

3° Run
0.6560964584350586
```

Imagem 6

Nota-se uma clara diferença entre as execuções, porém também há uma grande diferença entre a distância máxima entre dois vértices: rodando os algoritmos partindo do vértice 0, o número máximo de passos obtido foi 6 para a busca em largura, enquanto para a busca em profundidade foi 970 (de 0 a 29966). O mesmo ocorre para outros valores testados, como mostrado na Tabela 1.

Vértice	**Maior Nível de Busca**	
Origem	**Largura**	**Profundidade**
0	6	970
16	7	978
103	7	997
2158	8	924
22856	8	979
30000	8	912

Tabela 1

Utilizando tais dados então podemos determinar que o diâmetro de tal grafo (maior distancia entre dois vértices), a partir da busca em largura é 8 e a partir da busca em profundidade é 997.

3.2. Grafo de Colaboração em Pesquisa

3.2.1. Representações em Lista e em Matriz

A média aritmética do tempo de execução do algoritmo de representação em lista para o grafo em questão foi de aproximadamente 0.28635 segundos (dados tirados da Imagem 7) com um uso máximo de memória de aproximadamente 1125 MB (como visto na imagem 8) – Tudo isso partindo do vértice 1.

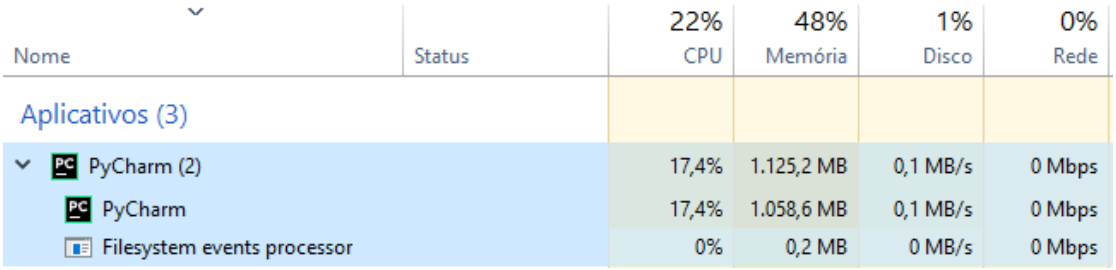
```
Adjacency List Execution Time (s)

1° Run
0.28108954429626465

2° Run
0.29683637619018555

3° Run
0.28115010261535645
```

Imagem 7



Nome	Status	22% CPU	48% Memória	1% Disco	0% Rede
Aplicativos (3)					
PyCharm (2)		17,4%	1.125,2 MB	0,1 MB/s	0 Mbps
PyCharm		17,4%	1.058,6 MB	0,1 MB/s	0 Mbps
Filesystem events processor		0%	0,2 MB	0 MB/s	0 Mbps

Imagem 8

Durante a execução do algoritmo para a conversão do arquivo em matriz de adjacências, houveram complicações: o ápice do uso de memória foi maior que 6800 MB, quando tal nível foi atingido o computador que estava sendo usado para teste travou, tornando impossível o registro de qualquer valor, fazendo-se necessária uma reiniciação forçada do mesmo. Tal acontecimento se repetiu em todos os testes, assim sendo não há como serem feitas comparações sobre o desempenho para com o outro algoritmo.

3.2.2. Distribuição Empírica do Grau dos Vértices e Componentes Conexas

No Gráfico 2, é mostrada a distribuição empírica do grau dos vértices. Nota-se uma predominância de vértices de grau 2. Utilizando os métodos da biblioteca, foram também obtidos: o vértice de maior grau é o 72, com grau igual a 3691; o vértice de menor grau é o 0, de grau 31; a média aritmética dos graus é 3.42728 (valor aproximado). Nota-se que o maior grau possível que poderia haver em tal grafo é o próprio número de vértices (71998), mas isso não ocorre.

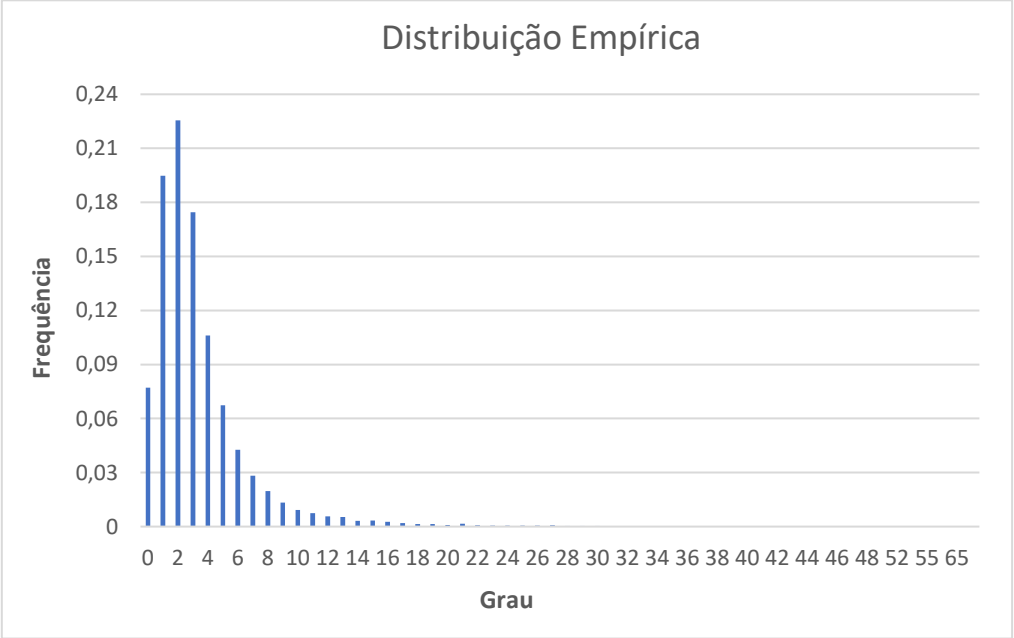


Gráfico 2

Novamente houveram problemas na execução dos testes: durante a execução do algoritmo que encontra as componentes conexas foi obtido um erro de recursão, dizendo que o número máximo de recursões permitidas pela IDE Pycharm foi alcançado (tal número é 1000 recursões). Na tentativa de contornar tal erro a IDE foi configurada para permitir até 5000 recursões, porém o problema persistiu. Assim, foi considerado arriscado aumentar ainda mais o número máximo de recursões e os testes foram cancelados. Seguem abaixo o *stack tracing* do erro.

```
Traceback (most recent call last):
  File "C:/Users/Lucas/PycharmProjects/Grafos/TrabalhoGrafos/main.py", line 69, in <module>
    relatedComponentsR = BasicDimacs.RelatedComponentsR(adjList)
  File "C:\Users\Lucas\PycharmProjects\Grafos\TrabalhoGrafos\GraphManipulation\BasicDimacs.py", line 238, in RelatedComponentsR
    DepthFirstSearchRecursive(G, node, visited, mark)
  File "C:\Users\Lucas\PycharmProjects\Grafos\TrabalhoGrafos\GraphManipulation\BasicDimacs.py", line 204, in DepthFirstSearchRecursive
    DepthFirstSearchRecursive(Graph, nodeV, visited, mark)
  File "C:\Users\Lucas\PycharmProjects\Grafos\TrabalhoGrafos\GraphManipulation\BasicDimacs.py", line 204, in DepthFirstSearchRecursive
    DepthFirstSearchRecursive(Graph, nodeV, visited, mark)
  File "C:\Users\Lucas\PycharmProjects\Grafos\TrabalhoGrafos\GraphManipulation\BasicDimacs.py", line 204, in DepthFirstSearchRecursive
    DepthFirstSearchRecursive(Graph, nodeV, visited, mark)
  [Previous line repeated 994 more times]
  File "C:\Users\Lucas\PycharmProjects\Grafos\TrabalhoGrafos\GraphManipulation\BasicDimacs.py", line 203, in DepthFirstSearchRecursive
    if visited[nodeV] == 0:
RecursionError: maximum recursion depth exceeded in comparison

Process finished with exit code 1
```

Imagem 9

3.2.3. Busca em Largura e em Profundidade

Ao executar os algoritmos de busca em largura e busca em profundidade para o atual grafo, nota-se a mesma ocorrência que foi apontada anteriormente durante a analise do Grafo de Conexões da Web: a busca em largura leva mais tempo do que a busca em profundidade, sendo suas respectivas médias aproximadas de 70,22913 e 0,47384 segundos. Segue abaixo as amostras dos testes.

```
Breadth-First Search Execution Time (s)

1° Run
71.10177683830261

2° Run
68.94104027748108

3° Run
70.64459419250488
```

Imagem 10

```
Depth-First Search Execution Time (s)

1° Run
0.43743062019348145

2° Run
0.42174649238586426

3° Run
0.562363862991333
```

Imagem 11

4. Conclusões

Não há dúvidas da importância dos grafos para a resolução de problemas complexos, mas a manipulação dos mesmos pode ser um pouco complicada. Como foi notado, a biblioteca desenvolvida provou-se limitada no manejo de grafos muito grandes. Possivelmente sob outras condições de teste, tendo disponível mais memória, os resultados sejam diferentes.

No mais, levando em conta os processos que não houveram qualquer problema, nota-se a clara vantagem do uso do algoritmo de busca em profundidade tratando em tempo de execução. Caso seja intencionado que a arvore de busca seja mais balanceada e que o menor uso de memória seja priorizado, então o algoritmo de busca em largura torna-se a melhor opção. Quanto aos grafos utilizados nos testes, faz-se mais presente as poucas conexões, indicando poucas conexões diretas entre os elementos.

Além dos algoritmos citados, a biblioteca inclui métodos com as mesmas funções das apresentadas, porém que utilizam uma matriz de adjacências para realizar o processamento. Foi escolhida a utilização da lista de adjacência, pois os métodos desenvolvidos com essa são menos complexos e requerem um custo computacional menor. Ademais, os algoritmos de busca possuem uma função a mais: os arquivos de saída são ordenados de forma que os vértices são apresentados em ordem crescente. Tal ordenação é feita utilizando um algoritmo Quick Sort Iterativo, também presente na biblioteca. Todos os códigos estão presentes em um repositório no Github.

5. Referencias

LIMA, L. B. (2021). Manipulação de grafos. https://github.com/LucasBrazza/GraphManipulation

FONSECA, G. H. G. (2020). A03: Representação computacional. Disponível em https://www.moodlepresencial.ufop.br/pluginfile.php/857799/mod_resource/content/0/A03%20Representação%20Computacional.pdf, acessado em Julho de 2021.

FONSECA, G. H. G. (2020). A04: Percursos e conectividade. Disponível em https://www.moodlepresencial.ufop.br/pluginfile.php/867246/mod_resource/content/0/A04%20Percursos%20e%20Conectividade.pdf, acessado em Julho de 2021.

FONSECA, G. H. G. (2020). A04: Percursos e conectividade. Disponível em https://www.moodlepresencial.ufop.br/pluginfile.php/867247/mod_resource/content/0/A08%20Busca%20em%20Largura%20e%20em%20Profundidade.pdf, acessado em Julho de 2021.