Biblioteca de Grafos - Aplicação dos conceitos de grafos em *Python*

Ayron Luigi de Paiva[1], Kevin Lucas de Oliveira Brito[2]

[1]Instituto de Ciências Exatas e Aplicadas – Universidade Federal de Ouro Preto (UFOP)
João Monlevade – MG – Brazil

[2]Departamento de Ciências Exatas e Aplicadas
DECEA

[3]Departamento Computação e Sistemas
DECSI

ayronpaiva@outlook.com

Abstract. *This work aims to enhence the skills as well as the knowledge of students, regarding to the concepts and definitions studied in Graph Theory. Additionally, to develop this proposal, we will work on practical applications, facilitating a better visualization of the use of this course in the daily activities. The activity proposes implementations of algorithms based on computational representations, search methodologies, routes and connectivity, and certain usual problems related to the content of Graph Theory.*

Resumo. *Esse trabalho objetiva o aprofundamento das habilidades bem como dos conhecimentos dos alunos, em relação aos conceitos e definições estudadas em Teoria dos Grafos. Além disso, para seu desenvolvimento, iremos trabalhar com aplicações práticas, facilitando uma melhor visualização do uso dessa disciplina em atividades do cotidiano. A atividade propõe implementações de algoritmos baseados em representações computacionais, metodologias de busca, percursos e conectividade, e determinados problemas usuais relacionados ao conteúdo de Teoria dos Grafos.*

1. Introdução

A Teoria dos Grafos é uma disciplina fundamental dentro da área de informática e computação, visto que os grafos são uma ferramenta matemática com diversas aplicações, incluindo representações de problemas já existentes bem como suas soluções. Um grafo é um conjunto de pontos, chamados vértices (ou nodos ou nós), conectados por linhas, chamadas de arestas (ou arcos), essas arestas podem ou não ter valores como podem ou não ter sentido, no caso desse trabalho foram escolhidos grafos não-orientados ponderados, onde as arestas tem valores mas não tem sentido.

O objetivo desse trabalho é desenvolver a habilidade de programação de algoritmos em grafos, em linguagem python, reforçando os aprendizados desenvolvidos em aula, avaliaremos qual representação de um grafo(matriz de adjacências ou lista de adjacências) possuem o melhor desempenho computacional tanto por quantidade de memoria quanto por tempo de execução, testamos também buscas(profundidade e largura) nesses grafos, além da procura do menor, maior e medio grau e a distribuição empírica do mesmo, conceitos de complexidade e optimização de algoritmos foram utilizados para a implemetação dos algoritmos. Todo o código referente ao assunto discorrido neste artigo pode ser visualizado através de um repositório do GitHub [PAIVA 2021].

2. Descrição

2.1. Entrada

Os algoritmos foram implementados em linguagem Python, utilizando da IDE *Pycharm Community edition 2021.1.1*, os grafos foram disponibilizados em arquivos **txt** no formato Dimacs onde temos no cabeçalho o número de vértices e arestas e nas linhas subsequentes temos cada aresta no formato origem destino peso.

Figura 1. Pseudocódigo para abrir e fechar o arquivo

```
Algoritmo 1: abrirArquivo{
Entrada:(i)Nome do arquivo no formato String(NomeArq)
Saída:Arquivo aberto
        arquivo.open(NomeArq,'w')          ->Abri o arquivo para escrita

        Return arquivo
    }
}


Algoritmo 2: fecharArquivo{
Entrada:(i)Arquivo já aberto(Arq)

        Arq.close()                 ->Fecha o arquivo


}
```

2.2. Representação

Uma das maiores dificuldades em tratar problemas como grafos está na representação desse problema como um próprio grafo. Nesse raciocínio, a melhor maneira de tratar esses tipos de problemas seria fazendo uso de um computador. Isso nos leva à necessidade de uma representação computacional para os grafos.

Dessa forma iniciamos nosso trabalho, buscando uma representação para os grafos oferecidos nas instruções do trabalho. Após a abertura do arquivo desejado pelo usuário, ambas representações estudadas em aula, lista e matriz de adjacência, se aplicam aos algoritmos desenvolvidos. Em uma lista de adjacências, para cada vértice é armazenada uma lista dos vértices que estão ligados a ele (adjacentes), como utilizamos grafos orientados, a solução mais viável foi criar um par (destino,peso) para cada posição da lista de adjacências. Enquanto matriz de adjacências é uma estrutura onde cada vértice corresponde a uma linha e a uma coluna da matriz, logo, temos uma matriz quadrada $A = [a_i j]$ de ordem $|V| \times |V|$. A lista de adjacência efetua um melhor desempenho no funcionamento do programa, dado que sua eficiência em armazenamento é uma grande vantagem, com apenas $\theta(|V| + |E|)$ posições de memória, enquando as matrízes de ajacência em armazenamento custam $\theta(|V^2|)$ posições de memória.

Figura 2. Pseudocódigo para converter em matriz de adjacências

```
Algoritmo 3:converte_matriz1{
Entrada:(i)Arquivo já aberto(Arq)
Saída:Matriz de adjacência


        arq.seek(0)                                                     ->Aponta pro início do arquivo Byte 0
        tamanhoArq<-primeiro número antes do espaço da primeira linha
        matriz<- matriz vazia de tamanhoArq linhas e colunas

        linha<-segunda linha

        enquanto linha != '' faça

                matriz[primeiro numero antes do espaço de linha][segundo numero antes do espaço de linha]<-terceiro numero antes do espaço de linha
                matriz[segundo numero antes do espaço de linha][primeiro numero antes do espaço de linha]<-terceiro numero antes do espaço de linha

                linha<-próxima linha

        Return matriz
}
```

Figura 3. Pseudocódigo para converter em lista de adjacências

```
Algoritmo 4:converteLista{
Entrada:(i)Arquivo já aberto(Arq)
Saída:Lista de adjacências

    arq.seek(0)                               ->Aponta pro início do arquivo Byte 0

    linha1<-primeiro número antes do espaço da primeira linha
    linha<- segunda linha

    lista<-cria uma lista vazia com tamanho do primeiro numero antes do espaço

    enquanto linha != ''  faça
            el1<-primeiro numero antes do espaço de linha
            el2<-segundo numero antes do espaço de linha
            el3<-terceiro numero antes do espaço de linha
            lista[el1]<-(el2,el3)                    ->Adiciona os elementos na lista
            lista[el2]<-(el1,el3)

   Return lista



}
```

2.3. Informações

O estudo em Teoria dos Grafos, abriga uma enorme variedade de conceitos, como visto até agora, entre esses temos o grau, também chamdo de valência, de um vértice de um grafo corresponde ao número de arestas incidentes para com o vértice, isto é, o número de arestas adjacentes à ele. O grau máximo de um grafo e o grau mínimo, equivalem aos graus máximo e mínimo de seus vértices. Quando se trata de um grafo regular, todos os graus são iguais.

Outro importante conceito utilizado na implementação da atividade é a frequência relativa. A frequência relativa, é um conceito complementar para os graus de um grafo. Chamamos a distribuição dos graus de um grafo, de frequência relativa de determinado grau. O somatório de todos os valores de frequência relativa dos graus de um grafo devem ser igual à 1. Exemplo das informações de um grafo de 5 arestas:

Maior Grau: 2 - vértice: 0 Menor Grau: 1 - vértice: 2
Grau Médio: 1.6
Frequência Realativa:
Grau 1: 0.40
Grau 2: 0.60

Figura 4. Pseudocodigo grau maior

```
Algoritmo 5:grauMaior{
Entrada:(i)Arquivo já aberto(arq) (ii)Lista ou Matriz(listOrMatriz)
Saída:(i)O maior grau;(ii) vertice


    maiorGrau<-0
    vertice<-0

    se listOrMatriz==1
        listAux<-converteLista(arq)                  ->cria uma lista de adjacências do arquivo
            para cada i adjacente ao tamanho da lista
                se maiorGrau < tamanho da lista[i]
                        vertice<-i
                        maiorGrau<-tamanho da lista[i]

    se listOrMatriz==2
        matrizAux<-converte_matriz1(arq)             ->cria uma matriz de adjacêncas do arquivo
        para cada i adjacente ao tamanho da matriz
            aux<-[]                                  ->cria um vetor auxiliar
                para cada j adjacente ao tamanho da matriz
                    se matrizAux[i][j] !=0
                        aux<-matrizAux[i][j]
                se maiorGrau<tamanho de aux
                        vertice<-i
                        maiorGrau<-tamanho de aux

    Return maiorGrau, vertice


}
```

Figura 5. Pseudocodigo grau menor

```
Algoritmo 6:grauMenor{
Entrada:(i)Arquivo já aberto(arq) (ii)Lista ou Matriz(listOrMatriz)
Saída:(i)O maior menor;(ii) vertice

    listAux<-converteLista(arq)                  ->cria uma lista de adjacências do arquivo
    grauMenor<-tamanho ListaAux+1                -> número maior que qualquer grau
    vertice<-0

    se listOrMatriz==1

            para cada i adjacente ao tamanho da lista
                se grauMenor > tamanho da lista[i]
                        vertice<-i
                        grauMenor<-tamanho da lista[i]

    se listOrMatriz==2
        matrizAux<-converte_matriz1(arq)             ->cria uma matriz de adjacêncas do arquivo
        para cada i adjacente ao tamanho da matriz
            aux<-[]                                  ->cria um vetor auxiliar
                para cada j adjacente ao tamanho da matriz
                    se matrizAux[i][j] !=0
                        aux<-matrizAux[i][j]
                se grauMenor>tamanho de aux
                        vertice<-i
                        grauMenor<-tamanho de aux

    Return grauMenor, vertice


}
```

Figura 6. Pseudocodigo grau medio

```
Algoritmo 7:grauMedio{
Entrada:(i)Arquivo já aberto(arq) (ii)Lista ou Matriz(listOrMatriz)
Saída:(i)O grau medio


    medioGrau<-0


    se listOrMatriz==1
        listAux<-converteLista(arq)                 ->cria uma lista de adjacências do arquivo
        para cada i adjacente ao tamanho da lista
            medioGrau<-medioGrau+tamanho da lista

        medioGrau<-medioGrau/tamanho da lista

    se listOrMatriz==2
         matrizAux<-converte_matriz1(arq)            ->cria uma matriz de adjacêncas do arquivo
        para cada i adjacente ao tamanho da matriz
            aux<-[]                                 ->cria um vetor auxiliar
                para cada j adjacente ao tamanho da matriz
                        se matriAux[i][j]!=0
                            aux<-matrizAux[i][j]
                medioGrau<-medioGrau/tamanho da matriz

    Return medioGrau

}
```

Figura 7. Pseudocodigo frequência Relativa

```
Algoritmo 8:frequenciaRelativa{
Entrada:(i)Arquivo já aberto(arq) (ii)Lista ou Matriz(listOrMatriz)
Saída:(i)Lista de frequência Relativa composto por tuplas (grau, frequência Relativa)

    lista<-[]

    se listOrMatriz==1
        listAux<-converteLista(arq)                 ->cria uma lista de adjacências do arquivo
        para cada i adjacente ao tamanho da listaAux
            somaGraus<-0
            para cara j adjacente ao tamanho da listaAux
                se i == tamnho de listaAux[j]
                    somaGraus<-somaGraus+1
                lista<-(i,somaGraus/tamanho da listaAux)
        para cada chave adjacente ao tamanho da lista -1,-1,-1
                se lista[chave][i]== 0.0
                    lista.pop(chave)                          ->remove o elemento

    se listOrMatriz==2
        aux<-[]
        matrizAux<-converte_matriz1(arq)            ->cria uma matriz de adjacêncas do arquivo
        para cada i adjacente ao tamanho da matriz
            somaGraus<-0
            para cada j adjacente ao tamanho da matriz
                se matriAux[i][j]!=0
                    somaGraus<-somaGraus+1
            aux->somaGraus

        para cada i adjacente ao tamanho da matriz
            lista<-(i,somaGraus/tamanho da listaAux)
        para cada chave adjacente ao tamanho da lista -1,-1,-1
                se lista[chave][i]== 0.0
                    lista.pop(chave)                          ->remove o elemento

    Return Lista

}
```

2.4. Busca em Grafos: Largura e Profundidade

No ramo da inteligência artificial, a procura por uma solução específica de um determinado problema é uma grande dificuldade. Entretanto, como estamos nos dedicanto ao grafos, e se tratando de uma busca cega. na qual o processo é feito sistematicamente por todo grafo. Iremos desenvolver dois métodos famosos nesse área, os métodos de busca em profundidade e em largura.

A busca em largura, também chamada de busca em amplitude (Breadth-First Search - BFS, do inglês), é uma estratégia de busca cega onde o método prioriza os vértices mais próximos ao vértice origem, para depois explorar os vértices mais distantes. Esse tipo de busca é caracterizada como uma busca não informada, ou desinformada, que expande e examina todos os vértices de um grafo direcionado ou não-direcionado. A complexidade do pior caso desse algoritmo é da ordem de $O(|V| + |E|)$.

Figura 8. Pseudocódigo para a busca em largura

```
Algoritmo 9:buscaLargura{
Entrada:(i)Um grafo G=(V,E);(ii)Vétice origem s
Saída:(i)Escreve no arquivo a busca
elemento a elemento(ultimo vertice ate encontrar, quantos vertices percorreram);

    tuplaInicial<-(num,0)
    visitados<-[]                   ->inicia a lista de visitados
    queue<-[]                       ->inicia a pilha

    visitados<-tuplaInicial[0]      ->adiciona a origem na lista de visitados
    queue<-tuplaInicial[0]          ->adiciona a origem na pilha


    file<-open('NomeDoArquivo.txt','w')  ->cria o arquivo onde inserir a busca
    indice<-vetor vazio do tamanho do grafo
    indice[tuplaInicial[0]]<-0            ->inicializa a primeira posição

    enquanto tamanho da pilha > 0
        u<-remova o primeiro elemento da pilha
        para cada v adjacente ao grafo[u]
            nodeV<-v[0]

            se nodeV não estiver em visitados
                visitados<-nodeV
                indice[nodeV]<-indice[u]+1
                queie<-nodeV
        escreve no arquivo(u:indice[u])

}
```

A busca em profundidade, ou busca em profundidade-primeiro (Depth-First Search - DFS, em inglês) é uma estratégia de busca cega que explora os vértices mais profundos possíveis em um grafo primeiro.

Em outras palavras, esse algoritmo realiza uma busca não informada, que dá procedência a partir do primeiro nó filho do grafo, e se aprofunda cada vez mais, até que o objetivo seja encontrado ou até que ele se depare com um nó, que não possui mais filhos, retornando assim ao ínicio e progredindo para o próximo nó. Esse efeito é também chamado de backtrack. A complexidade de pior caso desse algoritmo é da ordem de $O(|V| + |E|)$, a mesma da busca e largura.

Quanto a eficiência de uma programa na resolução de um problema, por meio desses algoritmos, a preferência varia em termos do tipo de problema em questão. Existem problemas em que os nós de interesse são as folhas de uma árvore por exemplo, outros os nós filhos.

Figura 9. Pseudocódigo busca em profundidade

```
Algoritmo 10:Busca Profundidade{
Entrada:(i)Um grafo G=(V,E);(ii)Vértice origem s(num)
Saída:(i)Escreve no arquivo a busca elemento a elemento

    desc<-[0 para cada i adjacente ao tamanho do Grafo]  ->Marque todos os vértices como não-descobertos
    S<-[num]                                            ->Adicione s à fila S
    R<-[num]                                            ->Adicione s ao vetor de vértices descobertos R
    desc[num]<-1                                        ->Marca s como descoberto

    indice<-[None para cada i adjacente ao tamanho do Grafo]
    indice[num]<-0

    arquivo<-open('NomeDoArquivo.txt','w')              ->cria o arquivo onde inserir a busca
    desc[num]<-1
    enquanto o tamanho de S for != 0
        u<-S[-1]
        desempilhar<-True
        para cada v adjacente a G[u]

            se desc[v[0]]==0
            indice[v[0]]=indice[u]+1
            desempilhar<-False
            adiciona v[0] ao final de S
            adiciona v[0] ao final de R
            marca desc[v[0]] como descoberto
            escreve no arquivo(v[0]:indice[v[0]])
            break
        se não
            desempilhe u de S
```

2.5. Componentes Conexos

Sabemos que a computação possui uma gama de assuntos e tópicos a serem estudados, em Teoria dos Grafos, dentre seus conceitos básicos temos também algumas características quanto aos grafos, no que diz respeito à sua conectividade. Um grafo é dito conexo se todos os seus pares de vétices estão ligados por um caminho.

No decorrer dos códigos, precisamos trabalhar e entender as definições e aplicações de componentes conexas. Quando um grafo não é conexo, podemos particiona-lo em componentes conexos, ou seja, um subgrafo conexo de um grafo não orientado, no qual não é possível adicionar vértices sem perder a conexidade.

Figura 10. Pseudocódigo Componentes Conexos

```
Algoritmo 11:descobreComponentes conexos{
Entrada:(i)Um grafo(G)

    comp<-o para cada i adjacente ao tamanho de G
    chama a função compoentes_conexas(G,comp)
    maior<-0

    para cada i adjacente a comp
        se comp[i]>maior
        maior->comp[i]
    imprime(f"componentes conexas: {maior}")
    para cada i adjacente  a comp
        contador<-0
        para j adjacente a comp
            se comp[j]==i
                contador=contador+1
        se contador!= 0
        print(f"-{contador}vertices")


}
Algoritmo 12:componentes_conexa{
Entrada:(i)Um grafo(G),(ii)um vetor Complementar
Saída:(i)Vetor complementar
    marca->0

    para cada u adjacente ao tamanho de G
        se comp[u]==0
            marca<-marca+1
            chama a função busca_profundidade_rec(G,u,marca,comp)
    Return comp

}
Algoritmo 13: busca_profundidade_rec{
Entrada:(i)UM grafo,(ii)A origem(s),(iii)Uma marca,(iiii)um vetor Complementar
    comp[s]<-marca
    para cada v adjacente a G[s]
        y<-v[0]
        if comp[y]==0
            chama a função busca_profundidade_rec(G,u,marca,comp)

}
```

3. Resultados

O *Hardware* utilizado para esse trabalho possui a seguinte configuração:

- Processador: Intel(R) Core(TM) i5-4460 CPU @ 3.20GHz 3.20 GHz
- Memória Ram: 8,00 GB DDR5
- Sistema Operacional: Windows 10 Pro (64Bits)
- Placa De vídeo: AMD Radeon RX 270 series

Os tempos foram capturados utilizando a função *time()* da biblioteca *time*, que captura o tempo de clock inicial do processador e o tempo final após a utilização da função e realizando a subtração do tempo final pelo tempo inicial conseguimos obter o tempo do algoritmo, já os gastos de memória foram capturados pelo gerenciador de tarefas do windows.

3.1. Grafo de Colaborações em Pequisa

O grafo do arquivo collaboration graph.txt é composto por um vértice para cada pesquisador e arestas caso já tenham publicado artigos científicos juntos 2 . Utilizando esse grafo e a biblioteca desenvolvida:

- Após a comparação do desempenho, em termos de quantidade de memória, para a representação em lista e em matriz de adjacência, constatamos que as quantidades de memória utilizadas foram 45,0Mb e 4433,7MB respectivamente, concluindo que a matriz de adjacência utiliza muito mais memória que a lista.
- Visto que só conseguimos rodar o algoritmo representado como uma lista de adjacência, compararamos suas buscas, em profundidade levou 0.28603172302246094 segundos, em largura 79.75852036476135 segundos, concluindo que a busca em largura demandou exponencialmente mais tempo que a busca em profundidade.
- No processamento desse grafo, fornecido pelo professor, tivemos respectivamente os graus 0 e 72, como menor e maior grau, o maior grau possivel seria 71997, visto que o grafo posssui 71998 arestas sendo aproximadamente 1000 vezes maior que o maior grau encontrado.

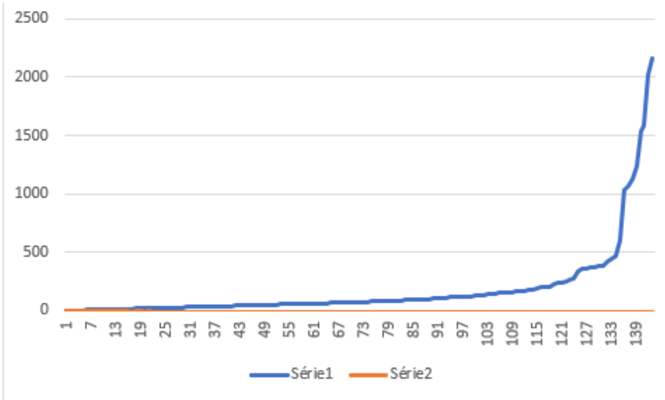
- Devido a limitação da Recursão na IDE utilizada não conseguimos efetuar corretamente, a parte da atividade que corresponde essa questão, segue o erro apresentado

```
RecursionError: maximum recursion depth exceeded while calling a Python object
```

3.2. Grafo de Conexões da Web

O grafo do arquivo as *graph.txt* contém as conexões das redes que formam a Internet . Utilizando esse grafo e a biblioteca desenvolvida:

- O menor grau do grafo é 1 e o maior 2161, o maior grau possivel seria 32384, visto que o grafo posssui 32385 arestas sendo aproximadamente 15 vezes maior que o maior grau encontrado.



- O grafo possui somente um componente conexo com 32385 vertices.
- visto que o grafo possui 32385 e todos os vertices são conexos seu maior e menor componente conexo tem como tamanho 32385.
- A maior distância do vertice 0 é 4, a maior distância do vertice 1 é 1, a maior distância do vertice 2 é 1, a maior distância do vertice 500 é 2 e a maior distância de 32384 é 4, podemos então concluir que nesse grafo em relação a maior distância de um vertice está relacionado com o valor do vertice, crescendo sua maior distância de acordo com o crescimento do valor do vertice.
- O diametro da internet terá como tamanho 4, visto que é a maior distância do grafo e a menor é 1.

4. Conclusão

No decorrer da elaboração do trabalho de grafos, implementamos algoritmos, com diversas aplicações, cuja intenção seria compor uma bilioteca em linguagem de programação python. Durante, o desenvolvimento podemos compreender e aplicar, e também nos aprofundar no estudo dos conceitos e exemplos práticos da relevância da disciplina Teoria dos Grafos.

Na maioria das atividades realizadas, a representação do grafo por meio de matriz de adjacências se mostrou extremamente custosa, para a máquina, tendo altos tempos de execução e alto gasto de memória, por se tratar de sua complexidade $O(n^2)$ sendo utilizado muitas vezes só para exemplos didáticos, em grafos com muitos vertices como o *collaboration graph* seria ineficiente sua utilização, tendo como solução a sua utilização por lista de adjacências onde no seu melhor caso sua complexidade se dá por $O(V)$ tendo um gasto de memória esponencialmente menor que a matriz.

5. Referências

Material de apoio disponibilizado no moodle pelo professor George Henrique Godim da Fonseca

J.A. Bondy and U.S.R. Murty. Graph Theory with Applications. Macmillan/Elsevier, 1976. Internet: http://www.ecp6.jussieu.fr/pageperso/bondy/books/gtwa/gtwa.html

BOAVENTURA NETTO, P. O. Grafos: Teoria, Modelos, Algoritmos. 2 ed, Edgard Blücher (1996)

ALMOULOUD, Saddo Ag. Fundamentos da didática da matemática. Curitiba: Ed. UFPR, 2007.

Artigo Complexidade de Algoritmos 1, https://www.ic.unicamp.br/ zanoni/teaching/mo417/2011-2s/aulas/handout/10-grafos-buscas.pdf (2011)