TAG 2023.2 - Projeto 2
Vitor Guedes Frade - 221017130

Descrição do problema e especificações para a solução:

Considere para efeito deste projeto que uma determinada universidade oferece anualmente uma lista de cinquenta (50) projetos financiados e abertos a participação de alunos. Cada projeto é orientado e gerenciado por professores que estabelecem as quantidades mínima e máxima de alunos que podem ser aceitos em determinado projeto, bem como requisitos de histórico e tempo disponível que os alunos devem possuir para serem aceitos. Esses requisitos de histórico e tempo são pré-avaliados e cada aluno possui uma Nota agregada inteira de [3, 4, 5], sendo 3 indicando suficiente, 4 muito boa, e 5 excelente. Neste ano duzentos (200) alunos se candidataram aos projetos. O ideal é que o máximo de projetos sejam realizados, mas somente se o máximo de alunos qualificados tenham tido o interesse para tal. Para uma seleção impessoal e competitiva um algoritmo que realize um **emparelhamento estável máximo** deve ser implementado. Este projeto pede a elaboração, implementação e testes com a solução final de emparelhamento máximo estável para os dados fornecidos. Os alunos podem indicar no máximo três (3) preferências em ordem dos projetos. Variações do algoritmo Gale-Shapley devem ser usadas, com uma descrição textual no arquivo README do projeto indicando qual variação/lógica foi utilizada/proposta. O programa deve tentar, e mostrar todas na tela, dez (10) iterações de emparelhamento em laço, organizando as saídas até a alocação final. Os pares Projetos x Alunos finais devem ser indicados, bem como os números finais. As soluções dadas em (Abraham, Irving & Manlove, 2007) são úteis e qualquer uma pode ser implementada com variações pertinentes. Um arquivo entradaProj2TAG.txt com as indicações de código do projeto, número de vagas, requisito mínimo das vagas, bem como dos alunos com suas preferências de projetos na ordem e suas notas indviduais, é fornecido como entrada. Uma versão pública do artigo de (Abraham, Irving & Manlove, 2007) é fornecida para leitura.

OBS: O programa possui um **executável** e um arquivo **"saida"** do tipo txt que se encontram na **pasta "dist" .** A cada vez que for clicado no executável o programa será executado abrindo na tela o arquivo "saida.txt" contendo os resultados da respectiva execução.

Documentação do programa em python para solução do problema:

Entrada de dados: Foi realizada a contagem manualmente sobre quais linhas do arquivo começavam e terminavam os dados referentes aos projetos ou alunos, assim dividindo a lista de dados de entrada em projetos e alunos.

```
#Le arquivo de entrada e separa os dados dos projetos e alunos
entradaDados = open("entradaProj2TAG.txt","+r").readlines()
entrada_projetos = entradaDados[3:53]
entrada_alunos = entradaDados[56:]

#Trata dados dos projetos e alunos em dicionarios
entrada_projetos = project_data(entrada_projetos)
entrada_alunos = student_data(entrada_alunos)
```

Para tratamento dos dados foram criados algoritmos individuais para os alunos e projeto. Ambos algoritmos de tratamento de dados recebem listas com os dados originalmente como vieram no arquivo de entrada, e retornam dicionários com os alunos ou projetos como chave, e seus respectivos atributos como valores.

Tratamento de dados dos projetos:

```
def project_data(entrada_projetos):
    project_data = {}
    for linha in entrada_projetos:
        #Remove caracters especiais não uteis para analises
        linha = linha.replace("(","").replace(")","")
        projeto,vagas,nota = linha.split(",")

        #Popula o dicionario de projetos
        project_data[projeto] = (int(vagas.replace(" ","")),int(nota.replace(" ","")))

    return project_data
```

Tratamento de dados dos alunos:

```
def student_data(entrada_alunos):
    student_data = {}
    for linha in entrada_alunos:
        #Faz separação dos dados dos alunos
        aluno,atributos = linha.split(":")

        #Remove caracters especiais não uteis para analises
        atributos = atributos.replace("(", "").replace(")", "").replace(",", "").replace("\n", "")

        try:
            #Separa as preferencias e nota do aluno em variaveis diferentes
            preferencias = atributos.split(" ")
            nota = int(preferencias.pop(-1))

            #Popula o dicionario de alunos com os valores de seus atributos
            student_data[aluno] = (preferencias,nota)

        except:
            print(f"ERRO no aluno: {aluno} com {preferencias}")


    return student_data
```

Construção do Grafo: O grafo foi construído com uso da biblioteca networkx para auxiliar nas buscas por valores específicos no grafo durante a implementação dos algoritmos necessários para solucionar o problema. O grafo possui vértices de projetos e alunos, onde as arestas são formadas em pares (aluno,projeto) onde a existência da aresta depende do interesse do aluno pelo projeto, e a nota do aluno define o peso da mesma.No tratamento dos dados de entrada é notório ausência de arestas entre os projetos ou alunos, então o conjunto de alunos e projetos formam partição diferentes. Portanto, os vértices foram passados ao grafo atribuindo valores que determinam qual partição ele é pertencente.

```
def geraGrafo(entrada_alunos,entrada_projetos):
    Grafo = nx.Graph()

    #Passa os vertices dos projetos como particao 0
    for projeto,especificacoes in entrada_projetos.items():
        Grafo.add_node(projeto, vagas = especificacoes[0],
                       nota = especificacoes[1], bipartite=0)

    #Passa dos vertices dos alunos particao 1
    for aluno,atributos in entrada_alunos.items():
        Grafo.add_node(aluno, vagas = 0,
                       nota = atributos[1], bipartite=1)

        notaAluno = atributos[1]

        #Cria aresta com nota do aluno como peso
        for projetoInteresse in atributos[0]:
            Grafo.add_edge(aluno,projetoInteresse,weight = notaAluno)

    return Grafo
```

Para verificar se o grafo construído é bipartido foi utilizado um método booleano da biblioteca networkx. Caso o grafo seja bipartido são levantados os dados referentes ao grafo e suas partições, como números de vértices e arestas.

```
if nx.is_bipartite(grafo):
            print(f"O grafo é bipartido com {Grafo.number_of_nodes()} vertices e {Grafo.number_of_edges()}
arestas")
        projetos,alunos = getBiparticao(Grafo)
        print(f"Quantidade de vertices da particao de projetos:{len(list(projetos))}")
        print(f"Quantidade de vertices da particao de alunos:{len(list(alunos))}")
```

```
    else:
        print("O grafo nao possui biparticao")
```

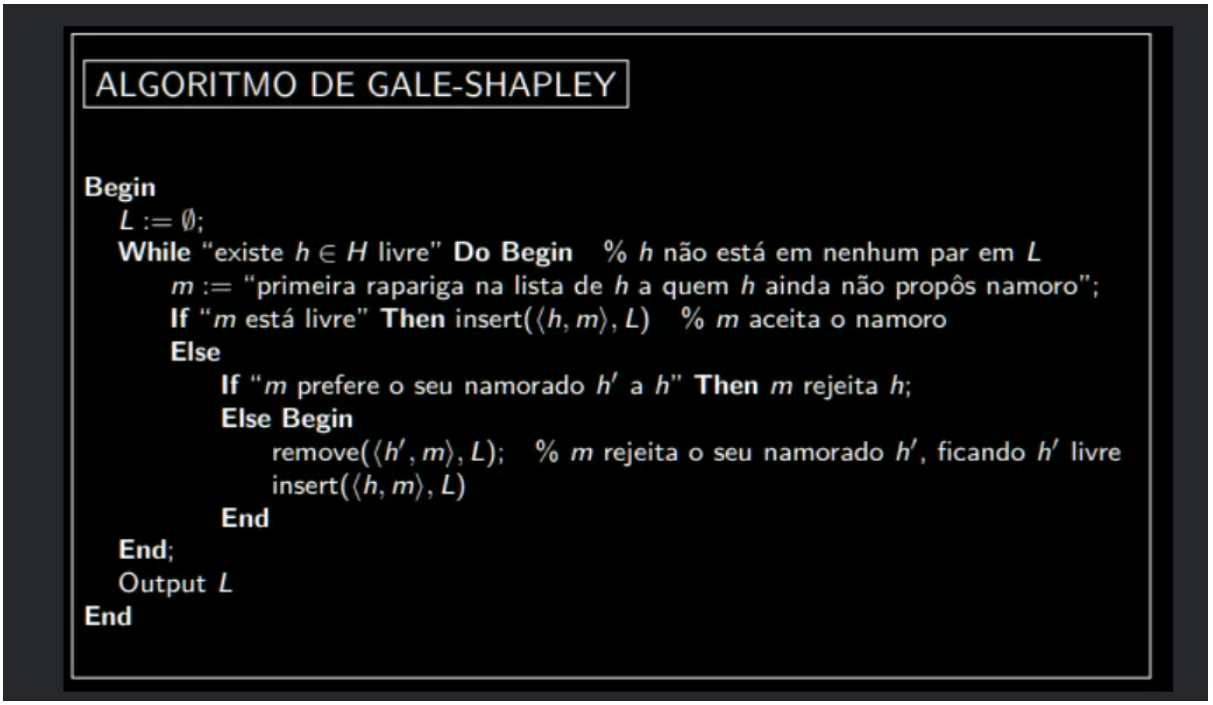
Após ser verificada a existência de bipartição, para obter os vértices de cada partição foi usada uma função não pertencente a biblioteca networkx:

```
def getBiparticao(Grafo):
    #Obtem particao zero
    projetos = {vertice for vertice, atributos in Grafo.nodes(data=True)
                if atributos["bipartite"] == 0}

    #Obtem a particao 1 subtraindo a particao zero do conjunto de vertices do grafo
    alunos = set(Grafo) - projetos

    return projetos,alunos
```

Para obter um emparelhamento estável no grafo o algoritmo implementado foi baseado no de Gale-Shapley usado no clássico problema do pareamento estável de casais. O problema citado consiste em formar o máximo de casais possível entre conjuntos de homens e mulheres, onde a busca garantir que os casais formados não irão desfazer-se por preferirem outro parceiro. Descrição do algoritmo:

Informal

1. Cada Rapaz e Rapariga ou estão comprometidos ou livres
2. Cada Rapariga, assim que fica comprometida, continuará nesse estado durante o resto da execução do algoritmo. Poderá, eventualmente, trocar de par
3. Todo o Rapaz que pede em namoro mais do que uma vez, fica com namoradas cada vez menos desejáveis
4. As Raparigas ficam tanto mais favorecidas quanto maior for o número de trocas de namorado
5. Quando um Rapariga livre recebe uma proposta, aceita e fica comprometida
6. Quando uma Rapariga comprometida recebe nova proposta, compara o novo pretende com o namorado e escolhe ficar com o que lhe favorece mais
7. Cada Rapaz pede namora às Raparigas seguindo a sua ordem de preferência
8. Assim que um Rapaz é rejeitado, propõe-se imediatamente à Rapariga seguinte na sua Lista de Preferências.



Fonte: https://resumos.leic.pt/emd/archive/relacionamentos-gale-shapley/

Para solucionar o problema em que está sendo trabalhado nesse projeto, os alunos foram tratados como os homens, e os projetos no qual se candidatam como as mulheres, e o critério de pertencimento ao projeto é feito analisando se a nota de argumento do aluno é maior ou igual a nota mínima estabelecida pelo professor. Os alunos que se candidatam a entrar em algum projeto e possuem nota suficiente, entram no projeto se houverem vagas disponíveis,caso todas vagas do projeto já tenham sido preenchidas, é avaliado se o candidato pode ocupar a vaga do aluno de nota mais baixa já proprietária de uma vaga. O processo descrito é realizado enquanto houver aluno que não tenha se candidatado a todos projetos que tenha interesse. Portanto, é retornado um dicionário contendo todos alunos aceitos nos projetos que tiveram vagas preenchidas.

Descrição informal do algoritmo implementado:

1. Inicialização:
   - Converte a entrada de alunos em uma lista e a embaralha, garantindo uma ordem aleatória de avaliação.
   - Inicializa um dicionário vazio chamado 'stable_matching' para armazenar o emparelhamento estável entre alunos e projetos.

2. Emparelhamento:
   - Enquanto houver alunos não avaliados:
     - Seleciona um aluno não avaliado aleatoriamente da lista de alunos.
     - Obtém os projetos que o aluno se candidata e sua nota.

3. Avaliação dos Projetos:
- Para cada projeto que o aluno se candidata:
  - Verifica se a nota do aluno é maior ou igual à nota mínima necessária para entrar no projeto.
- Se a nota for suficiente e ainda houver vagas no projeto:
  - Adiciona o aluno ao projeto no emparelhamento estável e decrementa o número de vagas do projeto.
- Caso contrário, se o projeto já estiver no emparelhamento:
  - Encontra o aluno com a nota mais baixa no projeto.
  - Se a nota do aluno atual for maior que a nota mais baixa no projeto:
    - Remove o aluno de nota mais baixa do projeto e o coloca de volta na lista de alunos não avaliados.
    - Adiciona o aluno atual ao projeto no lugar do aluno removido.

4. Conclusão:
- O algoritmo continua emparelhando os alunos com os projetos até que todos os alunos tenham sido avaliados e alocados a um projeto.

Algoritmo descrito acima implementado em python:

```python
def galeShapley(Grafo,projetos,alunos):
    alunos = list(copy.deepcopy(alunos))
    Grafo = copy.deepcopy(Grafo)
    random.shuffle(alunos)  #Embaralha a lista
    stable_matching = {} #{projeto:{aluno:nota}}

    #faz eparelhamento enquanto tiver aluno sem ser avaliado
    while len(alunos) > 0:
        #numeroAleatorio = random.randint(0,len(alunos) - 1)
        aluno = alunos.pop(0) #aluno atual
        alunoProjetosCandidato = Grafo.neighbors(aluno)  #projetos que o aluno se candidata
        notaAluno = Grafo.nodes[aluno]["nota"]  #Nota de argumento do aluno

        for projeto in alunoProjetosCandidato:

            #Nota minima p/ entrar no projeto
            requesitoProjeto = Grafo.nodes[projeto]["nota"]
            quantVagasProjeto = Grafo.nodes[projeto]["vagas"]

            #Add o aluno ao projeto e decrementa vagas
            if quantVagasProjeto > 0:
                if (notaAluno >= requesitoProjeto):
                    if projeto not in stable_matching.keys():
                        stable_matching[projeto] = {aluno:notaAluno}
                    else:
                        stable_matching[projeto][aluno] = notaAluno

                    Grafo.nodes[projeto]["vagas"] = quantVagasProjeto - 1
                    break

            elif projeto in stable_matching.keys():
                notaMaisBaixaProjeto = min(stable_matching[projeto].values())

                #Substitui o aluno de nota mais baixa no projeto
                if notaAluno > notaMaisBaixaProjeto:
```

```
                    #Remove do projeto o aluno de nota mais baixa
                    alunosProjeto = stable_matching[projeto]
                    alunoMenorNotaProjeto = min(alunosProjeto, key=alunosProjeto.get)
                    stable_matching[projeto].pop(alunoMenorNotaProjeto)

                    #O aluno que saiu do projeto volta aos alunos que devem ser avaliados
                    alunos.append(alunoMenorNotaProjeto)

                    #Aluno atual entra no projeto
                    stable_matching[projeto][aluno] = notaAluno

                    print(f'Projeto:{projeto};{alunoMenorNotaProjeto} saiu, entrou {aluno} ')
                    break

    print('')

    return stable_matching
```

Obtidos os alunos pertencentes a cada projeto, foram formados pares (projeto,aluno) onde os projetos além da sua nomenclatura padrão, receberam atributos alfabéticos de A a F para diferenciar os pares de alunos no mesmo projeto, uma vez que o conceito de emparelhamento não permite que um vértice contenha mais de uma aresta. Assim criando um novo grafo com todos vértices do grafo original, mas somente com arestas que formem um pareamento estável, de acordo com as preferências dos alunos e suas respectivas notas e aceitação nos projetos.

```
def GrafoEmparelhado(grafo,stable_matching,projetos,alunos):
    graph_matching = nx.Graph()
    derivacoes = ['A','B','C','D','E','F']

    projetosEmparelhados = set()
    alunosEmparelhados = set()

    for projeto,alunos_casados in stable_matching.items():
        projetosEmparelhados.add(projeto)

        for aluno in range(len(alunos_casados.keys())):
            #Cria um vertice de projeto e aluno para cada par (projeto,aluno)
            projeto_derivado = projeto + "_" + derivacoes[aluno]
            student = list(alunos_casados.keys())[aluno]

            alunosEmparelhados.add(student)

            #Divide projetos e alunos em particoes diferentes
            graph_matching.add_node(projeto_derivado,bipartite = 0)
            graph_matching.add_node(student,bipartite = 1)

            #Cria aresta (projeto,aluno)
            graph_matching.add_edge(projeto_derivado,student)

    print(f"Quantidade de alunos:{len(alunosEmparelhados)}")
    print(f"Quantidade de projetos:{len(projetosEmparelhados)}")

    #Obtem todos projetos e alunos que não formaram par
    projetosDesemparelhados = projetos - projetosEmparelhados
    alunosDesemparelhados = alunos - alunosEmparelhados

    graph_matching.add_nodes_from(projetosDesemparelhados,bipartite = 0)
    graph_matching.add_nodes_from(alunosDesemparelhados,bipartite = 1)
```

```
    return graph_matching
```

Para imprimir os pares de vértices do emparelhamento estável e os demais dados de análise, foi feita uma função para tal tarefa, onde imprime o emparelhamento estável encontrado e faz comparações com o grafo original.

```
def imprimeEmparelhamento(emparelhamento):
    print(f"\n####################### Maior emparelhamento ######################")

    for projeto, aluno in emparelhamento.edges():
        print(f"{projeto} -- {aluno}")

    print(f"\nQuantidade de vagas preenchidas:{len(emparelhamento.edges())}")

    projetosDesemparelhados = searchVertexDegree(emparelhamento,"bipartite",0)
    alunosDesemparelhados = searchVertexDegree(emparelhamento,"bipartite",1)

    alunos = set(node for node, atributo in emparelhamento.nodes(data=True) if atributo.get('bipartite') == 1)
    alunos = alunos - alunosDesemparelhados
    print(f"Quantidade de alunos em projetos:{len(alunos)}")

    if len(projetosDesemparelhados) == 0:
        print(f"Todos projetos possuem alunos")
    else:
        print(f"Total de projetos sem alunos: {len(projetosDesemparelhados)}")


    if len(alunosDesemparelhados) == 0:
        print(f"Todos alunos estao em algum projeto")
    else:
        print(f"Total de alunos sem projeto: {len(alunosDesemparelhados)}")
```

Dadas funções cima, foi criada uma função principal que reúne todas funções para análise dos dados:

```
def item(grafo):
    if nx.is_bipartite(grafo):
            print(f"O grafo é bipartido com {grafo.number_of_nodes()} vertices e {grafo.number_of_edges()}
arestas")
        projetos,alunos = getBiparticao(grafo)
        print(f"Quantidade de vertices da particao de projetos:{len(list(projetos))}")
        print(f"Quantidade de vertices da particao de alunos:{len(list(alunos))}")

    else:
        print("O grafo nao possui biparticao")

    emparelhamentos = []

    for i in range(10):
        print(f"\n################### Iteracao {i} ####################")
        stable_matching = galeShapley(grafo,projetos,alunos)

        stable_matching_graph = GrafoEmparelhado(grafo,stable_matching,projetos,alunos)
        emparelhamentos.append(stable_matching_graph)

        edges = set(stable_matching_graph.edges())
        quantEdges = len(edges)

        if i == 0:
            print(f"Primeiro emparelhamento com {quantEdges} pares")
```

```
        else:
            arestas_anteriores = emparelhamentos[i - 1].edges()

            arestas_diferentes = edges.difference(arestas_anteriores)

            if len(arestas_diferentes) > 0:
                print(f"Houveram mudancas de {len(arestas_diferentes)} arestas:{arestas_diferentes}")
            else:
                print(f"Nao houveram mudancas de arestas;")


    maiorEmparelhamento = max(emparelhamentos, key=len)
    imprimeEmparelhamento(maiorEmparelhamento)
```

A função acima tem como objetivo construir o grafo a partir dos dados dados obtidos no arquivo de entrada, verificar se o grafo construído é bipartido e executar o algoritmo de Gale-Shapley 10 vezes em busca de encontrar o emparelhamento máximo.

Portanto, o seguinte script realiza as análises de dados e retorna a saída do programa no arquivo “saida.txt” e no terminal:

```
#Define arquivo que obtera a saida do programa
nome_arquivo = 'saida.txt'

#Abre o arquivo de saida para escrita
with open(nome_arquivo, 'w') as arquivo_saida:
    #Redireciona a saída para o arquivo
    with redirect_stdout(arquivo_saida):
        Grafo = geraGrafo(entrada_alunos,entrada_projetos)
        item(Grafo)

os.startfile(nome_arquivo)

saidaTerminal = open(nome_arquivo,"+r").read()
print(saidaTerminal)
```

Repositorio no github com o projeto por completo:
https://github.com/VitorGuedes22/Grafos/tree/master/Projeto_2

Bibliotecas utilizadas:

```
import networkx as nx
import random
import subprocess
import sys
from contextlib import redirect_stdout
import os
import copy
```

https://networkx.org/documentation/stable/install.html

https://docs.python.org/pt-br/3/library/random.html

https://docs.python.org/3/library/subprocess.html

https://docs.python.org/3/library/sys.html

https://docs.python.org/3/library/contextlib.html

https://docs.python.org/3/library/os.html

Referências:

https://resumos.leic.pt/emd/archive/relacionamentos-gale-shapley/#descri%C3%A7%C3%A3o-do-algoritmo

https://networkx.org/documentation/stable/index.html

https://www.ibilce.unesp.br/Home/Departamentos/MatematicaAplicada/docentes/socorro/emparelhamentos.pdf

https://pt.wikipedia.org/wiki/Problema_do_emparelhamento_est%C3%A1vel

https://www.puc-rio.br/ensinopesq/ccpg/pibic/relatorio_resumo2014/relatorios_pdf/ctc/MAT/MAT-Jos%C3%A9%20Eliton%20Albuquerque%20Filho.pdf

https://ir.nctu.edu.tw/bitstream/11536/4210/1/A1990EX83000006.pdf