Relatório_dinâmico

December 7, 2019

0.1 Relatório dinâmico - Construção de Horário de aulas com atendimento de restrições e preferências.

Biblioteca utilizada para ler e manipular arquivos tipo xls, outra opção de biblioteca -> Pandas.

[ ]:
```
#coding: utf-8
import xlrd #biblioteca para ler arquivo xls
import time #biblioteca para receber tempo de execução do código
```

Função que retorna um valor para cada dia, a partir de uma string com seu nome

[ ]:
```
def getDia(dia):
    if(dia == "Segunda"): #recebe a string da lista dia para transformar
        dia = 0
    elif(dia == "Terça"):
        dia = 1
    elif(dia == "Quarta"):
        dia = 2
    elif(dia == "Quinta"):
        dia = 3
    elif(dia == "Sexta"):
        dia = 4
    else:
        dia = -10
    return dia
```

Função que busca a configuração dos horários na planilha e retorna valores de 0 a n para cada horário da planilha

[ ]:
```
def getHorario(horario, Configuracoes):
    for i in range(len(Configuracoes)):
        if(horario == Configuracoes[i]):
            return i
    return -10
```

Classe horário, que possui os atributos “dia”, “horário” e “cor”, classe que verifica a validade dos dados que estão entrando e atribui uma união de strings para determinar a cor

[ ]:
```
class Horario:
    def __init__(self, dia, horario):
        self.dia = dia
        self.horario = horario
        self.cor = str(dia) + "/" + str(horario) #simulacao de uma tupla para␣
↪representar a cor.

    def __setattr__(self, name, value):
        if(name == "cor"):
            super().__setattr__(name, value)
            aux = value.split("/")
            self.dia = int(aux[0])
            self.horario = int(aux[1])
        else:
            super().__setattr__(name, value)

    def add(self, horarioMax):
        if(self.dia in range(4)):
                self.dia += 1
            #elif(self.horario in range((horarioMax)-1)):
            #        self.dia = 0
            #        self.horario += 1
        else:
            self.dia = 0
            self.horario += 1
            self.cor = str(self.dia) + "/" + str(self.horario)
```

Classe Vertice, esta inicializa um vértice do grafo, possui um iD, sua cor obtida da classe Horario e a sua lista de adjacência. É a classe mais genérica utilizada para instanciação de 'vértices de restrição', utilizados na parte de restrições do projeto

[ ]:
```
class Vertice:
    idVertice = 0
    def __init__(self):
        self.saturacao = 0
        self.horario = Horario(-1, -1)
        self.adjacencia = []
        self.idVertice = Vertice.idVertice
        Vertice.idVertice += 1
```

Classe Professor, feita para a instância dos professores no problema, possuem um iD do professor e seu nome. Além disso, há também duas Listas que são as restrições e as preferências do docente.

[ ]:
```
class Professor:
    idProfessor = 0
    def __init__(self, nome):
        self.nome = nome
        self.idProfessor = Professor.idProfessor
```

```
        Professor.idProfessor += 1 #cada professor recebe um id iniciando de 0␣
↪até n professores, sao n-1 ids
        self.restricao = [] #lista de informacoes vindo das tabelas
        self.preferencia = [] #lista de informacoes vindo das tabelas
```

Classe Matéria, feita para a instância das matérias da escola, possuem um iD para identificação e seu nome.

[ ]:
```
class Materia:
    idMateria = 0
    def __init__(self, nome):
        self.nome = nome
        self.idMateria = Materia.idMateria
        Materia.idMateria += 1
```

Classe Turma, feita para a instância das turmas da escola, possuem um iD para identificação, além do nome da turma e de uma lista com suas restrições.

[ ]:
```
class Turma:
    idTurma = 0
    def __init__(self, nome):
        self.nome = nome
        self.idTurma = Turma.idTurma
        Turma.idTurma += 1
        self.restricao = [] #lista de informacoes vindo das tabelas
```

Classe MTP, rebece os atributos da classe pai Vertice, em que se instanciam vértices que possuem as informações de Matéria, Turma e Professor (através de seus iDs), são basicamentes vértices que juntam as características das 3 classes acima citadas

[ ]:
```
class MTP(Vertice):

    # Método de inicialização da MTP (Materia, Turma, Professor)

    def __init__(self, idMateria, idTurma, idProfessor):
        self.idMateria = idMateria
        self.idTurma = idTurma
        self.idProfessor = idProfessor
        self.geminada = False
        super().__init__() #iniciando a classe pai

    # Método feito para conceder uma cor aos vértices com base em suas␣
↪adjacências

    def recebeCor(self, vertice, cor):
        for a in vertice.adjacencia:
            if(self.Vertices[a].horario.cor == cor):
                return False
```

```
        vertice.horario.cor = cor
        return True
```

class Escola instância todas as listas e métodos filhos

[ ]:
```
class Escola:

    def __init__(self, escola):
        self.Vertices = []                          #Lista contendo todos os␣
↪vértices, Genéricos ou MTPs
        self.Configuracoes = []                 #Lista contendo todos os␣
↪horários da escola
        self.Materias = []                          #Lista contendo todas as␣
↪matérias cadastradas
        self.Turmas = []                           #Lista contendo todas as turmas␣
↪cadastradas
        self.Professores = []                 #Lista contendo todos os␣
↪profesores cadastrados
        self.MTPs = []                                  #Lista contendo todas as␣
↪referências de MTPs contidas na lista Vertices
        self.Preferencias = []                  #Lista contendo as preferências␣
↪não atendidas
        self.numPrefs = 0                          #Variável que controla a␣
↪quantidade total de preferências de professores
        self.numPrefAtendidas = 0         #Variável que controla a quantidade␣
↪total de preferências atendidas
        self.leDoArquivo(escola)
        self.criaListaAdj()
        self.addRestricao()
        self.addPreferencias()
        self.colorir()
        self.showValues()
```

Método para saída de dados padronizada

[ ]:
```
def showValues(self):
                print("Quantidade de cores:", self.contaCor())
                print("Vértices não coloridos:", self.semCor)
                print("Preferências atendidas sobre o total de preferências:",␣
↪self.numPrefAtendidas, "/", self.numPrefs)
                for p in self.Professores:
                        if(len(p.preferencia) > 0):
                                print(p.nome, ":", len(p.preferencia))
                print("")
```

O seguinte método cria as listas de adjacências de cada vértice(MTP), verificando com 2 laços de repetição se o seu par possui alguma turma em comum ou algum professor em comum fazendo com

que estes não possam ser pintados com a mesma cor, adicionando-os em suas respectivas listas de adjacências

[ ]:
```
def contaCor(self):
        cores = []
        for v in self.MTPs:
            if(self.Vertices[v].horario.cor not in cores):
                cores.append(self.Vertices[v].horario.cor)
        return len(cores)
```

[ ]:
```
def criaListaAdj(self):
    for idV1 in self.MTPs:
        for idV2 in self.MTPs:
            if(idV1 != idV2):
                if(self.Vertices[idV1].idTurma == self.Vertices[idV2].idTurma␣
↪or self.Vertices[idV1].idProfessor == self.Vertices[idV2].idProfessor):
                    self.Vertices[idV1].adjacencia.append(idV2)
```

O método a seguir adiciona ás listas de adjacencias dos vértices(MTPs) as restrições que foram obtidas através do arquivo, fazendo com que os vértices tenham cores diferentes dos casos onde essas restrições ocorrem

[ ]:
```
def addRestricao(self):
        for idV in self.MTPs:
            #print("MTP", "Materia: ", self.Materias[self.Vertices[idV].
↪idMateria].nome, "Turma: ", self.Turmas[self.Vertices[idV].idTurma].nome,␣
↪"Professor: ", self.Professores[self.Vertices[idV].idProfessor].nome)
            for t in self.Turmas:
                if(self.Vertices[idV].idTurma == t.idTurma):
                    #print("Turma: ", t.nome)
                    for restricao in t.restricao:
                        #print("Cor: ", self.Vertices[restricao].horario.cor)
                        self.Vertices[idV].adjacencia.append(self.
↪Vertices[restricao].idVertice)
            for p in self.Professores:
                #print("Professor: ", p.nome)
                if(self.Vertices[idV].idProfessor == p.idProfessor):
                    for restricao in p.restricao:
                        #print("Cor: ", self.Vertices[restricao].horario.cor)
                        self.Vertices[idV].adjacencia.append(self.
↪Vertices[restricao].idVertice)
```

Nesse método, todas as preferências, de todos os professores, são alocadas Caso uma preferência seja possível, o vértice MTP correspondente será pintado Caso não seja, ela é alocada numa lista de preferências não atendidas

[ ]:
```
def addPreferencias(self):
        numProfs = len(self.Professores)
```

```
        # Enquanto houverem preferências passíveis de resolução
        while(self.numPrefAtendidas + len(self.Preferencias) < self.numPrefs):
            lastProfId = -1
            # Busca nos vértices MTPs
            for v in self.MTPs:
                # Se o vértice não tiver cor
                if(self.Vertices[v].horario.cor == "-1/-1"):
                    # Se o professor atual não foi o último atendido
                    if(self.Vertices[v].idProfessor != lastProfId):
                        # Procura em suas preferências alguma que pode encaixar
↪com o vértice
                        for pref in self.Professores[self.Vertices[v].
↪idProfessor].preferencia:
                            # Caso consiga encaixar
                            if(self.recebeCor(self.Vertices[v], pref.cor)):
                                # Atualiza os valores para controle
                                self.numPrefAtendidas += 1
                                lastProfId = self.Vertices[v].idProfessor
                                if(pref in self.Preferencias):
                                    self.Preferencias.remove(pref)
                                self.Professores[self.Vertices[v].idProfessor].
↪preferencia.remove(pref)
                                break
                            else:
                                if(pref not in self.Preferencias):
                                    self.Preferencias.append(pref)
```

Esse método tentará todas as combinações de cores possíveis para um vértice parando apenas quando uma for aceita

[ ]:
```
def escolheCor(self, vertice):
        #print(vertice.idVertice)
        #print(vertice.horario.cor)
        horario = Horario(0, 0)
        maxHorario = len(self.Configuracoes)
        while(not self.recebeCor(vertice, horario.cor)):
            horario.add(maxHorario)
        #print(vertice.horario.cor)
        if(vertice.horario.cor != "-1/-1"):
            return True
        return False
```

Em def colorir(self), estamos utilizando os conceitos de coloração do algoritmo DSatur, verificando grau dos vertices e grau de saturação

[ ]:
```
def colorir(self):
        listNoColor = []
```

```
        for v in self.MTPs:
            #print(self.Vertices[v].horario.cor)
            if(self.Vertices[v].horario.cor == "-1/-1"):
                #print("Entrou")
                listNoColor.append(self.Vertices[v].idVertice)
        while(len(listNoColor) > 0):
            maxGrau = -1
            maxSat = -1
            index = -1
            for v1 in listNoColor:
                if(index == -1):
                    index = v1
                    maxGrau = len(self.Vertices[v1].adjacencia)
                    maxSat = self.Vertices[v1].saturacao
                elif(maxGrau < len(self.Vertices[v1].adjacencia)):
                    index = v1
                    maxGrau = len(self.Vertices[v1].adjacencia)
                    maxSat = self.Vertices[v1].saturacao
                elif(maxGrau == len(self.Vertices[v1].adjacencia) and maxSat <␣
↪self.Vertices[v1].saturacao):
                    index = v1
                    maxGrau = len(self.Vertices[v1].adjacencia)
                    maxSat = self.Vertices[v1].saturacao
            self.escolheCor(self.Vertices[index])
            listNoColor.remove(self.Vertices[index].idVertice)
            for adj in self.Vertices[index].adjacencia:
                self.Vertices[adj].saturacao += 1
```

Todo o método a seguir foi feito para puxar os dados da planilha fornecida com os dados e possibilitar a manipulação e testes do problema através desses, a biblioteca utilizada para isso foi a xlrd, os dados das planilhas são incluidos em suas respectivas listas para manipulação

[ ]:
```
def leDoArquivo(self, escola):

    # Zera os ids

    Vertice.idVertice = 0
    MTP.idMTP = 0
    Professor.idProfessor = 0
    Turma.idTurma = 0
    Professor.idProfessor = 0

    # Abre o arquivo

    wb = xlrd.open_workbook(escola) #instanciando wb com a biblioteca xlrd
    #wb = workbook
    # Abre a planilha 0, Dados
```

```
   ws = wb.sheet_by_index(0) #instanciando ws para manipular as páginas do␣
↪arquivo
   #sheet_by_index(int) acessa a pág. pelo índice, sheet_by_name acessa pelo␣
↪nome da pág.
   for i in range(ws.nrows): #nrows argumento padrão da biblioteca number rows
       # Ignora linha de cabeçalho
       if i == 0:
           continue
       else:
           materiaNome = ws.cell(i, 0).value #ws.cell(int row, int column)␣
↪acessa uma célula do arquivo
           turmaNome = str(ws.cell(i, 1).value)
           professorNome = ws.cell(i, 2).value
           qtd = ws.cell(i, 3).value

           matId = 0
           if(len(self.Materias) != 0):
               existe = False
               for materia in self.Materias:
                   if(materiaNome == materia.nome):
                       existe = True
                       matId = materia.idMateria
               if(not existe):
                   materia = Materia(materiaNome)
                   matId = materia.idMateria
                   self.Materias.append(materia)
           else:
               self.Materias.append(Materia(materiaNome))

           turmaId = 0
           if(len(self.Turmas) != 0):
               existe = False
               for turma in self.Turmas:
                   if(turmaNome == turma.nome):
                       existe = True
                       turmaId = turma.idTurma
                       break
               if(not existe):
                   turma = Turma(turmaNome)
                   turmaId = turma.idTurma
                   self.Turmas.append(turma)
           else:
               self.Turmas.append(Turma(turmaNome))

           professorId = 0
           if(len(self.Professores) != 0):
               existe = False
```

```
                for professor in self.Professores:
                    if(professorNome == professor.nome):
                        existe = True
                        professorId = professor.idProfessor
                        break
                if(not existe):
                    professor = Professor(professorNome)
                    professorId = professor.idProfessor
                    self.Professores.append(professor)
            else:
                self.Professores.append(Professor(professorNome))
            for i in range(int(qtd)):
                mtp = MTP(matId, turmaId, professorId)
                self.Vertices.append(mtp)
                self.MTPs.append(mtp.idVertice)

    # Abre a planilha 1, Configurações

    ws = wb.sheet_by_index(1)
    for i in range(ws.nrows):
        if i == 0:
            continue
        else:
            self.Configuracoes.append(ws.cell(i, 0).value)

    # Abre a planilha 2, Restrição

    ws = wb.sheet_by_index(2)
    for i in range(ws.nrows):
        if i == 0:
            continue
        else:
            professor = ws.cell(i, 0).value
            for p in self.Professores:
                if(p.nome == professor):
                    v = Vertice()
                    v.horario.cor =  str(getDia(ws.cell(i, 2).value)) + "/" + ␣
↪str(getHorario(ws.cell(i, 1).value, self.Configuracoes))
                    self.Vertices.append(v)
                    p.restricao.append(v.idVertice)
                    break

    # Abre a planilha 3, Restrições_Turma

    ws = wb.sheet_by_index(3)
    for i in range(ws.nrows):
        if i == 0:
```

```
            continue
        else:
            turma = str(ws.cell(i, 0).value)
            for t in self.Turmas:
                if(t.nome == turma):
                    v = Vertice()
                    v.horario.cor =  str(getDia(ws.cell(i, 2).value)) + "/" + ␣
↪str(getHorario(ws.cell(i, 1).value, self.Configuracoes))
                    self.Vertices.append(v)
                    t.restricao.append(v.idVertice)
                    break

    # Abre a planilha 4, Preferências

    ws = wb.sheet_by_index(4)
    for i in range(ws.nrows):
        if i == 0:
            continue
        else:
            professor = ws.cell(i, 0).value
            for p in self.Professores:
                if(p.nome == professor):
                    h = Horario(getDia(ws.cell(i, 2).value), getHorario(ws.
↪cell(i, 1).value, self.Configuracoes))
                    p.preferencia.append(h)
                    self.numPrefs += 1
                    break
```

Dado um vértice qualquer e uma cor, essa função tenta alocar essa cor ao vértice Em caso positivo, a cor é definida e o retorno é True, em caso Negativo o retorno e False

[ ]:
```
def recebeCor(self, vertice, cor):
        if(cor != "-1/-1" and cor != "-10/-10"):
            for a in vertice.adjacencia:
                if(self.Vertices[a].horario.cor == cor):
                    return False
        vertice.horario.cor = cor
        return True
```

Código abaixo desenvolvido para mostrar os resultados obtidos, tempo de execução, preferencias atendidas e restrições atendidas

[ ]:
```
inicio = time.time()
print("Escola A")
a = Escola("Escola_A.xlsx")
fim = time.time()
print("Tempo de execução:", fim-inicio)
```

```
print("")
inicio = fim
print("Escola B")
b = Escola("Escola_B.xlsx")
fim = time.time()
print("Tempo de execução:", fim-inicio)
print("")
inicio = fim
print("Escola C")
c = Escola("Escola_C.xlsx")
fim = time.time()
print("Tempo de execução:", fim-inicio)
print("")
inicio = fim
print("Escola D")
d = Escola("Escola_D.xlsx")
fim = time.time()
print("Tempo de execução:", fim-inicio)
print("")
```