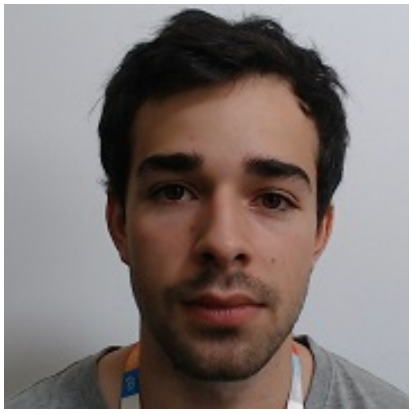
Universidade do Minho
Escola de Engenharia

Processamento de Linguagens

Trabalho Prático 1

Exercício 2

João Pedro da Santa Guedes A89588
Luís Pedro Oliveira de Castro Vieira A89601



A89588

A89601

5 de abril de 2021

Índice

Listings

1 Introdução

No âmbito da Unidade Curricular de Processamento de Linguagens, foi proposto ao grupo, como forma de desenvolver e demonstrar os seus conhecimentos sobre aquilo que tem vindo a ser alvo de atenção na UC, a realização de um exercício com diversas alíneas presentes num enunciado. Neste, existiam cinco exercícios diferentes, sendo que o grupo ficou com o exercício dois, Processador de Pessoas listadas nos Róis de Confessados.

Assim, no presente relatório, passaremos a descrever o processo de resolução do problema e de cada alínea, bem como explicar o código implementado para o mesmo.

2 Estrutura do Relatório

- A primeira parte corresponde à introdução, que irá retratar qual o objetivo do trabalho bem como a escolha do enunciado a desenvolver.
- A segunda parte corresponde ao presente tópico onde será explicado como está organizado este documento, referindo os capítulos existentes e explicando o conteúdo de cada um.
- A terceira parte corresponde à análise e especificação. É neste capítulo que é feita uma análise detalhada do problema proposto de modo a se poder especificar o desenvolvimento do mesmo.
- No quarto capítulo será apresentado e explicado o raciocínio por trás da resolução de cada alínea, usando como auxiliar o código implementado.
- Na quinta parte serão apresentados exemplos do programa através de *capturas de ecrã*.
- No sexto capítulo, irá constar a conclusão que irá conter uma síntese do que foi retratado ao longo do documento, a conclusão e uma análise crítica do trabalho realizado.

3 Exercício e o seu Dataset

Como mencionado previamente, o exercício a ser alvo de avaliação e resolução por parte do grupo é o exercício dois.

Os Róis de Confessados são arquivos do arcebispado existentes no ADB que contém o registo de rapazes que pretendiam seguir a vida clerical e se candidatavam aos seminários, tendo-os sido possível obter uma versão digital desse arquivo. Construa, então, um ou vários programas Python para processar o ficheiro de texto 'processos.xml'

Ao analisarmos cuidadosamente o dataset verificamos rapidamente que o mesmo está dividido em secções denominadas ***processo*** e que cada uma destas contém as seguintes tags:

- processo $->$ Onde poderemos também ter acesso ao ID do processo.
- pasta $->$ Pasta onde se encontra guardado o processo.
- data $->$ Data do registo do processo.

- nome $->$ Nome do candidato.
- pai $->$ Nome do pai do candidato.
- mae $->$ Nome da mãe do candidato.
- obs $->$ Observações referentes ao processo.

Após algum tempo a analisar mais a fundo cada uma das tags e o seu conteúdo, e mesmo durante a resolução das diferentes alíneas do exercício, reparámos que existem algumas falhas a nível do dataset, propositadas ou não, que dificultaram alguns dos raciocínios que tivemos durante a resolução do problema de forma geral.

Seguem abaixo alguns dos erros que observámos durante este trabalho:

- Existem processos com o mesmo ID e conteúdo igual, e outros com o mesmo ID mas conteúdo diferente.
- Problemas gramaticais, normalmente falta de espaços ou pontuação (no $< obs >$ existe o nome "eAntonio" na descrição de dois filhos).
- Pessoas podem ter informação adicional no seu nome, normalmente entre parenteses ou aspas.
- Parentes podem não estar associados a processos no $< /obs >$, logo não é possível confirmar se existem.
- Pessoas podem não ter pai e/ou mãe.

Assim, de forma geral, a maioria dos problemas não afeta de forma absurda a resolução do exercício. No entanto, o grupo considerou que o primeiro problema mencionado, a repetição de processos com o mesmo ID, tendo ou não conteúdos diferentes, se torna algo que inviabilizaria o nosso programa de forma séria caso não tratassemos do mesmo. A melhor solução seria alteral os ID's dos processos como forma de podermos utilizar toda a informação do dataset fornecido. No entanto, parecendo pouco ético, e de forma a manter um programa eficiente, durante toda a resolução do problema, consideramos para dois ID's iguais, o primeiro processo que aparecer, não fazendo distinção pelo conteúdo que possuem.

4 Resolução do Exercício

De seguida passaremos a explicar o raciocínio e o código presentes na resolução de cada alínea, de modo a que seja facilmente percetível aquilo que o grupo pretendia demonstrar com os seu conhecimentos.

Achamos importante mencionar desde já, e para não haver necessidade de repetição em todas as explicações, que para a resolução das diferentes alíneas lemos o ficheiro fornecido por inteiro e dividimos a informação pelas tags $< processo >< /processo >$, as quais dividem todos os blocos de dados que são relevantes para a resolução do problema e nos dá acesso aos mesmo. Isto é feito através do método *getProcessos*.

```
1 def getProcessos(lines):
2     m = re.findall(r'(<processo .*>((.|\n)+?)</processo>)',lines)
3     return m
```

Listing 1: Método getProcessos

Para além disso, fazemos também a verificação, como mencionado acima, dos IDs dos processos que já foram avaliados de forma a não repetirmos processos que tenham o mesmo ID, o que poderia levar à duplicação de informação desnecessariamente. Esta verificação é feita através do uso de um **set** denominado *processos_avaliados* no qual, para cada ID obtido verificamos se este já se encontra inserido no set, e em caso afirmativo, passamos à frente, isto é, não trabalhamos com o mesmo; em caso negativo, adicionamos ao set e depois sim trabalhamos com a informação sobre esse processo.

```
1     for p in processos:
2         pr = p[0]
3         if _id_ := gf.getId(pr):
4             if _id_ in processos_avaliados:
5                 pass
6             else:
7                 #Codigo referente a alinea em questao
```

Listing 2: Verificação de IDs

4.1 Alinea a) - query1.py

Calcular o número de processos por ano; apresente a listagem por ordem cronológica e indique o intervalo de datas em que há registos bem como o número de séculos analisados.

Desde o início da resolução desta alínea que nos apercebemos que iríamos precisar de uma estrutura que pudesse facilmente albergar informação simultânea sobre os diferentes anos e o número de processos existente para cada um que exista no dataset.

Assim, decidimos fazer uso dos dicionários de Python para este propósito, guardando como ***key*** o ano em que cada processo foi registado, e como ***value*** um set com os IDs dos processos que foram registados no respetivo ano da key.

```
if data := re.search(r'<data>((\d{4})-\d{2}-\d{2})</data>',pr):
                    dt = data.group(2)
                    if lista := anos.get(dt):
                        lista.add(_id_)
                    else:
                        aux = set()
                        aux.add(_id_)
                        anos.update({dt : aux})
                    seculos.add(gf.getSeculo(dt))
```

Listing 3: Cálculo do número de processos em cada ano

Para além disso, e de forma a avaliar o número de séculos avaliados adicionamos a um *set* denominado **seculos** o século correspondente a cada ano, imprimindo no final o seu comprimento. Para o cálculo do século de um ano fazemos uso do método *getSeculo*.

```
    def getSeculo(ano):
        i = int(ano)
        if (i<100):
            cent = 1
        elif i % 100 == 0:
            cent = int(i/100)
        else:
            cent = int(i/100 + 1)
        return cent
```

Listing 4: Método getSeculo

Por fim, de forma a responder também à questão sobre o intervalo de datas em que existem processos registados, o grupo considerou que o pretendido era encontrar a primeira e última data presentes no dataset, em que existe algum registo de qualquer processo. Para tal inicializamos duas variáveis, *dMin* e *dMax* como ***None*** e à medida que vamos avaliando as datas de cada processo, vamos comparando com essas variáveis de forma a obtermos a data mais antiga, que será guardada em *dMin*, e a data mais recente, que será guardada em *dMax*, existentes no dataset fornecido.

```
    if dMin is None:
        dMin = data.group(1)
    if dMax is None:
        dMax = data.group(1)
    if data.group(1) < dMin:
        dMin = data.group(1)
    if data.group(1) > dMax:
        dMax = data.group(1)
```

Listing 5: Cálculo das datas mais antiga e recente

Por fim, ordenamos o dicionário que contém informação sobre os processos guardados por ano, pelo valor da *key*, ou seja, por ordem cronológica, imprimindo depois para cada um dos anos o comprimento do set correspondente, que nos dará o número de processos registados.

4.2 Alínea b) - query2.py

Calcular a frequência de nomes próprios (primeiro nome) e apelidos (último nome) global e mostre os 5 mais frequentes em cada século.

Tal como na alínea anterior, faremos uso dos dicionários de Python para nos auxiliar na resolução desta alínea, no entanto, desta vez faremos uso de *nested dictionaires*, isto é, vamos ter dois dicionários, um para os nomes próprios e outro para os apelidos, onde poderemos encontrar um dicionário dentro de um dicionário.

Conforme é pedido na alínea, é esperado que sejamos capazes de calcular a frequência dos nomes quer a nível global, quer por séculos. Então quer para os nomes próprios, quer para os apelidos, os seus dicionários são compostos por uma primeira chave que é o nome em questão, correspondendo a um dicionário onde a sua chave, a segunda chave, é o século em que o processo que se encontra a ser avaliado foi registado e o valor é o número de ocorrências desse nome no respetivo século.

Assim, conseguimos facilmente obter a frequência de cada nome, próprio ou apelido, em cada século, e com uma simples iteração do dicionário obtemos o valor global.

Para calcular a frequência dos nomes por século começamos por obter o ano em que o processo foi registado e fazemos uso do método ja acima mencionado, *getSeculo*. Apesar de já sabermos quais os séculos existentes através da alínea a), o grupo decidiu tratar cada alínea como algo individual e assumimos que não temos conhecimento nenhum sobre qualquer informação prévia pertinente. Assim, temos duas variáveis, *minSec* e *maxSec*, que mais tarde serão usadas, para sabermos qual o "range" de séculos que possuímos no nosso dataset.

```
if _data_ := re.search(r'<data>((\d{4})-\d{2}-\d{2})</data>',pr):
                    ano = _data_.group(2)
                    sec = gf.getSeculo(ano)
                    if sec < minSec:
                        minSec = sec
                    if sec > maxSec:
                        maxSec = sec
```

Listing 6: Obtenção do século

De seguida, procuramos pelo nome do processo em questão na tag $< nome >< /nome >$, fazendo o split do mesmo por espaços. Este split resultará num array com cada um dos nomes presentes no conteúdo da tag. Para obtermos o nome próprio e o apelido basta fazer-mos uso das propriedades de navegação em arrays do Python, isto é, para o nome próprio acedemos ao índice **0** do array e para o apelido acedemos ao índice **-1** do array.

```
if _nomeComp_ := re.search(r'<nome>(((\w+)[ .\t]*)+)</nome>',pr):
                        _nomeComp_ = _nomeComp_.group(1)
                        _proprio_ = _nomeComp_.split(r' ')[0]
                        _apelido_ = _nomeComp_.split(r' ')[-1]
```

Listing 7: Obtenção do nome próprio e apelido

Como já temos o século, o nome próprio e o apelido, basta-nos agora tratar a informação e atualizar o dicionário. Então, começamos por verificar se já existe o nome no respetivo dicionário. Caso não haja, atualizamos o dicionário com o nome como key para o valor $\{sec : 1\}$, que representa um dicionário com uma chave correspondente ao século que obtivemos e o valor 1, visto ser a primeira ocurrência daquele nome.

Caso já exista o nome no respetivo dicionário vamos verificar se para esse nome já existe alguma ocorrência no século que obtivemos. Caso não exista, atualizamos o valor do nome que encontrámos com {*sec:1*}, significando a primeira ocurrência daquele nome no respetivo século.
Caso já tenha sido registado previamente o nome no século que obtivemos então simplesmente atualizaremos o valor do século para *valor_anterior + 1*.

```
#Atualizar Dicionario de Nomes Proprios
if p := proprios.get(_proprio_):
    #Se ja estiver registado
    if p_dict_sec := p.get(sec):
        #Se ja existir seculo registado
        p.update({sec:p_dict_sec+1})
    else:
        #Caso o seculo ainda nao tenha sido registado
        p.update({sec:1})
else:
    #Caso o nome ainda nao tenha sido registado
    proprios.update({_proprio_: {sec:1}})

#Atualizar Dicionarios de Apelidos
if a := apelidos.get(_apelido_):
    #Se ja estiver registado
    if a_dict_sec := a.get(sec):
        #Se ja existir seculo registado
        a.update({sec : a_dict_sec+1})
    else:
        #Caso o seculo ainda nao tenha sido registado
        a.update({sec : 1})
else:
    #Caso o apelido ainda nao tenha sido registado
    apelidos.update({_apelido_: {sec:1}})
```

Listing 8: Atualização dos Dicionários dos Nomes

4.3 Alínea c) - query3.py

Calcular o número de candidatos (nome principal de cada processo) que têm parentes (irmão, tio, ou primo) eclesiásticos; diga qual o tipo de parentesco mais frequente.

Para a resolução desta alínea idealizamos novamente o uso de um dicionário de Python, com o grau de parentesco como key e a frequência com que ocorrem no dataset.
Como mencionado no capítulo 3, existem alguns problemas a nível gramatical no dataset que tiveram de ser tomados em conta, especialmente na resolução da presente alínea.

Para averiguarmos os graus de parentes eclesiásticos, tal como pedido, existentes, devemos observar o conteúdo presente nas tags $< obs >< /obs >$ e a partir dessa informação começar a resolução do nosso exercício. Como apenas queremos os parentes eclesiásticos, consideramos apenas aqueles que nas observações têm também referência ao seu processo, problema mencionado acima. Não faremos também distinção no número de parentes que aparecem, isto é, caso apareça, o que

realmente ocorre, "X, Y e Z, Irmaos." ou "X e Y, Tios Maternos." apenas contabilizamos uma ocorrência desse grau de parentesco, de forma a facilitar os cálculos.

Após obtermos o conteúdo existente dentro das observações, procedemos a uma formatação inicial de forma a termos uma frase corrida e de fácil manipulação. Esta frase sofrerá então um *split* pelo identificador do processo, o que nos permite diferenciar os diferentes parentes.

```
if _obs_ := re.search(r'<obs>((.|\n)+?)</obs>',pr):

        obsConteudo = _obs_.group(1)
        obsConteudo = re.sub(r'\s+',r' ',obsConteudo)

        obsConteudo_splited = re.split(r'\. ?Proc.\d+\. ?',obsConteudo)
```

Listing 9: Formatação Inicial das Observações

```
['Doc.danificado. Joao Jose,Irmao. Beneditino. CSB-42, fls.313-324v.']
```

Listing 10: Exemplo da Primeira Formatação

Procedemos ainda a uma nova formatação, desta vez para cada elemento existente em *obsConteudo_splited*, para melhor trabalharmos a informação que nos interessa e que é efetivamente relevante e necessária para o resultado final. Assim voltamos a fazer split desta vez por '.', para podermos separar expressões como 'Doc.Danificado.' ou 'Doc.danificado. Joao Jose,Irmao. Beneditino. CSB-42, fls.313-324v.', elementos que possuem informação que queremos, mas que seria de difícil acesso caso não tratassemos a mesma.

Da resultante formatação iremos procurar em cada elemento do novo array criado, *obsConteudo_splited2*, pelas expressões que estejam separadas por vírgulas, indicativo de que existe um nome associado a um grau de parentesco, algo que concluímos após uma cuidadosa observação do dataset fornecido.

```
for element in obsConteudo_splited:
    obsConteudo_splited2 = re.split(r'\.',element)

    for elem in obsConteudo_splited2:
      if m:= re.search(r'(([A-Z]\w+( e| e | |,)?)+) ?,((([A-Z]\w+) ?)*)',elem):
        g = m.group(4)
```

Listing 11: Segunda Formatação e Procura por Nome+Parentesco

```
['Doc', 'danificado', ' Joao Jose,Irmao', ' Beneditino', ' CSB-42, fls',
   '313-324v', '']
```

Listing 12: Exemplo da Segunda Formatação

No entanto, mesmo após análise dos resultados que obtivemos até este ponto, o programa já se encontrava capaz de devolver os vários graus de parentesco mencionado nas observações, sim, mas ainda devolvia informação que não era relevante para o exercício como 'Santa Maria', 'Sao Tome', entre outros, que não são os graus de parentesco como pretendemos. Para além disso, verificamos também que o programa devolvia linhas em branco, algo que não pretendemos adicionar ao nosso dicionário, nem contabilizar. Assim criamos um filtro para estas duas situações.

```
if re.search(r'(?i:(sao|san|nos))',g):
    pass
```

```
3
4 elif g == '':
5     pass
```

Listing 13: Filtro

Como também requisitado no enunciado, *'Calcular o número de candidatos (...) que têm parentes (...) eclesiásticos'*, antes de começar o processo da segunda formatação, inicializamos uma variável ***accountedFor*** a **False**, uma espécie de flag, que nos indica se aquele processo já foi contabilizado como candidato ou não, uma vez que, na segunda formatação iremos iterar sobre os vários elementos presentes nas observações de forma separada. Como só contam os candidatos que possuem parentes eclesiásticos, e esta verificação só é feita aquando da iteração sobre os elementos que sofreram a segunda formatação, a variável deve ser inicializada antes da segunda formatação e, caso seja encontrado uma parente eclesiástico, esta passa a ter o valor **True** e é contabilizado 1 candidato. Prevenimos assim a contagem incorreta de vários candidatos dentro do mesmo conteúdo das observações.

Tendo já na nossa posse a informação que pretendemos tratar, falta apenas inserir ou atualizar o dicionário com os vários graus de parentesco existentes e a respetiva contabilização. Para facilitar a contagem e a interpretação dos dados, passamos todos os graus de parentesco para o singular. De seguida verificamos se já existe o grau no dicionário: caso exista, adicionamos à contagem anterior mais 1; caso não exista, atualizamos o dicionário com {g_sing:1}.

```
1  for elem in obsConteudo_splited2:
2      if m:= re.search(r'(([A-Z]\w+( e| e | |,)?)+) ?,((([A-Z]\w+) ?)*)',elem):
3          g = m.group(4)
4
5          if re.search(r'(?i:(sao|san|nos))',g):
6              pass
7
8          elif g == '':
9              pass
10
11         else:
12             if not accountedFor:
13                 contCandidatos += 1
14                 accountedFor = True
15
16             g_sing = re.sub(r's',r'',g)
17
18             if count := graus.get(g_sing):
19                 graus.update({g_sing : count+1})
20
21             else:
22                 graus.update({g_sing : 1})
```

Listing 14: Captura e contabilização dos graus de parentesco

No final, e para apresentação da informação, ordenamos o dicionário pelos seus valores, de forma decrescente, de forma a obtermos o grau mais frequente.

4.4 Alínea d) - query4.py

Verificar se o mesmo pai ou a mesma mãe têm mais do que um filho candidato.

Um dos problemas que afeta a lógica desta alínea e, por consequente, a nossa resolução é o facto de existirem diversos nomes repetidos ao longo do dataset, quer no pai quer na mãe, que podem ou não ser pessoas diferentes, não temos como concluir.

Por isso, a nossa resolução baseia-se em: para cada processo, dependendo se o utilizador escolheu procurar pelo nome de um pai, ou pelo nome de uma mãe, verificar se o nome que se encontra dentro da tag correspondente (ou $< pai >< /pai >$ ou $< mae >< /mae >$) é igual ao nome introduzido pelo utilizador do programa.

Caso seja encontrado um nome como aquele que o utilizador introduziu, então procedemos à procura, dentro da tag das observações, pela palavra 'irmao' ou 'irmaos', o que significaria que tal pai ou mãe possui mais do que um filho.

Em caso de sucesso, retornamos uma mensagem com o ID do processo onde foi encontrado mais do que um filho para o nome introduzido.

```
    processos_avaliados = set()
    f = open('processos.xml', 'r')
    processos = gf.getProcessos(f.read())


    for p in processos:
        pr = p[0]

        if _id_ := gf.getId(pr):

            if _id_ in processos_avaliados:
                pass
            else:
                processos_avaliados.add(_id_)

                if re.search(rf'<pai>{pai}</pai>',pr):
                    if m := re.search(r'<obs>((.|\n)*)</obs>',pr):
                        i = m.group(1)
                        if re.search(r'(?i:(irmao|irmaos))',i):
                            print(f'O {pai} do processo {_id_} tem mais do que um filho.')
                            return

    print(f'{pai} n o tem mais que um filho.')
    f.close()
```

Listing 15: Função de Procura pelo Nome do Pai

```
def procuraMae(mae):
    f=open('processos.xml','r')
    processos = gf.getProcessos(f.read())

    processos_avaliados = set()


    for p in processos:
        pr = p[0]

        if _id_:= gf.getId(pr):

            if _id_ in processos_avaliados:
                pass
            else:
                processos_avaliados.add(_id_)

                if re.search(rf'<mae>{mae}</mae>',pr):
                    if m := re.search(r'<obs>((.|\n)*)</obs>',pr):
                        i = m.group(1)
                        if re.search(r'(?i:(irmao|irmaos))',i):
                            print(f'A {mae} do processo {_id_} tem mais do que
    um filho.')
                            return


    print(f'{mae} n o tem mais que um filho.')
    f.close()
```

Listing 16: Função de Procura pelo Nome da Mãe

4.5 Alínea e) - query5.py

Utilizando a linguagem de desenho de grafos DOT desenhe todas as árvores genealógicas (com base nos triplos <filho,pai,mãe>) dos candidatos referentes a um ano dado pelo utilizador.

O principal problema que nos deparámos nesta alínea, foi a questão de existirem imensas pessoas com os mesmos nomes, e por isso, chega a uma altura que começa a ser confuso. Por isso tivemos de arranjar solução para estas ambiguidades.

Para nos ajudar e simplificar tudo, definimos 2 funções extras: **getProcID** e **getIrmaos**. A primeira é utilizada para retirar o id do processo dando uma frase do tipo "Proc.XXXX." enquanto que a segunda serve simplesmente para devolver uma lista com todos os irmãos que estão referidos nas observações.

```
def getProcID(proc):
    if m := re.search(r' ?Proc\.(\d+)\.?',proc):
        return m.group(1)
```

Listing 17: Função de devolução do ID do processo

```
def getIrmaos(obs):
    l = []
    if irmaos := re.findall(r'([\w ,]*,Irmao(s)?[\w ]*\.( ?Proc\.\d+\.)?)',obs):
        for i in irmaos:
            l.append(i[0].strip())
    return l
```

Listing 18: Função de devolução dos Irmãos situados nas observações

A nossa resolução baseia-se então em filtrar todos os processos da data pedida pelo utilizador e criar uma ligação entre a pessoa, e os seus pais. De seguida procura-se os irmãos que possam estar nas observações e constrói-se outra aresta entre cada irmão e os pais. É de realçar que fazemos sempre controlo dos processos que já foram avaliados.

```
    processos = gf.getProcessosByData(f.read(), ano)

    for pr in processos:
        if _id_ := gf.getId(pr):
            if _id_ in processos_avaliados:
                pass
            else:
                auxID = str(startID)
                if nome := re.search(r'<nome>((.|\n)*)</nome>',pr):
                    nome = nome.group(1)
                    processos_avaliados.add(_id_)
                    dot.node(nome+_id_,nome)

                    mae = re.search(r'<mae>((.|\n)*)</mae>',pr)
                    if mae:
                        mae = mae.group(1)
                        dot.node(mae+auxID,mae)
                        dot.edge(nome+_id_,mae+auxID)
                    pai = re.search(r'<pai>((.|\n)*)</pai>',pr)
                    if pai:
                        pai = pai.group(1)
                        dot.node(pai+auxID,pai)
                        dot.edge(nome+_id_,pai+auxID)
```

Listing 19: Filtração dos processos corretos

Para que esta resolução não deixe ambiguidade e seja o mais próximo da realidade possível, o **id** do **nodo** da pessoa do processo que estamos a avaliar (e dos irmãos que tenham o número de processo) é dado pelo nome da pessoa concatenado pelo id do processo correspondente, de forma a que este seja único.

Além disso, como também existem pais diferentes, com o mesmo nome, foi criado um **id** auxiliar que vai aumentando a cada ciclo por 1, podendo seguir a mesma lógica descrita acima: o id dos nodos dos pais é dado pelo nome do pai/mãe concatenado com esse id auxiliar. Desta forma conseguimos eliminar toda a ambiguidade relativa a este assunto.

Para fazermos o parsing correto dos Irmãos e dos seus processos, depois da chamada da função *getIrmaos*, percorremos essa lista, dando split a todos os elementos pela expressão regular : ', $Irmaos?[\backslash w]*$ $\backslash .?$' . Assim, no index 0, iremos ter sempre o nome do Irmão (ou em casos especiais iremos ter + do que 1 onde tivemos esse cuidado e separámo-los) e no index 1 iremos ter o id do processo, caso

haja. Assim, para cada irmão, iremos criar um nodo, ligar aos pais, e marcar o id do processo do Irmão (caso haja) como avaliado. Para terminar, aumentamos o id auxiliar.

```
                    if obs := re.search(r'<obs>((.|\n)*)</obs>',pr):
                        obs = obs.group(1)
                        if inf := getIrmaos(obs):

                            for i in inf:
                                i = re.split(r',Irmaos?[\w ]*\. ?',i)
                                if i[1] != '':
                                    _bid_ = getProcID(i[1])
                                    processos_avaliados.add(_bid_)
                                else :
                                    _bid_ = i[1]
                                if irmaos := re.split(r'( e |,)',i[0]):
                                    for mano in irmaos:
                                        dot.node(mano+_bid_,mano)
                                        if pai:
                                            dot.edge(mano+_bid_,pai+auxID)
                                        if mae:
                                            dot.edge(mano+_bid_,mae+auxID)
                startID += 1
```

Listing 20: Parsing dos Irmãos

Finalmente, fazemos a escrita para o ficheiro chamado *gen_trees.gv*.

```
    g = open('gen_trees.gv','w+')
    g.write(dot.source)
    g.close()

    dot.render('gen_trees.gv',view=True)
```

Listing 21: Escrita ficheiro .gv

4.6 Main Program - tp1.py

Como forma de facilitar a utilização do nosso programa e não haver a necessidade de correr separadamente cada alínea, fizemos uma *"main"* que permite ao utilizador escolher entre as diversas alíneas através do uso de um dicionário que visa simular um *swtich case*.

```



import re
import time
import query1 as q1
import query2 as q2
import query3 as q3
import query4 as q4
import query5 as q5



```

```
14 def switch(i):
15     switcher={
16         0: exit,
17         1: q1.Query1,
18         2: q2.Query2,
19         3: q3.Query3,
20         4: q4.Query4,
21         5: q5.Query5
22     }
23     return switcher.get(i,lambda: "Invalid Query")()
24
25
26
27 def main():
28
29     choice = -1
30
31     while choice != 0:
32       print('### EXERCICIO 2 ###')
33       print('1 - Calcular o numero de processos por ano; apresente a listagem
    por ordem cronolo gica e indique o intervalo de datas em que ha registos bem
     como o numero de seculos analisados.\n')
34       print('2 - Calcular a frequencia de nomes proprios (primeiro nome) e
    apelidos (ultimo nome) global e mostre os 5 mais frequentes em cada seculo.\n
    ')
35       print('3 - Calcular o numero de candidatos (nome principal de cada
    processo) que tem parentes (irmao, tio, ou primo) eclesiasticos; diga qual o
    tipo de parentesco mais frequente.\n')
36       print('4 - Verificar se o mesmo pai ou a mesma mae tem mais do que um
    filho candidato.\n')
37       print('5 - Utilizando a linguagem de desenho de grafos DOT4 desenhe todas
    as arvores genealogicas (com base nos triplos < filho, pai, mae >) dos
    candidatos referentes a um ano dado pelo utilizador.\n')
38       print('0 - Sair do programa.\n')
39
40       choice = int (input('Indique a alinea que pretende averiguar: '))
41       start = time.time()
42       switch(choice)
43
44       end = time.time()
45       time.sleep(3)
46       print()
47       print(f'Time elapsed : {end-start} seconds')
48
49 main()
```

Listing 22: Programa Principal

5 Exemplos de Outputs

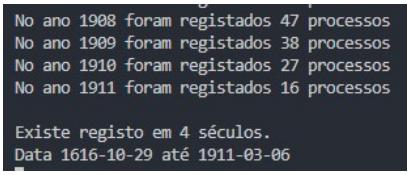


Figure 1: Exemplo de Output para a Alínea a)

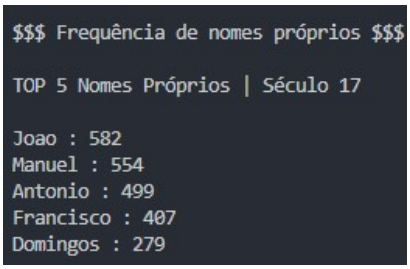


Figure 2: Exemplo de Output para a Alínea b)

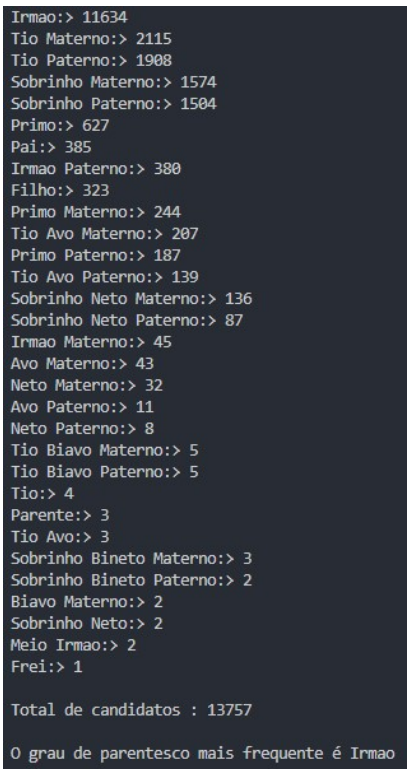


Figure 3: Exemplo de Output para a Alínea c)

```
Indique a alínea que pretende averiguar: 4
1 - Pesquisar por Pai
2 - Pesquisar por Mãe
0 - Sair
Indique a sua opção: 1
Indique o nome do Pai: Antonio Costa
O Antonio Costa do processo 8743 tem mais do que um filho.
```

Figure 4: Exemplo de Output para a Alínea d)

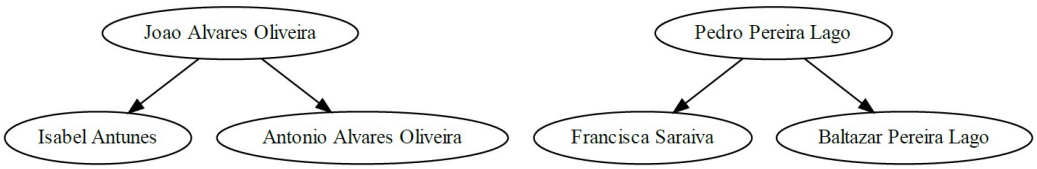


Figure 5: Exemplo de Output para a Alínea e), para o ano 1656

6 Conclusão e Análise Crítica

Perante a conclusão e apresentação da resolução a que o grupo chegou para o primeiro trabalho prático na UC de Processamento de Linguagens, é importante mencionar as diversas dificuldades e obstáculos que tivemos que ultrapassar, bem como algumas falhas que poderiam ser, e podem vir a ser, melhoradas.

Começando pelo dataset que nos foi fornecido, como mencionamos no início deste relatório, trouxe-nos bastantes pormenores que não tínhamos como controlar sem manipular diretamente o mesmo, algo que não era o intuito do exercício. Por exemplo, o facto de existirem processos diferentes com os mesmos IDs implicaria uma avaliação exaustiva de todos os parâmetros existentes em cada bloco de informação, o que tornaria o nosso programa extremamente lento. No entanto, possivelmente com mais algum conhecimento seríamos capazes de arranjar maneira de dar a volta à situação sem comprometer a eficiência, algo em que nos focámos neste trabalho. Para além disso, tivemos que ter em atenção falhas gramaticais, inexistência de referência a processos de certos parentes nas observações, a existência de pais ou mães diferentes, ou mesmo candidatos, mas com nomes iguais, o que dificultou a reoslução da alínea e), por exemplo, ou mesmo a ausência destes na informação dos processos.

No entanto, apesar de todos os contratempos e obstáculos, o trabalho prático revelou-se uma mais valia para cimentar o nosso conhecimento sobre Expressões Regulares, bem como a linguagem Python, e sendo ambos os aspetos alvo de grande interesse pelos elementos do grupo, foi sempre um trabalho que nos motivou a fazer mais e melhor, com o intuito de obter o melhor resultado possível.

Através do exercício que ao grupo competiu resolver, *Processador de Pessoas listadas nos Róis de Confessados*, consideramos que as nossas capacidades foram efetivamente comprovadas, mostrando o domínio do uso de expressões regulares, bem como a manipulação de informação com a linguagem Python.

Assim, fazemos uma avaliação deveras positiva do nosso desempenho neste trabalho, já que apesar das dificuldades encontradas , o grupo respondeu de forma autónoma e correta a todas as questões apresentadas, revelando um trabalho consistente e bem estruturado.

7 Código Utilizado

Apresentamos de seguida o código por completo, utilizado para a resolução de cada alínea.

7.1 query1.py

```
import re
import generalFunctions as gf


def Query1():
    f = open('processos.xml', 'r')

    processos_avaliados = set()
    anos = {}
    seculos = set()
    dMin = None
    dMax = None
    processos = gf.getProcessos(f.read())
    if processos is None:
        f.close()
        return


    for p in processos:
        pr = p[0]
        if _id_ := gf.getId(pr):
            if _id_ in processos_avaliados:
                pass
            else:
                if data := re.search(r'<data>((\d{4})-\d{2}-\d{2})</data>',pr):
                    dt = data.group(2)
                    if lista := anos.get(dt):
                        lista.add(_id_)
                    else:
                        aux = set()
                        aux.add(_id_)
                        anos.update({dt : aux})
                    seculos.add(gf.getSeculo(dt))

                if dMin is None:
                    dMin = data.group(1)
                if dMax is None:
                    dMax = data.group(1)
                if data.group(1) < dMin:
                    dMin = data.group(1)
                if data.group(1) > dMax:
                    dMax = data.group(1)

    anos_sorted = dict(sorted(anos.items(), key=lambda p:p[0]))



```

```
47      for a in anos_sorted.keys():
48          print(f'No ano {a} foram registados {len(anos_sorted.get(a))} processos'
    )
49
50      print(f'\nExiste registo em {len(seculos)} s culos.')
51
52      print(f'Data {dMin} ate {dMax}')
53
54      f.close()
```

Listing 23: Resolução da alínea a)

7.2 query2.py

```
import re
import generalFunctions as gf

def Query2():

    """

    proprios = {
        "Maria": {
            19:4
            18:3
            17:0
        }
        "Paulo" : {

        }

    }
    apelidos = {

    }

    """

    processos_avaliados = set()
    proprios = {}
    apelidos = {}

    maxSec = 0
    minSec = 30

    f = open('processos.xml','r')
    processos = gf.getProcessos(f.read())
    for p in processos:
        pr = p[0]
        if _id_ := gf.getId(pr):
            if _id_ in processos_avaliados:
                #Ja foi avaliado, caso de datasets repetidos
                pass
            else:
                #Processo ainda nao foi avaliado
                processos_avaliados.add(_id_)
                if _data_ := re.search(r'<data>((\d{4})-\d{2}-\d{2})</data>',pr)
:
                    ano = _data_.group(2)
                    sec = gf.getSeculo(ano)
                    if sec < minSec:
                        minSec = sec
                    if sec > maxSec:
                        maxSec = sec

                    if _nomeComp_ := re.search(r'<nome>(((\w+)[ .\t]*)+)</nome>'
```

```
,pr):
                        _nomeComp_ = _nomeComp_.group(1)
                        _proprio_ = _nomeComp_.split(r' ')[0]
                        _apelido_ = _nomeComp_.split(r' ')[-1]

                        #Atualizar Dicionario de Nomes Proprios
                        if p := proprios.get(_proprio_):
                            #Se ja estiver registado
                            if p_dict_sec := p.get(sec):
                                #Se ja existir seculo registado
                                p.update({sec:p_dict_sec+1})
                            else:
                                #Caso o seculo ainda nao tenha sido registado
                                p.update({sec:1})
                        else:
                            #Caso o nome ainda nao tenha sido registado
                            proprios.update({_proprio_: {sec:1}})

                        #Atualizar Dicionarios de Apelidos
                        if a := apelidos.get(_apelido_):
                            #Se ja estiver registado
                            if a_dict_sec := a.get(sec):
                                #Se ja existir seculo registado
                                a.update({sec : a_dict_sec+1})
                            else:
                                #Caso o seculo ainda nao tenha sido registado
                                a.update({sec : 1})
                        else:
                            #Caso o apelido ainda nao tenha sido registado
                            apelidos.update({_apelido_: {sec:1}})

    proprios = dict(sorted(proprios.items(), key=lambda p:p[0]))
    apelidos = dict(sorted(apelidos.items(), key=lambda p:p[0]))

    print("### Frequencia de nomes proprios GLOBAL ###")
    # Print de Nomes Proprios
    for n in proprios:
        count = 0
        for sec in proprios.get(n):
            count += int(proprios.get(n).get(sec))
        print(f'{n} :> {count}')

    print('\n######################\n')

    print("### Frequencia de apelidos GLOBAL ###")
    # Print de Apelidos
    for n in apelidos:
        count = 0
        for sec in apelidos.get(n):
            count += int(apelidos.get(n).get(sec))
        print(f'{n} :> {count}')

    print('\n######################\n')

```

```
104     print("$$$ Frequencia de nomes proprios $$$")
105     for sec in range(minSec,maxSec+1):
106         listProprios = []
107         for nome in proprios:
108         # print de nomes proprios
109             if count := proprios.get(nome).get(sec):
110                 listProprios.append((nome,count))
111 
112         listProprios = sorted(listProprios,key = lambda x : x[1],reverse=True)
113         print(f'\nTOP 5 Nomes Proprios | Seculo {sec}\n')
114         for a in range(5):
115             print(f'{listProprios[a][0]} : {listProprios[a][1]}')
116 
117 
118 
119     print('\n######################\n')
120 
121 
122     print("$$$ Frequencia de apelidos $$$")
123     # print de apelidos
124     for sec in range(minSec,maxSec+1):
125         listApelidos = []
126         for nome in apelidos:
127         # print de apelidos
128             if count := apelidos.get(nome).get(sec):
129                 listApelidos.append((nome,count))
130 
131         listApelidos = sorted(listApelidos,key = lambda x : x[1],reverse=True)
132         print(f'\nTOP 5 Apelidos | Seculo {sec}\n')
133         for a in range(5):
134             print(f'{listApelidos[a][0]} : {listApelidos[a][1]}')
135 
136 
137 
138     f.close()
```

Listing 24: Resolução da alínea b)

7.3 query3.py

```
import re
import generalFunctions as gf


def Query3():
    f = open('processos.xml', 'r')

    processos = gf.getProcessos(f.read())

    processos_avaliados = set()

    contCandidatos = 0
    accountedFor = False
    graus = {}

    for p in processos:
        pr = p[0]

        if _id_ := gf.getId(pr):
            if _id_ in processos_avaliados:
                pass
            else:
                processos_avaliados.add(_id_)

                if _obs_ := re.search(r'<obs>((.|\n)+?)</obs>',pr):

                    obsConteudo = _obs_.group(1)
                    obsConteudo = re.sub(r'\s+',r' ',obsConteudo)

                    obsConteudo_splited = re.split(r'\. ?Proc.\d+\. ?',
    obsConteudo)

                    accountedFor = False

                    for element in obsConteudo_splited:
                        obsConteudo_splited2 = re.split(r'\.',element)


                        for elem in obsConteudo_splited2:
                            if m:= re.search(r'(([A-Z]\w+( e| e | |,)?)+) ?,((([
    A-Z]\w+) ?)*)',elem):
                                g = m.group(4)

                                if re.search(r'(?i:(sao|san|nos))',g):
                                    pass

                                elif g == '':
                                    pass

                                else:
                                    if not accountedFor:
                                        contCandidatos += 1
```

```
                                        accountedFor = True

                                    g_sing = re.sub(r's',r'',g)

                                    if count := graus.get(g_sing):
                                        graus.update({g_sing : count+1})

                                    else:
                                        graus.update({g_sing : 1})

    f.close()

    graus = dict(sorted(graus.items(), key=lambda p:p[1],reverse=True))

    for grau in graus.keys():
        print(f'{grau}:> {graus.get(grau)}')

    print(f'\nTotal de candidatos : {contCandidatos}')
    print(f'\nO grau de parentesco mais frequente e {list(graus.keys())[0]}')
```

Listing 25: Resolução da alínea c)

7.4 query4.py

```
import re
import generalFunctions as gf

def procuraMae(mae):
    f=open('processos.xml','r')
    processos = gf.getProcessos(f.read())

    processos_avaliados = set()


    for p in processos:
        pr = p[0]

        if _id_:= gf.getId(pr):

            if _id_ in processos_avaliados:
                pass
            else:
                processos_avaliados.add(_id_)

                if re.search(rf'<mae>{mae}</mae>',pr):
                    if m := re.search(r'<obs>((.|\n)*)</obs>',pr):
                        i = m.group(1)
                        if re.search(r'(?i:(irmao|irmaos))',i):
                            print(f'A {mae} do processo {_id_} tem mais do que um filho.')
                            return


    print(f'{mae} nao tem mais que um filho.')
    f.close()




def procuraPai(pai):
    processos_avaliados = set()
    f = open('processos.xml', 'r')
    processos = gf.getProcessos(f.read())


    for p in processos:
        pr = p[0]

        if _id_ := gf.getId(pr):

            if _id_ in processos_avaliados:
                pass
            else:
                processos_avaliados.add(_id_)

                if re.search(rf'<pai>{pai}</pai>',pr):
```

```
51                      if m := re.search(r'<obs>((.|\n)*)</obs>',pr):
52                          i = m.group(1)
53                          if re.search(r'(?i:(irmao|irmaos))',i):
54                              print(f'O {pai} do processo {_id_} tem mais do que
    um filho.')
55                              return
56
57      print(f'{pai} nao tem mais que um filho.')
58      f.close()
59
60  def Query4():
61
62      opcao = -1
63      while opcao != 0:
64          print('1 - Pesquisar por Pai')
65          print('2 - Pesquisar por Mae')
66          print('0 - Sair')
67          opcao = int(input('Indique a sua opcao: '))
68
69          if opcao == 1:
70              pai = input('Indique o nome do Pai: ')
71              procuraPai(pai)
72          elif opcao == 2:
73              mae = input('Indique o nome da Mae: ')
74              procuraMae(mae)
75          elif opcao < 0 and opcao > 2:
76              print("Opcao inexistente.\nTry again.")
```

Listing 26: Resolução da alínea d)

7.5 query5.py

```
import re
import generalFunctions as gf
from graphviz import Digraph


def getProcID(proc):
    if m := re.search(r' ?Proc\.(\d+)\.?',proc):
        return m.group(1)

def getIrmaos(obs):
    l = []
    if irmaos := re.findall(r'([\w ,]*,Irmao(s)?[\w ]*\.( ?Proc\.\d+\.)?)',obs):
        for i in irmaos:
            l.append(i[0].strip())
    return l

def Query5():
    startID = 0
    dot = Digraph(comment="Gen_Trees")

    processos_avaliados = set()
    f = open('processos.xml', 'r')

    ano = input('Indique o ano que pretende investigar: ')

    processos = gf.getProcessosByData(f.read(), ano)

    for pr in processos:
        if _id_ := gf.getId(pr):
            if _id_ in processos_avaliados:
                pass
            else:
                auxID = str(startID)
                if nome := re.search(r'<nome>((.|\n)*)</nome>',pr):
                    nome = nome.group(1)
                    processos_avaliados.add(_id_)
                    dot.node(nome+_id_,nome)

                    mae = re.search(r'<mae>((.|\n)*)</mae>',pr)
                    if mae:
                        mae = mae.group(1)
                        dot.node(mae+auxID,mae)
                        dot.edge(nome+_id_,mae+auxID)
                    pai = re.search(r'<pai>((.|\n)*)</pai>',pr)
                    if pai:
                        pai = pai.group(1)
                        dot.node(pai+auxID,pai)
                        dot.edge(nome+_id_,pai+auxID)

                    if obs := re.search(r'<obs>((.|\n)*)</obs>',pr):
                        obs = obs.group(1)
                        if inf := getIrmaos(obs):
```

```
52
53                              for i in inf:
54                                  i = re.split(r',Irmaos?[\w ]*\. ?',i)
55                                  if i[1] != '':
56                                      _bid_ = getProcID(i[1])
57                                      processos_avaliados.add(_bid_)
58                                  else :
59                                      _bid_ = i[1]
60                                  if irmaos := re.split(r' e ',i[0]):
61                                      for mano in irmaos:
62                                          dot.node(mano+_bid_,mano)
63                                          if pai:
64                                              dot.edge(mano+_bid_,pai+auxID)
65                                          if mae:
66                                              dot.edge(mano+_bid_,mae+auxID)
67                  startID += 1
68
69      g = open('gen_trees.gv','w+')
70      g.write(dot.source)
71      g.close()
72
73      dot.render('gen_trees.gv',view=True)
74
75      f.close()
```

Listing 27: Resolução da alínea e)