

UNIVERSIDADE DO MINHO

DEPARTAMENTO DE INFORMÁTICA

Processamento de Linguagens
Trabalho Prático nº 1

João Freitas (A83782) Luís Fernandes (A76712)
Rui Fernandes (A89138)

Abril 2021

Resumo

O presente relatório descreve o trabalho prático realizado no âmbito da disciplina de *Processamento de Linguagens*, ao longo do segundo semestre do terceiro ano do Mestrado Integrado em Engenharia Informática da Universidade do Minho.

A realização deste trabalho prático tem como principal objetivo o processamento de um ficheiro XML de forma a extrair todos os dados considerados relevantes, ficheiro esse que contém o registo de rapazes que pretendiam seguir a vida clerical e se candidatavam aos seminários. O processamento deste ficheiro será feito através de Expressões Regulares para a descrição de padrões de frases dentro de textos.

Neste documento descrevemos sucintamente o programa desenvolvido e discutimos as decisões tomadas durante a realização do trabalho prático.

Conteúdo

1	Introdução	1
1.1	Enquadramento e Contexto	1
1.2	Problema e Objetivo	1
2	Concepção da Solução	2
2.1	Estruturas de Dados	2
2.2	Algoritmos	4
2.2.1	Alínea a	4
2.2.2	Alínea b	5
2.2.3	Alínea c	7
2.2.4	Alínea d	8
2.2.5	Alínea e	9
3	Codificação e Testes	10
3.1	Testes realizados e Resultados	10
3.1.1	Alínea a	10
3.1.2	Alínea b	12
3.1.3	Alínea c	13
3.1.4	Alínea d	14
3.1.5	Alínea e	15
4	Conclusão	16

Lista de Figuras

1	Número de processos por ano	10
2	Número de processos por ano	11
3	Número de processos por ano, num intervalo de tempo	11
4	Frequência global de nomes próprios	12
5	Frequência global de apelidos	12
6	Nomes próprios e apelidos mais comuns em cada século	13
7	Número de candidatos com parentes eclesiásticos	13
8	Graus de parentesco e respetiva frequência	14
9	Pai com no máximo um filho candidato	14
10	Mãe com mais do que um filho candidato	15
11	Árvores genealógicas dos candidatos referentes ao ano de 1836	15

1 Introdução

1.1 Enquadramento e Contexto

Os *Róis de Confessados* são arquivos do arcebispado existentes no Arquivo Digital de Braga que contém o registo de rapazes que pretendiam seguir a vida clerical e se candidatavam aos seminários, contendo milhares de registos. Cada um desses registos contém informação relativa ao nome do candidato e dos seus pais, assim como eventuais parentes que apadrinhem a sua candidatura.

O trabalho apresentado tem como objetivo usar o módulo `re` da linguagem `Python` para filtrar um conjunto de dados fornecidos, extraíndo desses dados a informação relevante.

Adicionalmente, recorrendo ao módulo `pyplot` da biblioteca `matplotlib`, assim como à biblioteca `graphviz`, foi possível gerar alguns gráficos que permitem uma melhor visualização dos dados, tal como será apresentado na secção 3.

1.2 Problema e Objetivo

Dado um ficheiro XML com alguns milhares de registos relativos a candidaturas aos seminários pretende-se desenvolver um programa em `Python` de modo a:

- Calcular o número de processos por ano;
- Calcular a frequência de nomes próprios e apelidos;
- Calcular o número de candidatos que têm parentes eclesiásticos;
- Verificar se mesmo pai ou a mesma mãe têm mais do que um filho candidato;
- Desenhar todas as árvores genealógicas dos candidatos referentes a um determinado ano.

Desta forma, numa primeira fase, o problema passa por limpar e normalizar todos estes registos. Em seguida, será necessário criar estruturas de dados adequadas para armazenar esta informação para processá-la posteriormente.

Estrutura do Relatório

O presente relatório encontra-se dividido em 4 partes.

No capítulo 1, Introdução, é feito um enquadramento e contextualização do trabalho prático e, em seguida, é feita uma descrição do problema.

No capítulo 2, Concepção da Solução, são expostas as estruturas de dados utilizadas e é feita uma descrição detalhada de todo o desenvolvimento do projeto até se obter a solução final.

De seguida, no capítulo 3, Codificação e Testes, são apresentados alguns testes realizados e os resultados obtidos.

Por fim, no capítulo 4, Conclusão, termina-se o relatório com uma síntese a análise crítica do trabalho desenvolvido.

2 Conceção da Solução

Dado um ficheiro XML com milhares de registos, é necessário extrair de cada candidatura toda a informação relevante e, posteriormente, inseri-la em estruturas de dados adequadas.

A seguir apresenta-se um exemplo da forma como um processo é representado. Note-se, no entanto, que alguns destes campos podem estar em falta.

```
1 <processo id="13123">
2   <pasta>569</pasta>
3   <data>1869-12-02</data>
4   <nome>Abilio Augusto Santos</nome>
5   <pai>Jose Joaquim Santos</pai>
6   <mae>Teresa Jesus</mae>
7   <obs>Antonio Jose Adao, Tio Materno. Proc.12530. Albino Antonio
      Ribeiro, Primo Paterno. Proc.12721.</obs>
8 </processo>
```

2.1 Estruturas de Dados

Para conseguirmos armazenar temporariamente a informação relativa a uma candidatura, existiu a necessidade de construir uma estrutura de dados que nos permitisse guardar os diferentes atributos. Nesse sentido, foi criada a classe **Processo**, que tem todas as variáveis necessárias para guardar a informação de uma candidatura.

```
1 class Processo:
2     def __init__(self, id, data, nome, pai, mae, obs):
3         self.id = id
4         self.data = data
5         self.nome = nome
6         self.pai = pai
7         self.mae = mae
8         self.obs = obs
9
10    def __eq__(self, o: object) -> bool:
11        return self.__class__ == o.__class__ and self.id == o
12            .id and self.data == o.data and self.nome == o.nome and \
13                self.pai == o.pai and self.mae == o.mae and self.obs == o.
14                    obs
15
16    def __hash__(self) -> int:
17        return hash(self.id)
```

De forma a processar o ficheiro XML contendo a informação relativa aos processos, foi desenvolvida uma função **parse_xml** responsável por ler o ficheiro, instanciar os processos como objetos da classe **Processo** e, posteriormente, adicioná-los a um dicionário. Este dicionário, em que cada século está associado um outro dicionário no qual a cada ano corresponde uma lista contendo os processos relativos a esse ano, permite um fácil acesso aos processos de um determinado intervalo de tempo.

Note-se que uma vez que existem entradas repetidas no ficheiro XML, estas foram removidas de forma a não tornar o resultado redundante.

```
1 def remove_duplicates(processos_dict):
2     for (sec, anos_dict) in processos_dict.items():
3         for (ano, _) in anos_dict.items():
4             processos_dict[sec][ano] = list(set(processos_dict[sec][ano])
5             )
6     return processos_dict
7
8 def parse_xml(lines):
9     processos_dict = {}
10    processos_XML = re.search(r'<processos>\s*(<processo id[\S\s]*</
11    processo>)\s*</processos>', lines)
12    processos = re.findall(r'<processo id[\S\s]*?</processo>',
13    processos_XML.group(1))
14    for p in processos:
15        id = re.search(r'<processo id="(\d+)">', p).group(1)
16        data = re.search(r'<data>(\d{4}-\d{2}-\d{2})</data>', p).group(1)
17        nome = re.search(r'<nome>(.*?)</nome>', p).group(1)
18        pai = re.search(r'<pai>([\w\s]*)\s*?</pai>', p)
19        if pai:
20            pai = pai.group(1)
21        else:
22            pai = ''
23        mae = re.search(r'<mae>([\w\s]*)\s*?</mae>', p)
24        if mae:
25            mae = mae.group(1)
26        else:
27            mae = ''
28        obs = re.search(r'<obs>(.*?)</obs>', p)
29        if obs:
30            obs = obs.group(1)
31        else:
32            obs = ''
33
34        ano = int(data[:4])
35        seculo = ano // 100 + 1
36
37        if seculo not in processos_dict.keys():
38            processos_dict[seculo] = {}
39
40        try:
41            processos_dict[seculo][ano].append(Processo(id, data, nome,
42            pai, mae, obs))
43        except KeyError:
44            processos_dict[seculo][ano] = []
45            processos_dict[seculo][ano].append(Processo(id, data, nome,
46            pai, mae, obs))
47
48    return remove_duplicates(processos_dict)
```

2.2 Algoritmos

2.2.1 Alínea a

Estando todos os processos já inseridos em dicionários organizado por séculos e anos, para determinar quantos processos existem em cada ano basta calcular o tamanho da lista de processos relativa a esse ano. Em relação ao número de séculos, devolve-se o tamanho do dicionário que, por sua vez, contém os dicionários relativos a cada um dos anos.

Por outro lado, para determinar o intervalo de datas em que há registos, começa-se por determinar a menor chave do dicionário de séculos, que corresponde ao menor século, e, com essa chave, determina-se a menor chave do dicionário de anos associado a essa chave, que corresponde ao menor ano. Por fim, tendo a lista de processos associada a esse ano, basta ordenar a lista tendo em conta a data de cada um dos processos. Desta forma, a data do primeiro processo corresponderá, então, à data do primeiro registo.

Num processo idêntico, para determinar a data do último registo calcula-se a maior chave do dicionário de séculos e, posteriormente, determina-se a maior chave do dicionário de anos associado a essa chave, que corresponderá ao maior ano. Assim, ordenando a lista tendo em consideração a data de cada um dos processos, basta verificar qual a data do processo mais recente.

```
1 def alinea_a(processos_dict):
2     x = []
3     y = []
4
5     for anos_dict in sorted(processos_dict.values(), key=lambda item:
6         sorted(item)):
7         for (ano, p) in sorted(anos_dict.items()):
8             print(f'{ano}: {len(p)} processos')
9             x.append(ano)
10            y.append(len(p))
11
12    print(f'Foram analisados {len(processos_dict)} seculos')
13
14    min_seculo = min(processos_dict.keys())
15    min_ano = min(processos_dict[min_seculo].keys())
16    min_date = sorted(processos_dict[min_seculo][min_ano], key=lambda p:
17        p.data)[0].data
18
19
20    max_seculo = max(processos_dict.keys())
21    max_ano = max(processos_dict[max_seculo].keys())
22    max_date = sorted(processos_dict[max_seculo][max_ano], key=lambda p:
23        p.data, reverse=True)[0].data
24
25
26    print(f'Existem processos entre {min_date} e {max_date}')
27
28    plt.style.use('seaborn')
29    plt.bar(x, y, linestyle='solid', color='green')
30    plt.title('Numero de Processos por Ano')
31    plt.ylabel('Numero de Processos')
32    plt.xlabel('Ano')
33    plt.gcf().autofmt_xdate()
34    plt.tight_layout()
35    plt.show()
```

2.2.2 Alínea b

Para determinar a frequência global de nomes próprios e apelidos, itera-se sobre todos os processos guardados e para cada nome no processo guarda-se o primeiro nome como chave num dicionário tendo como valor a contagem atual desse nome. Este processo é repetido para o último nome do titular do processo, noutro dicionário.

```
1 def alinea_b(processos_dict):
2     nomes_proprios = {}
3     apelidos = {}
4
5     for (sec, anos_dict) in processos_dict.items():
6         for (ano, proc) in anos_dict.items():
7             for p in proc:
8                 nome = re.split(r'\s+', p.nome)
9                 nome proprio = nome[0]
10                apelido = nome[-1]
11                try:
12                    nomes_proprios[nome proprio] += 1
13                except KeyError:
14                    nomes_proprios[nome proprio] = 1
15                try:
16                    apelidos[apelido] += 1
17                except KeyError:
18                    apelidos[apelido] = 1
19
20    print('GLOBAL')
21    print('\nNomes Proprios\n')
22    for (nome, freq) in list(sorted(nomes_proprios.items(), key=lambda
23        item: item[1], reverse=True)):
24        print(f'{nome} -> {freq}')
25    print('\nApelidos\n')
26    for (nome, freq) in list(sorted(apelidos.items(), key=lambda item:
27        item[1], reverse=True)):
28        print(f'{nome} -> {freq}')
29
30    print('\nPOR SÉCULO')
31    for sec in sorted(processos_dict.keys()):
32        print(f'\nSéculo {sec}')
33        (nome, apelido) = nome_apelido_seculo(sec, processos_dict)
34        print(f'Nomes: {", ".join([x[0] for x in nome])}')
35        print(f'Apelidos: {", ".join([x[0] for x in apelido])}')
```

Além disso, para obter os 5 nomes próprios e apelidos mais frequentes em cada um dos séculos, repete-se a iteração sobre os processos, mas desta vez apenas considerando os referentes a cada século, e são retornados os 5 nomes e apelidos mais frequentes.

```
1 def nome_apelido_seculo(seculo , processos_dict):
2     nomes_proprios = {}
3     apelidos = {}
4     for proc in processos_dict[seculo].values():
5         for p in proc:
6             nome = re.split(r'\s+', p.nome)
7             nome proprio = nome[0]
8             apelido = nome[-1]
9             try:
10                nomes_proprios[nome proprio] += 1
11            except KeyError:
12                nomes_proprios[nome proprio] = 1
13            try:
14                apelidos[apelido] += 1
15            except KeyError:
16                apelidos[apelido] = 1
17     nome = list(sorted(nomes_proprios.items(), key=lambda item: item[1],
18                        reverse=True))[:5]
19     apelido = list(sorted(apelidos.items(), key=lambda item: item[1],
20                          reverse=True))[:5]
21     return nome, apelido
```

2.2.3 Alínea c

Para calcular o número de candidatos com parentes eclesiásticos (irmão, tio ou primo), itera-se sobre todos os processos e no campo relativo às observações de cada um dos processo são procuradas todas as ocorrências das palavras irmão, tio ou primo, através de expressões regulares. Caso existam ocorrências, é incrementado o contador de candidatos com parentes eclesiásticos, assim como o contador da respetiva ocorrência do grau de parentesco. Por fim, é retornado o número de candidatos com parentes eclesiásticos bem como o parentesco mais frequente.

```
1 def alinea_c(processos_dict):
2     parentes = {'Irmão': 0, 'Tio': 0, 'Primo': 0}
3     r = 0
4     for (sec, anos_dict) in processos_dict.items():
5         for (ano, proc) in anos_dict.items():
6             for p in proc:
7                 tem_parente = False
8                 if irmaos := re.findall(r'(?i:irmaos?)', p.obs):
9                     parentes['Irmão'] += len(irmaos)
10                    tem_parente = True
11                 if tios := re.findall(r'(?i:tio)', p.obs):
12                     parentes['Tio'] += len(tios)
13                    tem_parente = True
14                 if primos := re.findall(r'(?i:primo)', p.obs):
15                     parentes['Primo'] += len(primos)
16                    tem_parente = True
17                 if tem_parente:
18                     r += 1
19     print(f'{r} candidatos tem parentes eclesiásticos')
20     print(
21         f'O grau de parentesco mais comum é {list(sorted(parentes.items()
22             , key=lambda item: item[1], reverse=True))[0][0]}')
23
24     x = list(parentes.keys())
25     y = list(parentes.values())
26     plt.style.use('seaborn')
27     plt.bar(x, y, linestyle='solid', color='green')
28     plt.title('Grau de Parentesco Mais Comum')
29     plt.xlabel('Grau de Parentesco')
30     plt.ylabel('Frequencia')
31     plt.tight_layout()
32     plt.show()
```

2.2.4 Alínea d

De modo a verificar se o mesmo pai ou a mesma mãe têm mais do que um filho candidato, itera-se sobre todos os processos e incrementa-se o valor de um contador caso o progenitor do titular do processo em questão corresponda ao nome pretendido. Quando for encontrado mais do que um processo cujo titular seja descendente do parente em questão, a iteração termina. Caso contrário, a iteração continua e, caso o ciclo termine, conclui-se, então, que o progenitor tem, no máximo, um filho candidato à vida clerical.

```
1 def alinea_d_pai(pai, processos_dict):
2     n = 0
3     for (sec, anos_dict) in processos_dict.items():
4         for (ano, proc) in anos_dict.items():
5             for p in proc:
6                 if p.pai == pai:
7                     n += 1
8                     if n > 1:
9                         print(f'{pai} tem mais do que um filho candidato '
10                              )
11                         return
12     print(f'{pai} nao tem mais do que um filho candidato')
13
14 def alinea_d_mae(mae, processos_dict):
15     n = 0
16     for (sec, anos_dict) in processos_dict.items():
17         for (ano, proc) in anos_dict.items():
18             for p in proc:
19                 if p.mae == mae:
20                     n += 1
21                     if n > 1:
22                         print(f'{mae} tem mais do que um filho candidato '
23                              )
24                         return
25     print(f'{mae} nao tem mais do que um filho candidato')
```

2.2.5 Alínea e

Para construir todas as árvores genealógicas referentes a um determinado ano, itera-se sobre todos os processos referentes a esse mesmo ano e, com base na linguagem de desenho de grafos DOT, desenham-se nodos para o titular do processo e para os progenitores, assim como um arco entre eles. Note-se, no entanto, que alguns dos processos não possuem informação relativa a um dos progenitores.

Por fim, recorrendo à biblioteca **graphviz**, é possível visualizar o grafo gerado.

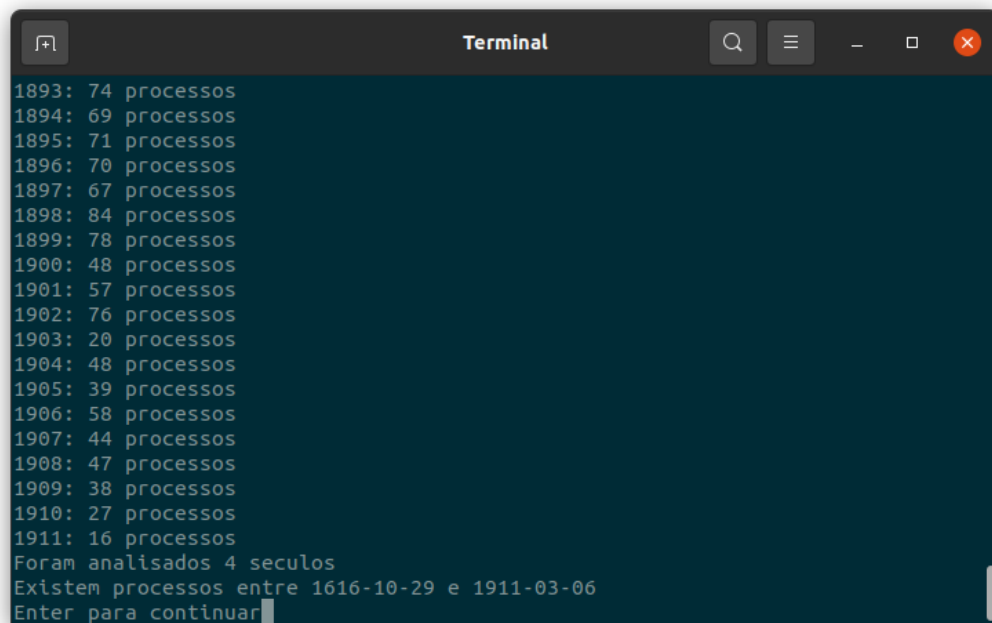
```
1 def alinea_e(ano, processos_dict):
2     seculo = ano // 100 + 1
3     f = open('arvore-genealogica.dot', 'w')
4     f.write('digraph arvore-genealogica {\n')
5     try:
6         for p in processos_dict[seculo][ano]:
7             if p.mae:
8                 f.write(f'\t{p.mae.replace(" ", "")} -> {p.nome.replace(" ", "")};\n')
9             if p.pai:
10                f.write(f'\t{p.pai.replace(" ", "")} -> {p.nome.replace(" ", "")};\n')
11        f.write('}\n')
12        f.close()
13        src = Source.from_file('arvore-genealogica.dot')
14        src.view()
15    except KeyError:
16        print('Nao existe informacao sobre esse ano')
```

3 Codificação e Testes

3.1 Testes realizados e Resultados

Ao longo da realização do trabalho prático fomos testando os métodos elaborados através de diferentes testes. Mostram-se a seguir alguns testes feitos e os respectivos resultados obtidos.

3.1.1 Alínea a



```
Terminal
1893: 74 processos
1894: 69 processos
1895: 71 processos
1896: 70 processos
1897: 67 processos
1898: 84 processos
1899: 78 processos
1900: 48 processos
1901: 57 processos
1902: 76 processos
1903: 20 processos
1904: 48 processos
1905: 39 processos
1906: 58 processos
1907: 44 processos
1908: 47 processos
1909: 38 processos
1910: 27 processos
1911: 16 processos
Foram analisados 4 seculos
Existem processos entre 1616-10-29 e 1911-03-06
Enter para continuar
```

Figura 1: Número de processos por ano

Adicionalmente, com o intuito de permitir uma melhor visualização dos dados, e fazendo uso do módulo `pyplot` da biblioteca `matplotlib`, foram gerados alguns gráficos que se apresentam de seguida.

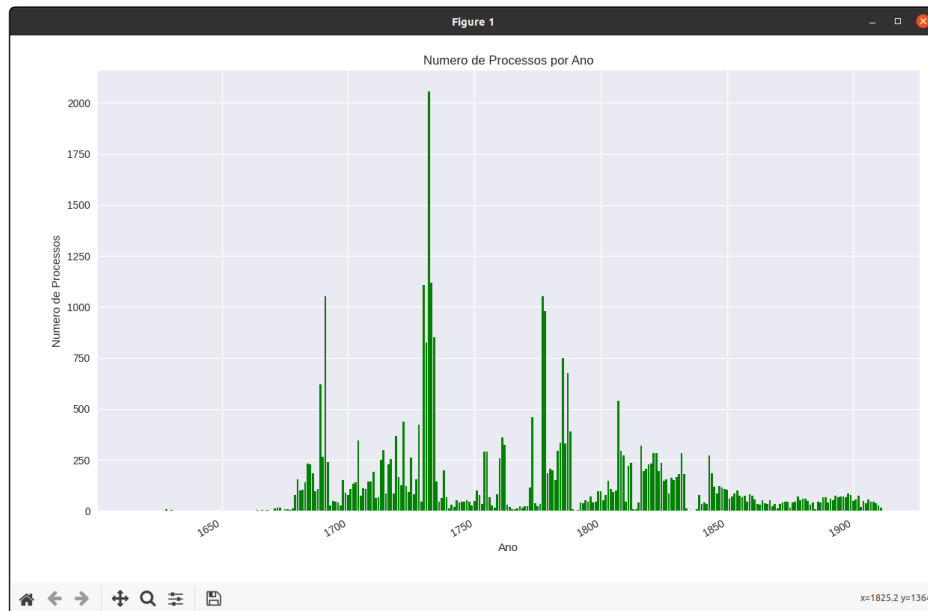


Figura 2: Número de processos por ano

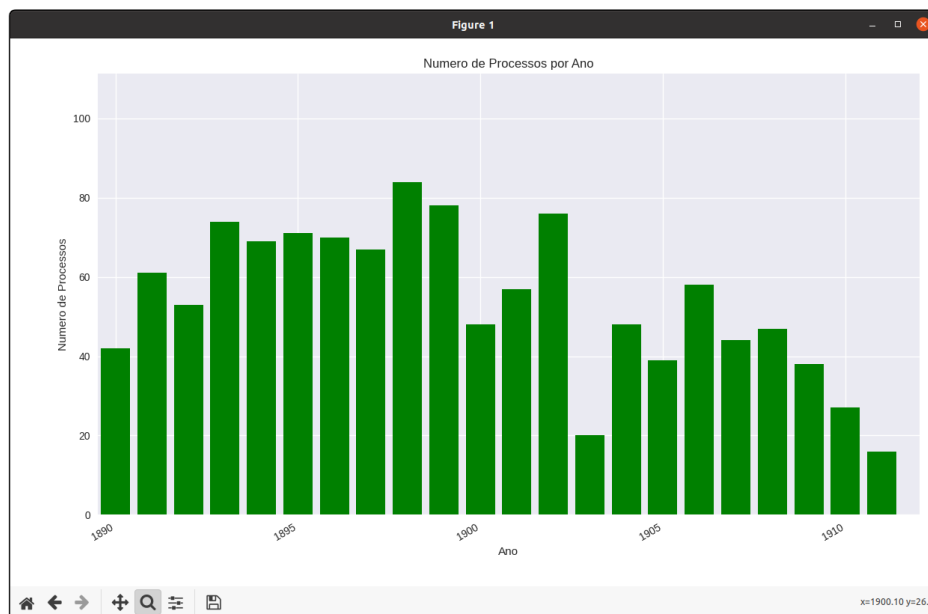
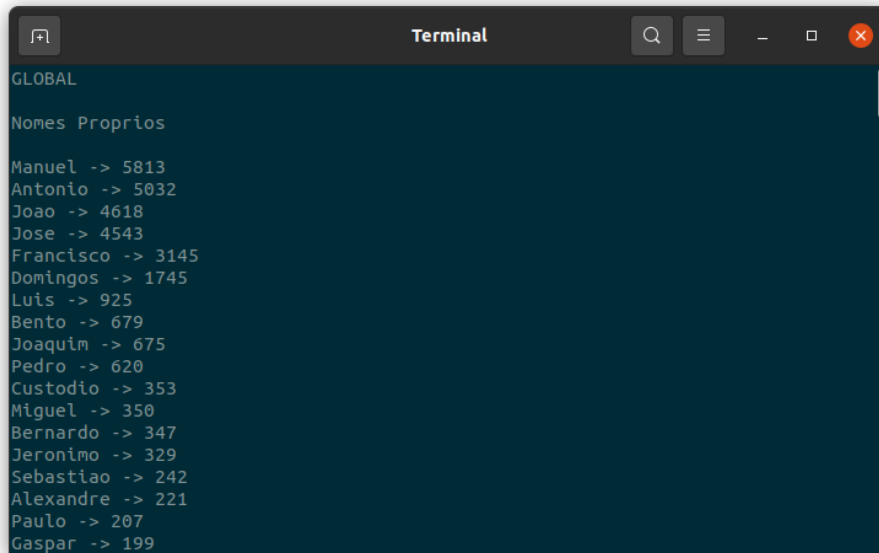


Figura 3: Número de processos por ano, num intervalo de tempo

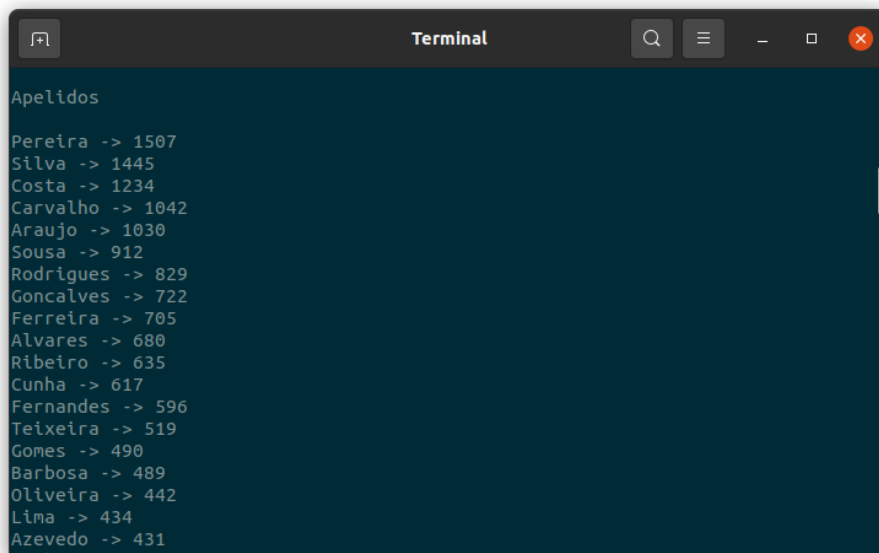
3.1.2 Alínea b



A terminal window titled "Terminal" with a dark background and light text. It displays the results of a command, showing a list of proper names and their corresponding frequencies. The window has standard macOS window controls (zoom, search, menu, window, close) in the title bar.

```
GLOBAL  
  
Nomes Proprios  
  
Manuel -> 5813  
Antonio -> 5032  
Joao -> 4618  
Jose -> 4543  
Francisco -> 3145  
Domingos -> 1745  
Luis -> 925  
Bento -> 679  
Joaquim -> 675  
Pedro -> 620  
Custodio -> 353  
Miguel -> 350  
Bernardo -> 347  
Jeronimo -> 329  
Sebastiao -> 242  
Alexandre -> 221  
Paulo -> 207  
Gaspar -> 199
```

Figura 4: Frequência global de nomes próprios



A terminal window titled "Terminal" with a dark background and light text. It displays the results of a command, showing a list of surnames and their corresponding frequencies. The window has standard macOS window controls (zoom, search, menu, window, close) in the title bar.

```
Apelidos  
  
Pereira -> 1507  
Silva -> 1445  
Costa -> 1234  
Carvalho -> 1042  
Araujo -> 1030  
Sousa -> 912  
Rodrigues -> 829  
Goncalves -> 722  
Ferreira -> 705  
Alvares -> 680  
Ribeiro -> 635  
Cunha -> 617  
Fernandes -> 596  
Teixeira -> 519  
Gomes -> 490  
Barbosa -> 489  
Oliveira -> 442  
Lima -> 434  
Azevedo -> 431
```

Figura 5: Frequência global de apelidos

```
Terminal

POR SEculo

Seculo 17
Nomes: Joao, Manuel, Antonio, Francisco, Domingos
Apelidos: Silva, Pereira, Costa, Araujo, Rodrigues

Seculo 18
Nomes: Manuel, Joao, Antonio, Jose, Francisco
Apelidos: Pereira, Silva, Costa, Carvalho, Araujo

Seculo 19
Nomes: Jose, Antonio, Manuel, Joao, Francisco
Apelidos: Pereira, Silva, Costa, Sousa, Carvalho

Seculo 20
Nomes: Antonio, Jose, Manuel, Joao, Joaquim
Apelidos: Silva, Costa, Oliveira, Pereira, Junior
Enter para continuar
```

Figura 6: Nomes próprios e apelidos mais comuns em cada século

3.1.3 Alínea c

```
Terminal

9827 candidatos tem parentes eclesiasticos
O grau de parentesco mais comum e Irmão
Enter para continuar
```

Figura 7: Número de candidatos com parentes eclesiásticos

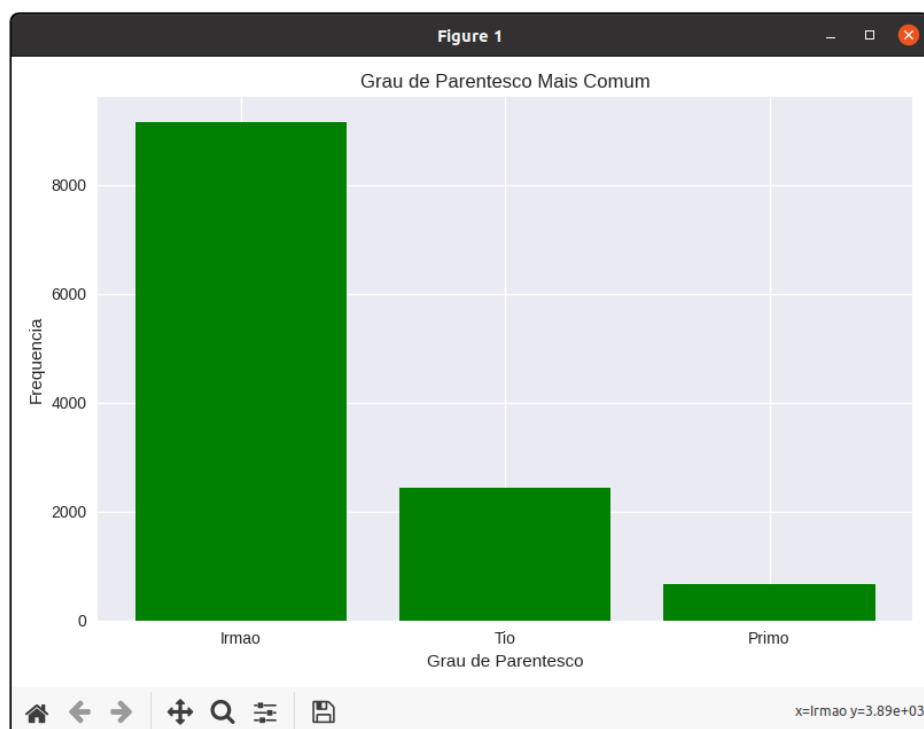


Figura 8: Graus de parentesco e respetiva frequência

3.1.4 Alínea d

```
Terminal
Pai: Antonio Manuel Almeida
Antonio Manuel Almeida nao tem mais do que um filho candidato
Enter para continuar
```

Figura 9: Pai com no máximo um filho candidato

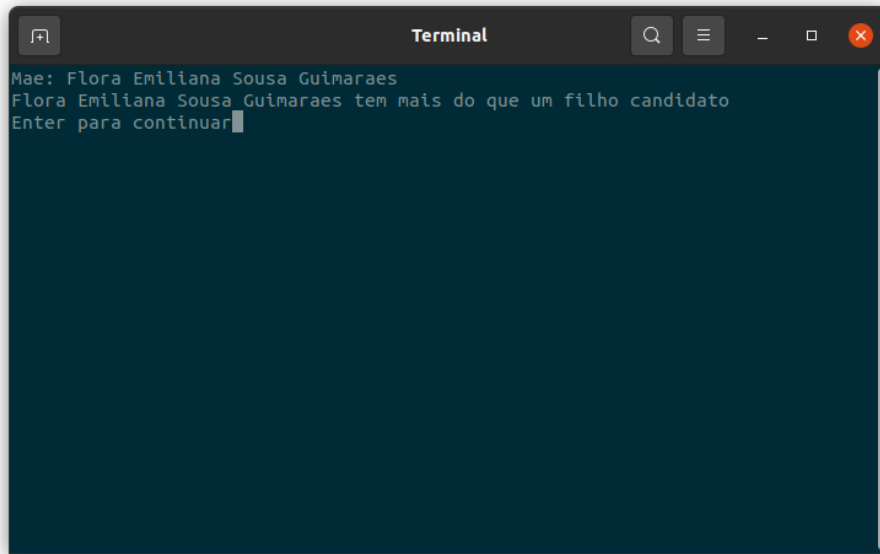


Figura 10: Mãe com mais do que um filho candidato

3.1.5 Alínea e

De seguida apresentam-se as árvores genealógicas dos candidatos referentes ao ano de 1836, assim como o ficheiro `.dot` gerado:

```

1 digraph arvore_genealogica {
2     JacintaMaria -> BernardoCunhaBrochado;
3     AntonioCunhaBrochado -> BernardoCunhaBrochado;
4     LuisaTeixeira -> AlvaroAfonsoTavares;
5     ManuelAfonsoTavares -> AlvaroAfonsoTavares;
6 }
  
```

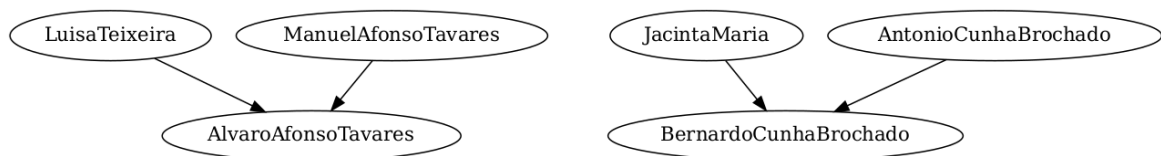


Figura 11: Árvores genealógicas dos candidatos referentes ao ano de 1836

4 Conclusão

Através do desenvolvimento deste projeto conseguimos aumentar a capacidade relativamente à escrita de expressões regulares como motor para a filtração e transformação de textos.

Fazendo uma análise geral ao trabalho desenvolvido, fazemos um balanço bastante positivo do trabalho realizado, conseguindo atingir todos os requisitos propostos. A utilização de Expressões Regulares para o tratamento de *strings* mostrou-se um mecanismo extremamente poderoso, permitindo simplificar tarefas que de outra forma seriam consideravelmente mais complexas e demoradas.

Com o resultado final do projeto e na perspetiva do grupo, pensa-se que se atingiu da melhor forma todos os objetivos propostos e sempre pensando na simplicidade e descomplicação do problema. De facto, foi possível responder a todas as questões, bem como acrescentar algumas funcionalidades extra que consideramos serem relevantes.