Universidade do Minho

Licenciatura em Engenharia Informática

Processamento de Linguagens

Grupo 20

TP1: Conversão de ficheiros CSV para JSON

Joana Maia Teixeira Alves (A93290)

Maria Eugénia Bessa Cunha (A93264)

Vicente Gonçalves Moreira (A93296)

Março 2022

1 Formulação do Problema

O desenvolvimento deste trabalho prático tem como objetivo a construção de um conversor de ficheiros **CSV** (*Comma Separated Values*) para o formato JSON.

O formato CSV tem algumas particularidades como, por exemplo, os cabeçalhos e os caracteres de separação de colunas. Os cabeçalhos definem a designação das várias colunas do documento assim como certas características das mesmas. Já os caracteres de separação de colunas (’**,**’ ou ’**;**’) são caracteres especiais, reservados para a delimitação dos dados pelas várias colunas.

Este trabalho prático, no entanto, engloba mais algumas características deste formato, para além daquelas já referidas, como listas e funções de agregação.

1.1 Listas

As listas são definidas como conjuntos de campos, isto é, várias colunas que se referem ao mesmo campo do cabeçalho. O seu tamanho é definido no cabeçalho, estando delimitado pelos caracteres ’**{**’ e ’**}**’ após a definição do nome da coluna. Esta definição de tamanho pode ter duas variantes: tamanho fixo ou tamanho variável (intervalo de tamanhos).

1.1.1 Listas de Tamanho Fixo

No caso das listas de **tamanho fixo**, existe apenas um único valor dentro dos caracteres de delimitação, obrigando a que todas as linhas tenham esse número de colunas referentes ao campo em questão. Apresentamos então dois exemplos de definição de colunas com listas de tamanho fixo, assim como o conteúdo esperado das linhas do documento:

```
Código,Nome,UC{3}, , ,
93290,Joana,PL,ADI,RC
93296,Vicente,CC,DSS,LI4
```

```
UC,Salas{5}, , , , ,
RC,1.01,1.02,1.03,1.04,1.05
CC,2.14,2.15,2.16,2.17,2.18
```

1.1.2 Listas de Tamanho Variável

As listas de **tamanho variável** ou com intervalos de tamanho, são caracterizadas pela existência de dois valores delimitadores de tamanho, o mínimo e o máximo. Assim, os valores são apresentados da seguinte forma: {min,max}. O valor **min** refere-se ao número mínimo de colunas relativas ao campo, e, naturalmente, o valor **max** refere-se ao valor máximo de colunas do campo. Exemplo:

```
NºGrupo,Integrantes{2,4}, , , ,
00,Vasco,João,Filipe,Patrícia
10,Luísa,Maria,,
20,Diogo,Rafael,Júlia,,
```

1.2 Funções de Agregação

As funções de agregação são funções aplicadas às listas definidas acima. No entanto, algumas destas funções apenas se podem aplicar a números (inteiros ou decimais), sendo por isso necessária a verificação do conteúdo das colunas. Assim, apresentamos as várias funções de agregação disponiveis seguidas das abreviações usadas no programa:

Função	Abreviação
Soma	sum
Média	media (avg)
Produtório	prod
Contador	count
Mínimo	min
Máximo	max

Para aplicar qualquer uma destas funções é necessário referir no cabeçalho do documento, no campo onde se quer aplicar, a abreviação da função de agregação a utilizar, tendo o nome da função de ser antecedido pelos símbolos '**::**'. Para uma melhor percepção apresentamos alguns exemplos:

```
Nº,Notas{3}::media, , , ,
45,11,12,13
77,13,13,14
```

```
Turma,Notas{5}::max, , , , ,
LEI,13,14,15,16,17
MiEI,16,16,17,18,12
```

Nota: O resultado da aplicação das funções de agregação deve ser colocado no ficheiro JSON, não alterando o documento de *input*.

2 Estratégia Adotada

2.1 Divisão em Fases

Com o estudo deste problema, decidimos desenvolver um programa que efetua uma leitura linha a linha do ficheiro CSV original, efetuando o tratamento necessário e adequado a cada linha, obtendo no fim um ficheiro JSON convertido. Para além disso, decidimos que, para cada linha, haverá 3 fases distintas de tratamento, permitindo um desenvolvimento paralelo de cada fase, modularizando assim o projeto e tornando-se, por consequência, mais eficiente. Estas 3 fases são denominadas por: *Tokenizer*, *Line Processor*, e por último, *JSON Converter*.

2.2 *Tokenizer*

Esta fase será responsável pela 'tokenização' da linha lida, identificando assim os vários tokens da linguagem presentes numa linha e efetuando um primeiro pré-processamento. Para esta fase, decidimos utilizar o módulo ***ply*** para efetuar a 'tokenização' da linha, devido à sua fácil utilização e capacidade de tratamento.

2.3 *Line Processor*

Esta segunda fase, sendo esta a que terá mais peso no programa, será responsável pela leitura e processamento dos *tokens* obtidos na fase anterior. Esta fase é responsável por, na primeira linha, definir os nomes dos vários atributos do ficheiro CSV assim como as suas possíveis condições e funções associadas e, nas restantes linhas, atribuir os campos lidos aos seus respetivos atributos e, caso necessário, processar um lista de items, assim como aplicar funções requeridas. Devido à sintaxe semelhante entre um objeto dicionário em *Python* e um objeto JSON, decidimos que, no final desta fase, o *Line Processor* terá que devolver um objeto dicionário da linguagem *Python*, contendo toda a informação processada.

2.4 *JSON Converter*

A última fase tem como objetivo manipular o dicionário fornecido e converter este num objeto JSON, alterando a sintaxe conforme necessário e, por razões de melhor visualização do ficheiro final, acrescentar parágrafos e indentação. Esta fase será desenvolvida recorrendo ao módulo ***re*** utilizando, em específico, a função "sub".

3 Solução do Problema

3.1 *Tokenizer*

Denominamos de *Tokenizer* o processo de interpretação de linhas e criação de *tokens*. Estes *tokens* são utilizados para identificar valores e etiquetá-los para que sejam corretamente processados.

3.1.1 Estado 'FirstLine'

Visto que a primeira linha do ficheiro CSV irá conter os nomes dos campos, assim como possíveis restrições de tamanho e funções de agregação a aplicar e, tendo estes campos uma gramática diferente das restantes linhas do ficheiro, decidimos criar um novo contexto denominado '**firstline**'. Este será exclusivo e irá conter *tokens* de forma a diferenciar e identificar os campos do cabeçalho.

Assim, na inicialização do nosso *Lexer*, definimos que este começará sempre no estado/contexto 'firstline' e, depois de terminar a leitura da primeira linha, este troca o seu contexto para '**INITIAL**', onde irá tokenizar as linhas normalmente.

Para este primeiro estado, decidimos criar tokens específicos por cada "ação" possível no cabeçalho:

- **Campos/Atributos (KEY) -** Definimos estes como uma *string* que irá corresponder ao nome do campo/atributo, sendo esta *string* mais restrita pois não poderá conter o caracter '{'.
- **Delimitador de listas (MIN/MAX) -** Os valores dos delimitadores são sempre definidos por um número inteiro e poderão corresponder a um delimitador simples, contendo apenas um limite fixo mínimo de campos pertencentes a um atributo, ou um delimitador composto, contendo um valor mínimo e máximo de campos a serem lidos.
- **Funções de Agregação (FUNC) -** Definido como uma *string* simples após a leitura dos caracteres '::'.

Listamos de seguida os vários *tokens* do estado 'firstline' desenvolvidos, assim como as suas correspondentes expressões regulares:

Tokens do estado '**firstline**'		
Token	Descrição	Expressão Regular
MIN	Valor mínimo de um campo-lista	{\d+(,|})
MAX	Valor máximo de um campo-lista	\d+}
FUNC	Função de agregação a ser utilizada no campo-lista	::[a-zA-Z]+
KEY	Nome do campo/atributo	[^,{\n]+
NEWLINE	Parágrafo Simples	\n
'ignore'	Caracteres a serem ignorados	, ;

No pré-processamento destes *tokens*, aplicamos a função "strip" de forma a remover caracteres extra indesejados. Recorremos também às funções simples de conversão de tipos de dados. Por último, o *token* "NEWLINE" será responsável por alterar o contexto interno do *lexer*, modificando este para o estado 'INITIAL'.

3.1.2 Estado 'INITIAL'

Depois da primeira linha ser interpretada, o *Lexer* irá modificar o seu estado de forma a ler as restantes linhas do ficheiro CSV.

Estas restantes linhas são de fácil interpretação, tendo apenas o *lexer* ser capaz de distinguir números, strings e vírgulas. Para os números, decidimos suportar todos os números possíveis, ou seja, decimais positivos e negativos sendo estes pré-processados como *floats*. Para as *strings*, acrescentamos a possibilidade de haver campos que utilizem vírgulas "literais", desde que este campo seja delimitado por aspas.

Listamos de seguida os *tokens* do estado 'INITIAL' elaborados, assim como as suas correspondentes expressões regulares:

Tokens do estado '**INITIAL**'		
Token	Descrição	Expressão Regular
NUM	Número Inteiro ou Decimal, positivo ou negativo	-?\d+(.\d+)?
STRING	String de campo	("[^"]+"|[^,;]+)
VIRG	Vírgula delimitadora de campos	,|;
'ignore'	Caracteres a serem ignorados	\t \n

3.2 *Line Processor*

Esta fase, como o nome indica, será responsável pelo processamento das linhas do ficheiro CSV, previamente etiquetadas pelo *Tokenizer*.

No entanto, como o processamento do ficheiro CSV é efetuado linha a linha, uma dificuldade que surge é o registo do nome dos campos e as suas restrições que é efetuado na primeira linha. Para resolver este problema, decidimos que, na primeira linha lida, a função de processamento será responsável por ler os vários *tokens* e gerar uma lista de "chaves". Esta "chave", será um dicionário e irá conter toda a informação de um dado campo/atributo.

De seguida, as restantes linhas serão processadas utilizando como referência a lista de chaves gerada na primeira linha. As chaves são interpretadas por ordem e ditam o comportamento de leitura dos vários *tokens* (por exemplo, se existir um delimitador mínimo n, terá que ler n *tokens*).

3.2.1 *First Line Processor*

Como dito anteriormente, decidimos que o tratamento desta primeira linha deverá ser um processo exclusivo devido à importância declarativa desta linha pois esta enuncia o tipo e a quantidade de dados a serem analisados nas restantes linhas. Este processamento é efetuado através da função "readFirstLine" que tem como objetivo gerar a lista de chaves.

Esta lê os vários *tokens* gerados pelo *Tokenizer* e, por cada *token* "KEY" lido, cria um novo dicionário com o atributo "KEY" lido e adiciona-o à lista de chaves. De seguida, caso leia restrições extra como delimitadores ou uma função, este irá acrescenta-las à entrada de dicionário mais recente na lista, associando a chave anterior às restrições lidas.

Esta função também tem em conta a ordem dos tipos de *tokens* lidos, tendo esta que ser respeitada (*Ordem completa: KEY -> MIN -> MAX -> FUNC). Também é verificado o valor da função lido, sendo este avaliado com uma lista de funções existentes. Caso algum destes processos falhe, é registado um erro que será tratado na função *main* principal.

Exemplo de uma lista de "chaves": (Retirado do ficheiro *listasInterTam.csv*)
[{'KEY': 'Número'}, {'KEY' : 'Nome'}, {'KEY': 'Curso'},
{'KEY': 'Notas', 'MIN': 3, 'MAX':5}]

*Ordem completa: Todas as restrições/ações possíveis são utilizadas.

3.2.2 Processamento de Campos (Restantes Linhas)

Depois da primeira linha do ficheiro CSV ser lida e armazenada numa lista de chaves, as restantes poderão ser processadas. As linhas de conteúdo são processadas em ordem às chaves, ou seja, para cada chave presente na lista, é lido o número de *tokens* necessários para satisfazer a chave.

Para isso, começamos por avaliar se a chave é simples (só contém nome do atributo), ou se contém restrições de tamanho e funções de agregação. Caso o atributo seja simples, apenas é lido o campo correspondente (*String* ou *Num*) e este é colocado no dicionário da linha atual (no caso de ler uma vírgula, ou seja, um campo vazio, é detetado Erro). Caso o atributo contenha um delimitador (MIN/MAX), são lidos "MAX" *tokens* (caso não haja MAX definido, então MAX = MIN), que serão adicionados a um lista, sendo verificado que a quantidade mínima de elementos é lida. No fim da leitura dos elementos da lista, caso este atributo contenha um função de agregação, esta é aplicada à lista e o resultado gravado no dicionário.

Quando este ciclo terminar, ou seja, quando as chaves forem todas respeitadas e conterem os devidos valores associados, o dicionário da linha lida fica pronto para a sua conversão num objeto JSON.

3.3 *JSON Converter*

Esta última fase irá converter a linha final de dicionário *Python* num objeto JSON. Para isso, começamos por converter o dicionário num linha manipulável, obtendo o seguinte resultado: (Exemplo retirado do ficheiro "agregacaoSUM.csv")

Figura 1: Dicionário to String

```
{'Número': '3162', 'Nome': 'Cândido Faísca', 'Curso': 'Teatro, Música', 'Notas_sum': 39}
```

A única mudança necessária, de forma a obter um objeto JSON válido, é a modificação do caracter delimitador de campos, sendo que o dicionário *Python* utiliza apóstrofes e o JSON aspas. Esta conversão é realizada a partir da função "sub" presente no módulo ***re***, obtendo o seguinte resultado:

Figura 2: Comando SUB utilizado

```
# TROCA PLICAS POR ASPAS ('Chave' -> "Chave")
# {"Número": "264", "Nome": "Marcelo Sousa", "Curso": "Ciência Política", "Notas_sum": 76}
dic = re.sub(r'(\'|\")([^\'\"]+)\1',r'"\2"',dic)
```

Figura 3: Resultado Troca de Apóstrofes

```
{"Número": "3162", "Nome": "Cândido Faísca", "Curso": "Teatro, Música", "Notas_sum": 39}
```

Por último, para melhorar o aspeto visual do ficheiro resultante, aplicamos vários caracteres de parágrafos e identação entre cada atributo, esta sub-fase requer a utilização da função "sub" 3 vezes distintas, sendo este o resultado final desejado:

Figura 4: Objeto JSON final

```
{
        "Número": "3162",
        "Nome": "Cândido Faísca",
        "Curso": "Teatro, Música",
        "Notas_sum": 39
}
```

3.4 Main

Este ficheiro contém o funcionamento principal do programa. Este efetua verificações dos ficheiros de *input* e *output*, faz as escritas iniciais e finais no ficheiro JSON e contém o *loop* principal do programa, recorrendo sequencialmente às funções disponibilizadas por cada fase, assim como trata qualquer erro que estas possam encontrar.

3.5 Decisões e Deteção de Erros

Nesta secção apresentamos uma listagem de deteção de possíveis erros que o programa disponibiliza em cada fase, assim como algumas justificações que nos levaram a tomar em conta estes erros:

- **TOKENIZER: Caracteres Ilegais -** Esta deteção de erros provém nativamente do funcionamento do módulo ***ply***, sendo declarado um erro quando é encontrado algum caracter que não foi definido na gramática.
- **LINE PROCESSOR: Ordem de *Tokens* no Cabeçalho -** Este controlo de erros assegura que o cabeçalho do ficheiro CSV contém um cabeçalho válido. Este controlo é feito de várias formas como: deteção de restrições de atributos, sem ter lido o nome do atributo; leitura de uma função de agregação, num atributo não lista e leitura de funções inválidas, etc. (**Nota adicional:** Neste cabeçalho, é ignorado quaisquer vírgulas, de forma a permitir a escrita flexível do cabeçalho).
- **LINE PROCESSOR: Composição de Listas e Funções de Agregação -** O controlo deste erro é um dos mais complexos, pois envolve muitos casos específicos. Um dos primeiros erros detetados é a leitura de campos vazios quando o número mínimo de elementos da lista não foi alcançado. Outro erro envolve as funções de agregação que requerem listas numéricas, como por exemplo, a média. Neste caso, para cada elemento é verificado se este é um valor numérico válido. (**Nota:** Função "count" permite listas compostas de números/*strings*, visto que apenas conta o número de elementos da lista).
- **LINE PROCESSOR: Campos Vazios -** Para o nosso programa, decidimos que todos os campos no ficheiro CSV teriam de ser não nulos (Com a exceção de elementos de uma lista, que não serão adicionado a esta), logo, incluímos também esta deteção de erro.
- **LINE PROCESSOR: Campos Extras -** Como utilizamos uma estratégia de leitura orientada às chaves, onde cada chave lê os seus respetivos *tokens*, é possível que as linhas contenham mais campos para além daqueles que o cabeçalho indica, sendo estes ignorados. Decidimos que, nestes caso, não seriam reportados erros e estes valores adicionais não são convertidos no ficheiro final.

4 Exemplos de Implementação

Nesta secção apresentamos *prints* dos ficheiros de *input* e os respetivos ficheiros de *output* do programa desenvolvido, focando-se estes em problemáticas diferentes.

4.1 Campos Simples

```
Número,Nome,Curso
3162,Cândido Faísca,Teatro
7777,Cristiano Ronaldo,Desporto
264,Marcelo Sousa,Ciência Política
```

Figura 5: Ficheiro Input.


```
[
    {
        "Número": "3162",
        "Nome": "Cândido Faísca",
        "Curso": "Teatro"
    },
    {
        "Número": "7777",
        "Nome": "Cristiano Ronaldo",
        "Curso": "Desporto"
    },
    {
        "Número": "264",
        "Nome": "Marcelo Sousa",
        "Curso": "Ciência Política"
    }
]
```


Figura 6: Ficheiro Output.

4.2 Campos com Intervalo de Tamanho

```
Número,Nome,Curso,Notas{3,5},,,,,
3162,Cândido Faísca,Teatro,12,13,14,,
7777,Cristiano Ronaldo,Desporto,17,12,20,11,12
264,Marcelo Sousa,Ciência Política,18,19,19,20,
```

Figura 7: Ficheiro Input.

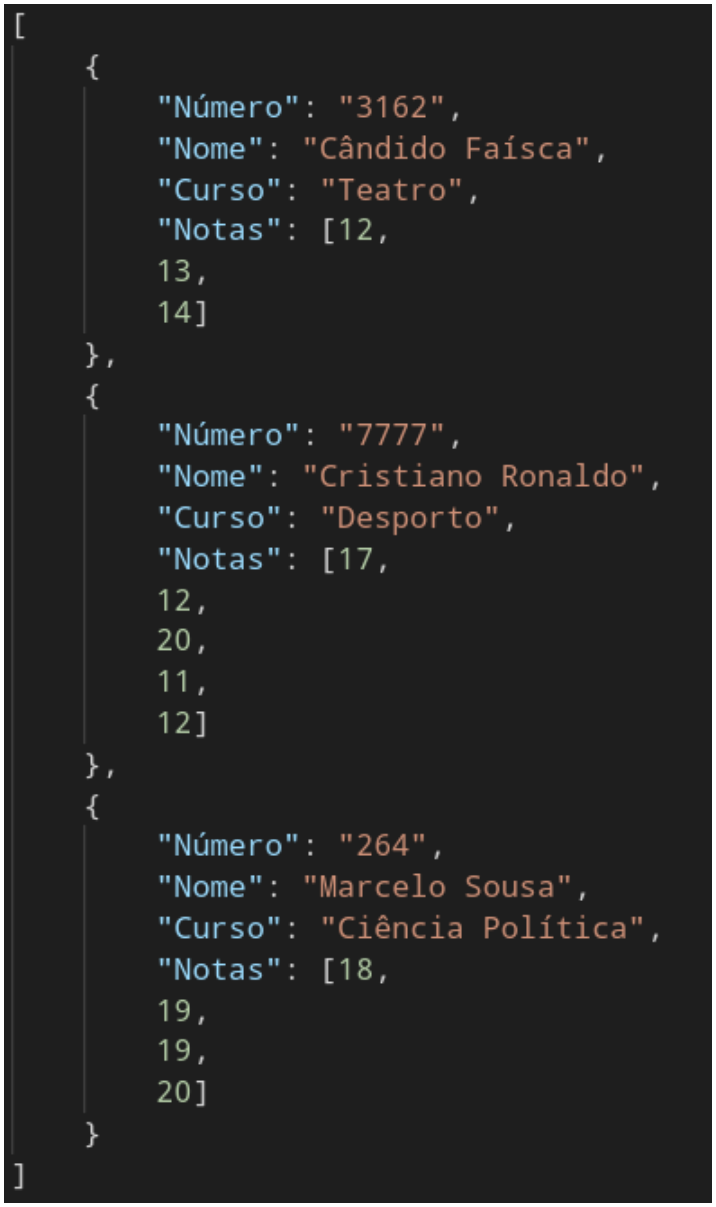

Figura 8: Ficheiro Output.

4.3 Campos com Listas e Funções de Agregação

```
Número,Nome,Curso,Notas{3,5}::media,,,,,
3162,Cândido Faísca,Teatro,12,13,14,,
7777,Cristiano Ronaldo,Desporto,17,12,20,11,12
264,Marcelo Sousa,Ciência Política,18,19,19,20,
```

Figura 9: Ficheiro Input.


```
[
    {
        "Número": "3162",
        "Nome": "Cândido Faísca",
        "Curso": "Teatro",
        "Notas_media": 13
    },
    {
        "Número": "7777",
        "Nome": "Cristiano Ronaldo",
        "Curso": "Desporto",
        "Notas_media": 14.4
    },
    {
        "Número": "264",
        "Nome": "Marcelo Sousa",
        "Curso": "Ciência Política",
        "Notas_media": 19
    }
]
```


Figura 10: Ficheiro Output.

4.4 Múltiplos Campos com Listas

```
Nº,Nome,UCs{5},Notas{5},,,,,Passou
1000,Vicente Moreira,PL,ADI,RC,BD,IO,17,18,14,12,16,True
2043,Joana Alves,PL,ADI,RC,BD,IO,20,20,20,20,20,False
1232,Maria Cunha,PL,ADI,RC,BD,IO,17,18,14,12,16,True
```

Figura 11: Ficheiro Input.

```
[
    {
        "Nº": "1000",
        "Nome": "Vicente Moreira",
        "UCs": ["PL",
        "ADI",
        "RC",
        "BD",
        "IO"],
        "Notas": [17,
        18,
        14,
        12,
        16],
        "Passou": "True"
    },
    {
        "Nº": "2043",
        "Nome": "Joana Alves",
        "UCs": ["PL",
        "ADI",
        "RC",
        "BD",
        "IO"],
        "Notas": [20,
        20,
        20,
        20,
        20],
        "Passou": "False"
    },
    {
        "Nº": "1232",
        "Nome": "Maria Cunha",
        "UCs": ["PL",
        "ADI",
        "RC",
        "BD",
        "IO"],
        "Notas": [17,
        18,
        14,
        12,
        16],
        "Passou": "True"
    }
]
```


Figura 12: Ficheiro Output.

4.5 Múltiplos Campos com Listas e Funções de Agregação

```
Nº,Nome,UC's{5}::Count,Notas{2,5}::MAX,Passou{1,1}
1000,Vicente Moreira,PL,ADI,RC,BD,IO,17,18,14,12,16,True
"2043",Joana Alves,PL,ADI,RC,BD,IO,20,20,20,20,20,False
1232,Maria Cunha,PL,ADI,RC,BD,IO,17,18,14,12,16,True
```

Figura 13: Ficheiro Input.


```
[
    {
        "Nº": "1000",
        "Nome": "Vicente Moreira",
        "UC's_count": 5,
        "Notas_max": 18,
        "Passou": ["True"]
    },
    {
        "Nº": "2043",
        "Nome": "Joana Alves",
        "UC's_count": 5,
        "Notas_max": 20,
        "Passou": ["False"]
    },
    {
        "Nº": "1232",
        "Nome": "Maria Cunha",
        "UC's_count": 5,
        "Notas_max": 18,
        "Passou": ["True"]
    }
]
```


Figura 14: Ficheiro Output.

5 Conclusão e Apreciação Crítica

Durante a realização deste projeto pudemos perceber, consolidar e trabalhar os conteúdos e conceitos lecionados nas aulas teóricas relativos a expressões regulares assim como o funcionamento do módulo *ply*, expandindo, desta forma, os nossos conhecimentos relativos à utilidade prática destes.

O nosso grupo encontra-se satisfeito com o trabalho realizado, tendo alcançado todas as metas propostas pelos docentes e acrescentando algumas funcionalidades de deteção de erros e apresentação. No entanto, estamos cientes que existem possíveis melhoramentos no programa desenvolvido, quer na utilização mais cuidada da ferramenta do módulo *ply*, quer na conversão para o ficheiro JSON (a identação de listas fica incorretamente feita), ou mesmo na estratégia geral adotada, sendo que uma estratégia de leitura linha a linha não será tão eficiente em termos computacionais em relação a uma leitura completa.

Futuramente, com o conhecimento obtido deste trabalho prático e com um estudo mais detalhado desta área, acreditamos que poderão ser desenvolvidas ferramentas mais complexas e robustas que ofereçam um bom número de funcionalidades.