Departamento de Engenharia Informática

Extracção eficiente de padrões textuais utilizando algoritmos e estruturas de dados avançadas

Alexandre Nuno Capinha Gil

Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para obtenção do grau de Mestre em Engenharia Informática

Orientador

Doutor Gabriel Pereira Lopes

Lisboa, 2002

Ao meu Príncipe e Princesas

À memória do meu Pai

Moça
deixa-me caminhar a teu lado
prometo guardar distâncias
pôr de fora as circunstâncias
e ir calado

Moça
deixa-me caminhar a teu lado
prometo ser indiferente
pôr os olhos bem à frente
e ir calado

Moça
é só para caminhar a teu lado
sentir de perto a tua leveza
ficar cheio da tua riqueza
e ir calado

Moça
ainda que estranhes tudo isto
e a maneira como insisto
deixa-me caminhar a teu lado

A tua companhia e o teu jeito
valem bem as promessas que tenho feito

Amilcar Gil

I RESUMO

Esta dissertação tem como ponto de partida o trabalho iniciado por Gaël Harry Dias[1], Sylvie Guilloré[2], e Gabriel Pereira Lopes[3] (Dias *et al.* 1999a), em torno da procura de métodos, puramente estatísticos, que permitam a identificação directa de *expressões relevantes* (e.g "Força Aérea", "União Europeia", "Taça ___ ___ em futebol") que ocorrem mais frequentemente que o simples acaso faria prever, em *corpus* de dimensão elevada constituídos por textos escritos numa língua natural. É principal objectivo do método proposto a extracção automática de *expressões relevantes*, formadas por sequências de *unidades lexicográficas* (e.g. caracteres, palavras, sinais de pontuação, etiquetas), contíguas ou não contíguas, que sejam assumidas como unidades sintáctico-semânticas, com significado próprio. O método proposto é composto por um algoritmo, o *GenLocalMaxs*, que permite a identificação de *expressões relevantes*, e por uma medida, a *Expectativa Mútua*, para a quantificação da força de associação entre os componentes de cada sequência de *unidades lexicográficas* ou *n-grama*.

Esta dissertação torna mais robusto, numa vertente puramente computacional, o método de extracção proposto, apresentando uma solução computacional que permite a implementação muito eficiente, em espaço e tempo, do *GenLocalMaxs* e da *Expectativa Mútua*. A solução maximiza a utilização de memória central e minimiza os tempos de processamento para o volume de dados envolvidos (milhões de *unidades lexicográficas*). Estes eram, no fundo, os objectivos requeridos.

É proposta uma nova representação para os *n-gramas*, baseada numa *máscara*, permitindo níveis de abstracção até então não explorados e abrindo novas perspectivas para o processamento, em tempos aceitáveis, de grandes colecções de textos. É apresentado e explorado um conjunto de propriedades, associadas ao *GenLocalMaxs* e à *Expectativa Mútua*, fundamentais para a simplificação da solução apresentada. É um trabalho fundamentalmente experimental suportado por progressivos desenvolvimentos em torno dum protótipo, desenvolvido em C++, sobre o qual foram testadas, avaliadas e validadas várias propostas alternativas.

---

[1] Universidade da Beira Interior, Departamento de Informática,

[2] Laboratoire d'Informatique Fondamentale d'Orléans, França

[3] Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia

II ABSTRACT

The basis for this thesis is the work initiated by Gaël Harry Dias[4], Sylvie Guillore[5] and Gabriel Pereira Lopes[6] (Dias et al. 1999a), in search of strictly statistical methods for direct identification of *multiword lexical units* (e.g., "Air Force", "European Community", "Football ___ ___ cup"). *Multiword lexical units* are expressions that occur more often than chance would enable one to predict, in corpora of huge dimension. The major objective of the method used in this dissertation is the automatic extraction of *multiword lexical units* from sequences of *lexical units* (e.g. characters, words, punctuation marks, labels, etc.), contiguous or not, representing syntactical-semantic units with particular meaning. The proposed method includes an algorithm, the *GenLocalMaxs*, for the identification of *multiword lexical units*, and an association measure, the *Mutual Expectation*, to quantify the binding strength between the components of each sequence of *lexical units* or *n-gram*.

In a pure computational way, this thesis extends the proposed method for extraction, presenting a computer-based solution for a very efficient implementation, in space and time, of the *GenLocalMaxs* and *Mutual Expectation*. Moreover, the proposed solution maximises the use of central memory and minimize the processing times for the volume of data analysed (millions of *lexical units*). These were the requirements that were set before work has started.

A new representation for the *n-grams* based on a *mask* is proposed. This led us to levels of abstraction that had never before been investigated, and opens new perspectives to massive processing of huge text collections in admissible time. This thesis also presents and takes advantage of a set of properties related to *GenLocalMax* and *Mutual Expectation*, giving raise to major simplifications for the presented solution. This is basically an experimental work, developed in progressive stages and supported by an evaluative prototype, implemented in C++, which was used for testing, assessing and validating alternative proposals.

---

[4] Universidade da Beira Interior, Departamento de Informática, Portugal

[5] Laboratoire d'Informatique Fondamentale d'Orléans, France

[6] Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia, Portugal

III AGRADECIMENTOS

Os primeiros agradecimentos vão naturalmente para os meus pais por me terem posto no mundo e por me terem dado a educação e formação que tenho. Jamais esqueci e esquecerei aqueles momentos, normalmente à noite, em que eu e minha mãe nos sentávamos a discutir os exercícios de álgebra ou de geometria. A paixão pelo abstracto, pelo simbólico, pelo método e pelo encantamento de descobrir uma solução escondida mas no entanto, simples, óbvia e bela, foi aqui que certamente despontou e condicionou a minha posterior paixão pela programação.

Ao Doutor Gabriel Pereira Lopes por me ter apresentado a proposta de tese e pela posterior orientação. A abordagem prática e pragmática que imprimiu ao trabalho e à sua orientação foram decisivos para a minha escolha e para o bom andamento da dissertação. Os meus sinceros agradecimentos.

Ao Mestre Gaël Harry Dias pela forma profissional, sempre pronta e muito agradável como me co-orientou e ajudou ao longo desta dissertação. Boa parte das ideias e soluções nasceram das nossas conversas e da troca de *e-mails* ao longo deste ano. O meu muito obrigado.

A todos os professores que no ISEL e na Nova, com a sua forma profissional e apaixonada de ensinar, me contaminaram, com o vírus das ciências da computação. Não querendo ser injusto para todos os outros, não posso deixar de realçar, pela especial importância que tiveram nas minhas opções, os ensinamentos recebidos dos professores Pimenta Rodrigues, Manuel Barata, Pedro Guerreiro, Margarida Mamede, Cardoso e Cunha e Steiger Garção. Bem-haja a todos eles.

Seria injusto não lembrar, igualmente, os muitos professores que desde a escola primária (que grande cabeça a D. Natália) e depois no liceu, souberam alimentar a minha curiosidade pelas coisas da ciência e incentivaram-me a prosseguir.

Aos muitos colegas da Lisnave e da IBM que sempre me acarinharam e ajudaram ao longo da minha carreira profissional. Com certeza contribuíram, de alguma forma, para este trabalho. Um grande abraço de agradecimento a todos eles.

Aos meus amigos de sempre Ricardo Gonçalves e João Pedro. Sem a insistência e o apoio deles não teria embarcado nesta aventura. Grandes companheiros ao longo de toda a vida académica, no ISEL e depois na Nova, e amigos para sempre. Um abraço caloroso de agradecimento aos dois.

Ao meus irmãos, Zanjo, Zé, João, Miguel, Teresa, Zázá, Patrícia e Chico pela simples razão de serem meus irmãos. Aos meus avós pela grata recordação que tenho deles. Aos meus tios, primas, sobrinhos e restante família que de tão longa não poderia mencionar. Obrigado a todos.

Para o fim os agradecimentos ao meu pequeno clã. À Maria João, Rita, Manuel e Georgina, o meu amor e o meu obrigado, pela paciência e compreensão que tiveram durante estes dois anos, por me terem facilitado a vida, pelos passeios não dados, pelas longas horas de ausência e por aturarem o meu mau humor em muitos momentos. Sem vocês não teria conseguido chegar aqui. Este trabalho também é vosso.

IV SIMBOLOGIA E NOTAÇÕES

documento — Sinónimo de texto no contexto deste trabalho. Sequência longa de símbolos duma língua agrupados de acordo com as regras lexicográficas, sintácticas e semânticas da mesma. Corresponde, em termos práticos, a um ficheiro de texto;

corpus — (1) Conjunto de documentos em análise;

(2) Sequência de *tokens* correspondente à *representação intermédia* do *corpus*;

nd — Número de documentos contidos no *corpus*;

caracter — Símbolo elementar constituinte dum texto escrito numa língua natural. Cada carácter tem uma representação gráfica e uma correspondente representação numérica na forma dum número inteiro natural dentro dum determinado intervalo;

unidade lexicográfica — Sequência limitada e contígua, com significado na língua em que o *corpus* está baseado, de um ou mais símbolos, não *delimitadores*. Tem, normalmente, um *delimitador* à esquerda e outro à direita.

Nas línguas naturais correspondem às palavras, aos números e aos caracteres de pontuação. A granulosidade ou número de caracteres por *unidade lexicográfica* e as regras para a sua identificação dependem da língua e da aplicação;

delimitador — Símbolo ou sequência de símbolos (e.g espaço, fim-de-linha, fim-de-texto) que delimitam as *unidades lexicográficas* dum texto. Os *delimitadores* poderão, em algumas aplicações, ser igualmente considerados como *unidades lexicográficas* independentes;

token — Número inteiro natural, maior ou igual a 1, que identifica univocamente uma determinada *unidade lexicográfica*[7] num *dicionário* ☒;

[7] Para simplificação da escrita e sempre que tal não seja confuso, o termo *token* será utilizado como sinónimo de *unidade lexicográfica*.

$\boxtimes$	*Dicionário* ou conjunto de todas as diferentes *unidades lexicográficas*, associadas aos respectivos *tokens*, existentes num *corpus*;
$\|\boxtimes\|$	Dimensão do *dicionário*, ou seja, o número total de *unidades lexicográficas* ou *tokens* diferentes existentes no *corpus*;
N	Dimensão dum *corpus* em número total de *unidades lexicográficas* ou *tokens*;
string	Cadeia ou sequência contígua de caracteres (de *unidades lexicográficas* ou *tokens*);
substring	Parte, contígua, duma *string*;
n-grama	Sequência válida, contígua ou não contígua, de *unidades lexicográficas* (*tokens*) retirada dum *corpus*. Na versão contígua é equivalente a uma *string*;
gap	Posição vazia, não ocupada por uma *unidade lexicográfica* (*token*), num *n-grama* não contíguo;
m	Dimensão da janela de análise, correspondente à dimensão máxima dos *n-gramas* analisados;
d_{max}	Distância máxima ou número máximo de *gaps* consecutivos entre duas *unidades lexicográficas* (*tokens*) num *n*-grama;
M_T	Número total de combinações de *unidades lexicográficas* (*tokens*) geradas a partir duma sequência contígua de *unidades lexicográficas* (*tokens*);
M	Número total de *n-gramas* válidos gerados, de acordo com as regras de validade aplicáveis, a partir duma sequência contígua de *unidades lexicográficas* (*tokens*);
$\blacktriangleright$	Conjunto de todos os *n-gramas* gerados a partir dum *corpus*;
$\|\blacktriangleright\|$	Número total de *n-gramas* associados a um corpus;
$\blacktriangleright_d$	Conjunto de todos os *n-gramas* diferentes gerados a partir dum *corpus*;
$\|\blacktriangleright_d\|$	Número total de *n-gramas* diferentes associados a um corpus;
$\oslash_{n-1}(w)$	Conjunto de *sub-gramas* válidos do *n-grama w*. Cada *sub-grama* é obtido pela substituição duma e só duma das *unidades lexicográficas* (*tokens*) de *w* por um *gap*;
$\oslash_{n+1}(w)$	Conjunto de *super-gramas* válidos do *n-grama w*. Cada *super-grama* é obtido pela substituição dum e só dum *gap* de *w* por uma *unidade lexicográfica* (*token*) ou pela adição duma e só duma *unidade lexicográfica* (*token*) à direita

	ou à esquerda de *w*, com inclusão, se necessário, de um ou mais *gaps* intermédios;
MWU	Do inglês *MultiWord Lexical Unit*. Expressão relevante associada a um padrão textual correspondente a uma sequência, contígua ou não contígua, de *unidades lexicográficas* (*tokens*) na língua em que o *corpus* está expresso;
W	Número total de *MWUs* extraídas do *corpus*.
Web	*Abreviatura para World Wide Web*
IR	Abreviatura para *Information Retrieval*. Pesquisa de Informação em Documentos.
ME	Abreviatura para *Expectativa Mútua*. A sigla advém da expressão inglesa *Mutual Expectation*;
GenLocalMaxs	Abreviatura para o algoritmo que permite a identificação de máximos locais duma função;

V ÍNDICE DE MATÉRIAS

VI ÍNDICE DE FIGURAS

1 INTRODUÇÃO

Os primeiros anos da década de 90, com o nascimento da *World Wide Web* (*Web*), trouxeram alterações profundas à percepção que havia, até então, relativamente aos sistemas de *Pesquisa de Informação em Documentos* (*IR*[8]). Uma área centrada na automatização de índices para bibliotecas ou em investigação algorítmica, puramente académica, passou a abarcar áreas como a modelação, classificação e filtragem de documentos, interfaces com o utilizador, arquitectura de sistemas de armazenamento e pesquisa de informação, entre outras (Baeza-Yates *et al*. 1999). A *Web* é cada vez mais uma grande enciclopédia (Fujii *et al*. 2000) de fácil acesso e ao serviço de todos, que necessita de ferramentas, cada vez mais poderosas e eficientes, para que o conhecimento acumulado possa ser devidamente pesquisado e utilizado.

Urge então procurar novas metodologias para a classificação dos documentos, que permitam passarmos duma visão centrada em dados (e.g. palavras chave singulares, documentos) para uma visão centrada na informação, ou seja, em conceitos e contextos (Baeza-Yates *et al*. 1999). Esta nova metodologia deverá estar baseada em ferramentas de extracção automática de informação de documentos, das quais, os sistemas de extracção de *expressões relevantes*, designadas por *MWUs*[9] em (Dias *et al*.. 1999a) e nos outros artigos dos mesmos autores referenciados na bibliografia, são uma parte. Estes sistemas têm como objectivo a identificação automática de *MWUs*, em *corpus* constituídos por centenas ou milhares de documentos, correspondentes a dezenas ou centenas de milhões de *unidades lexicográficas*.

É neste contexto que Dias *et al*. (1999a) propuseram uma nova abordagem ao problema da extracção de *expressões relevantes*, defendendo que um processo, totalmente independente da língua associada ao *corpus* e sem recorrer a quaisquer limiares[10] pré-definidos, conduz a melhores resultados que os obtidos por métodos anteriores. Para tal, propuseram uma medida de associação, a *Expectativa Mútua*

---

[8] Do Inglês, *Information Retrieval*

[9] *MultiWord Lexical Unit.* Na literatura em língua inglesa é, igualmente, utilizado o termo *collocation* para designar conjuntos, contíguos e não contíguos, de palavras que ocorrem frequentemente juntas e que têm um significado próprio quando tal.

[10] *thresholds* no original

(*ME*[11]), e um algoritmo para extracção de *MWUs*, o *GenLocalMaxs*. O método proposto tem como vantagem adicional o facto de ser aplicável a padrões textuais, daqui para a frente designados por *n-gramas*, contíguos ou não contíguos.

Baeza-Yates *et al.* (1999) lembram, no capítulo dedicado aos métodos de avaliação de sistemas, que as medidas mais comuns, do desempenho dum sistema[12], são o *tempo* e o *espaço*. São igualmente estas duas medidas que se procura minimizar num sistema de extracção de *MWUs*, quando analisado do ponto de vista computacional. Neste contexto, é necessário lembrar a forte dependência, normalmente existente, entre estas duas vertentes. É normal, mas não obrigatório, que poupanças ao nível do espaço ocupado pelas estruturas de dados, impliquem menor eficiência temporal dos algoritmos, e vice-versa.

Os mesmos autores referem, ainda, a importância de outras medidas de avaliação que permitem medir a eficácia[13] do método, ou seja, a capacidade do método para dar respostas correctas. Em *IR* é normal utilizarem-se duas medidas de avaliação complementares: *precisão* e *cobertura*[14]. A *precisão* é a razão entre o número de respostas correctas obtidas e o total de respostas recolhidas. A *cobertura* é a razão entre o número de respostas correctas obtidas e o número de respostas correctas existentes na amostra utilizada. No caso particular dos sistemas vocacionados para a extracção de *MWUs*, a *precisão* é a razão entre o número de *MWUs* correctas e o total de *MWUs* extraídas. A *cobertura* é a razão entre o número de *MWUs* correctas extraídas e o número de *MWUs* existentes na amostra. Esta última medida obriga à existência de *corpus* de referência para os quais se conheça o número de *MWUs* existente. Estes *corpus* não existem, ainda, e a sua construção não é evidente dado que não existe um consenso quanto ao conceito de *MWU* correcta (Dias *et al*. 1999a).

O trabalho efectuado por Dias *et al*. (1999a), ponto de partida para a presente dissertação, está centrado na busca de sistemas mais eficazes (maior *precisão* e *cobertura*) para a extracção de *MWUs*. Esta dissertação prolonga o trabalho destes autores, na vertente computacional, procurando uma implementação eficiente (menos *espaço*, menor *tempo*) do método proposto. Assim, tendo um método eficaz, o projecto, subjacente a esta dissertação de mestrado, tem como objectivo a procura de estruturas de dados e algoritmos avançados para a extracção eficiente de *MWUs*, através do *GenLocalMaxs*, com base no cálculo eficiente da *frequência* e da *ME* do conjunto dos *n-gramas* associado ao *corpus*. É premissa do projecto o desenvolvimento dum protótipo em C/C++ que corra em memória, ou seja, que maximize a utilização de estruturas de dados em memória com o objectivo de reduzir, ao máximo, os tempos de processamento.

---

[11] Do Inglês *Mutual Expectation*

[12] *System performance* no original

[13] *Retrieval performance* no original

[14] *Recall* em Inglês

Estando a trabalhar com largos volumes de texto (milhões de *unidades lexicográficas*), procurou-se estruturas de dados e algoritmos que permitissem minimizar o espaço ocupado em memória central e o tempo associado aos processos de extracção. Estas preocupações estão directamente ligadas à eficiência do sistema. Assim, a busca da estrutura de dados e de algoritmos óptimos, teve por base a procura de mínimos de complexidade através da analise, codificação e teste de várias opções de implementação.

Todas as opções tomadas tiveram sempre como objectivo encontrar soluções primeiramente aplicáveis a *corpus* constituídos por textos escritos numa língua natural mas expansíveis, directamente, a qualquer outra sequência de símbolos que possa constituir um *corpus*, donde se pretenda extrair sequências de símbolos mais fortemente ligadas em termos da força de associação entre os mesmos, como sejam, sequências de ADN (Gusfield 1999), (Burkhardt *et al*. 1999). O protótipo desenvolvido para o módulo de extracção de *MWUs* segue este princípio, pelo que é totalmente independente da língua e do formato do *corpus*.

Nesta dissertação é proposta uma nova representação para os *n-gramas*, componente central da solução apresentada e fundamental para o elevado nível de eficiência das estruturas de dados e dos algoritmos propostos para o cálculo da *frequência*, da *ME* e do *GenLocalMaxs*. Esta nova representação baseia-se numa *máscara* e na utilização de referências, optimizando não só a complexidade da estrutura de dados como a dos algoritmos. É especialmente importante para os algoritmos que permitem a obtenção eficiente dos conjuntos de *sub-gramas* e de *super-gramas*, necessários ao cálculo da *ME* e para o *GenLocalMaxs*. De realçar o facto, particularmente interessante, de uma estrutura de dados, o conjunto de *máscaras*, perfeitamente desprezível em termos do espaço necessário para o seu armazenamento e do tempo implicado na sua construção, estar na base da quase totalidade das optimizações propostas quer em termos espaciais quer temporais.

A solução proposta tira partido dum conjunto de propriedades associadas à *frequência*, à *ME* e ao *GenLocalMaxs*, quando utilizado em conjunto com a *ME*, justificativas de parte das opções tomadas e com impacto significativo na eficiência final da solução proposta.

Esta dissertação pretende igualmente colmatar a falta de literatura relacionada com o processamento de *n-gramas* não contíguos, associados a largos volumes de texto, nomeadamente com a apresentação de algoritmos para o cálculo eficiente da *frequência* associada aos mesmos.

Este documento segue uma organização quase cronológica. Na presente introdução começou-se por fazer a apresentação do enunciado do problema proposto para resolução no decorrer desta dissertação de mestrado, dando-se, em seguida, uma panorâmica geral sobre a organização dos restantes capítulos, correspondentes a outras tantas fases do trabalho.

O capítulo 2 é dedicado ao enquadramento do problema proposto na área da *IR*, caracterizando os sistemas de extracção de *MWUs* e apresentando o caminho seguido em termos de análise dos trabalhos existentes nesta área. Descreve, resumidamente, as principais fontes de inspiração, com realce para os trabalhos efectuados em torno dos *suffix-arrays* e dos algoritmos de ordenação de *strings*, ambos utilizados, directa ou indirectamente, na solução proposta.

No capítulo 3 são apresentados formalmente o algoritmo *GenLocalMaxs* e a *Expectativa Mútua*, assim como os conceitos de *frequência, token*, *corpus*, *n-grama*, *MWU*, *sub-grama* e *super-grama*, essenciais à definição dos dois primeiros. São, igualmente, apresentadas as propriedades da *frequência*, da *ME* e do *GenLocal Maxs*, posteriormente exploradas na solução proposta.

O capítulo 4 é dedicado à descrição da solução proposta para o módulo de extracção de *MWUs,* na forma duma estrutura de dados e dum conjunto de algoritmos. São ainda apresentadas as simplificações resultantes da aplicação das propriedades enunciadas no capítulo 3. Finalmente são caracterizadas as várias alternativas de implementação postas em prática e testadas.

No capítulo 5 é apresentada a estrutura do protótipo, desenvolvido em C++, e dos vários módulos que o compõem. Contém igualmente as informações básicas que permitem a utilização dos executáveis produzidos para os diferentes módulos.

Os resultados, em termos de espaço consumido e de tempos, são apresentados no capítulo 6. O sistema foi testado em duas máquinas diferentes e sobre dois sistemas operativos distintos (*Windows 2000* e *Linux*). Para os testes foram utilizados documentos extraídos do *corpus* do Público/MCT[15]. Foi possível obterem-se tempos na casa das dezenas de minutos para *corpus* com alguns milhões de *tokens*, o que é francamente bom.

Um dos factores motivadores da proposta para a realização desta dissertação foi a existência duma implementação pouco eficiente do *GenLocalMaxs* em conjunção com a *ME*. Esta implementação, designada por *SENTA*[16] (Dias *et al*. 2000a) está baseada nas ferramentas, disponíveis em *Linux*, para o processamento de textos. Foi tomada a decisão, para não influenciar as opções tomadas, pela sua não avaliação prévia. Somente, no final desta dissertação foram efectuadas medidas comparativas para avaliação da correcção dos resultados. O *SENTA* apresenta níveis de eficiência temporal muito baixos quando comparado com a implementação proposta nesta dissertação.

Finalmente, no capítulo 7, são tiradas as conclusões e apresentada uma lista de hipóteses de expansão dos resultados obtidos em trabalhos futuros. Particularmente importantes são os caminhos propostos

---

[15] O *corpus* do Público/MCT tem como base notícias variadas publicadas no jornal Público ao longo dum período longo

[16] *Software for the Extraction of N-ary Textual Associations*. Acessível via *Web* através do endereço http://lince.di.ubi.pt/~senta

em termos de paralelização dos algoritmos e de utilização de estruturas de dados não exclusivamente residentes em memória central, estendendo as vantagens do conceito de *máscara* a outros meios, nomeadamente ficheiros e bases de dados. Ambos estes caminhos procuram aumentar os limites do sistema, permitindo o tratamento de *corpus* ainda mais extensos.

2 EXTRACÇÃO DE EXPRESSÕES RELEVANTES

A *IR* é uma área actualmente em expansão dada a quantidade crescente de documentos, em formato electrónico, disponível na *Web* e em outros tipos de redes ou sistemas similares (e.g. bibliotecas electrónicas). Esta imensidão de documentação necessita ser tratada e classificada para que possa ser acedida duma forma fácil e rápida (Baeza-Yates *et al*. 1999).

Os motores de pesquisa de documentos na *Web* (e.g Altavista, Google, Yahoo), são a face mais visível da aplicação da investigação e desenvolvimento efectuados em *IR*. No entanto, podem ser considerados como sistemas ainda muito primitivos se atendermos ao modo como os documentos estão classificados, basicamente recorrendo a palavras-chave, ao formato admitido para os pedidos de informação[17], expressões lógicas simples do tipo "palavra *and* palavra *or* palavra" e à, normalmente, baixa *precisão* e *cobertura* das respostas, obrigando o utilizador a perder muito tempo até obter a resposta desejada (Baeza-Yates *et al*. 1999), (Fujii *et al*. 2000). São sistemas que ainda estão mais virados para os dados (i.e. documentos, palavras) que para a informação lá contida (i.e. conceitos, contextos). Nos próximos anos, é essencial que o foco passe a estar, cada vez mais, na informação e menos nos dados (Baeza-Yates *et al*. 1999). Para tal, são precisas ferramentas cada vez mais poderosas, eficazes e eficientes, para a organização, classificação e pesquisa de informação em documentos.

Tal como afirmam Baeza-Yates e Ribeiro-Neto (1999), para que um sistema de *IR* seja efectivo na busca de informação deve, de algum modo, "interpretar" os documentos armazenados, extraindo-lhes informação sintáctica e semântica, e utilizar esta informação para permitir consultas mais ricas, do ponto de vista da formulação dos requisitos de informação pretendidos, e para melhorar a *precisão* e a *cobertura* das respostas dadas. O modo mais simples de representação é a utilização de listas de palavras, pré-tratadas ou não, como índices representativos, para facilitar as posteriores pesquisas. Os índices podem assumir estruturas variadas, desde simples tabelas de palavras chave até estruturas hierárquicas, mais ou menos complexas. Durante séculos os índices foram criados manualmente. Actualmente é usual ser uma tarefa semi-automática ou totalmente automatizada (Baeza-Yates *et al*.

[17] *Queries* na literatura Inglesa

1999). A criação de índices é normal não abarcar todas as palavras existentes nos documentos mas somente um subconjunto, seleccionado por um especialista na matéria tratada, que se considere relevante para as posteriores necessidades de pesquisa (Baeza-Yates *et al*. 1999). Apesar das capacidades cada vez maiores dos computadores actuais, a necessidade de condensação do conhecimento em listas de termos mais relevantes continua a ser totalmente válida dada a quantidade enorme de textos existentes, por exemplo, em toda a *Web* ou numa biblioteca. É igualmente importante que as chaves para a classificação dos documentos deixem de ser somente baseadas em palavras singulares, mas que possam igualmente ser armazenados termos, na forma de *MWUs*, devidamente contextualizados (Silva *et al*. 2001). Somente deste modo será possível tornar mais ricas as consultas e mais precisas e completas as respostas.

A automatização das tarefas subjacentes à classificação de documentos torna-se essencial para podermos responder, em tempo útil, às quantidades enormes de documentos publicados diariamente (Silva *et al*. 2001). Várias aproximações têm sido seguidas para esta automatização, quase todas baseadas, directamente ou indirectamente, em métodos linguísticos. Por exemplo, é normalmente aceite que são os substantivos que carregam a maior carga semântica dum texto, sendo consequentemente eleitos, automaticamente, como palavras chave para a classificação. A utilização de grupos de palavras, contíguas ou não contíguas, especificando *conceitos*, é uma estratégia que tem sido seguida na criação de índices. Quando uma só palavra não é suficiente para expressar completamente um determinado *conceito*, recorre-se a duas ou mais palavras (e.g. "míssil balístico") (Baeza-Yates *et al*. 1999).

É no contexto desta nova realidade que os sistemas de extracção automatizada de termos, *expressões relevantes* ou *MWUs*, ganham importância. Estes sistemas não são mais que aplicações que permitem a identificação automática de sequências, contíguas ou não contíguas, de *unidades lexicográficas* (e.g. palavras, sinais de pontuação) que constituam uma *MWU*, ou seja, que estejam associadas a um *conceito* perfeitamente identificável. Estes padrões textuais são, por exemplo, substantivos compostos, expressões idiomáticas, verbos compostos, locuções preposicionais, ou locuções adverbiais. Genericamente, são expressões que ocorrem mais vezes do que o simples acaso faria prever (Dias *et al*. 1999a).

A extracção de *MWUs* é importante não só para a classificação e indexação de documentos mas para muitas outras áreas como seja a tradução automática (Dias *et al*. 1999), (Ikehara *et al*. 1996), ou o alinhamento de textos paralelos (Ribeiro *et al*. 2001).

Existem, como é afirmado em (Dias *et al.* 1999), basicamente três abordagens para a extracção de *MWUs* dum documento ou dum conjunto de documentos agrupados num *corpus*:

1. Utilização de técnicas baseadas em métodos linguísticos;

2. Utilização de métodos puramente estatísticos onde o reconhecimento de *MWUs* é um processo totalmente independente da língua base aos documentos;
3. Um misto das duas anteriores onde são estabelecidos limiares a partir dos quais certos padrões textuais são assumidos como relevantes.

Exemplos da primeira e da terceira vias são a utilização de padrões ou modelos linguísticos e ou de etiquetas morfosintáticas para a extracção de termos. São métodos dependentes da língua ou então dependentes de heurísticas baseadas na utilização de determinados padrões ou sequências de tipos de palavras ou de etiquetas. Obrigam à existência de bases de dados, actualizadas, de padrões linguisticos (Fujji *et al*. 2000), (Ikehara *et* al. 1996) e ao desenvolvimento de medidas e dos respectivos limiares de aceitação.

A segunda via tem a vantagem de ser completamente independente de qualquer conhecimento prévio sobre a língua e sobre a estrutura do *corpus* a analisar. Nagao e Mori (1994) propõem um método estatístico para a identificação de *MWUs* baseado no cálculo eficiente do número de ocorrências de cada *sufixo*[18] do texto em análise. Uma versão melhorada deste trabalho, em termos da *precisão* e da *cobertura* das *MWUs* extraídas, é apresentada em (Ikehara *et al*. 1996) e posteriormente em (Yakamoto *et al*. 2000). O algoritmo *GenLocalMaxs* e a medida de associação *ME*, bases para esta dissertação, integram-se na vertente puramente estatística para a extracção de *MWUs* (Dias *et al*. 1999a).

Os métodos estatísticos propostos são todos baseados, directa ou indirectamente, no conhecimento da *frequência*, ou seja, no número de vezes em que cada *n-grama* ocorre no *corpus*. Assim, sabendo que os volumes de informação envolvidos são extremamente elevados (Ikehara *et* al. 1996), da ordem das dezenas ou centenas de milhões de *unidades lexicográficas*, resultando em muitos mais milhões de combinações possíveis de *unidades lexicográficas* para a formação de *n-gramas*, um dos maiores desafios que qualquer sistema de extracção de *MWUs*, baseado puramente em medidas estatísticas, enfrenta é o de encontrar métodos eficientes, em espaço e tempo, para o cálculo da *frequência* associada a cada *n-grama*.

Na literatura relacionada com *IR*, nomeadamente a referenciada no final deste documento, existem várias referências a estruturas de dados e algoritmos para resolução do problema da pesquisa de *strings* em textos de grandes dimensões, baseados, nomeadamente, em diferentes técnicas de indexação de textos (Charras *et al*. 1997), (Frakes *et al*. 1992), (Gusfield 1999). É analisada a complexidade, as vantagens e as desvantagens de vários destes algoritmos. Em menor número, aparecem referências a algoritmos de contagem de ocorrências e os que são descritos aplicam-se a

[18] Um *sufixo* é uma *string* que se inicia em qualquer posição do texto e se prolonga até ao final do mesmo (Baeza-Yates *et al*. 1999).

sequências contíguas de *tokens*, não sendo directamente expansíveis a sequências não contíguas. A falta de literatura relacionada com sequências não contíguas é uma das dificuldades que este trabalho pretende colmatar.

Tanto em (Nagao *et al*. 1994), como em (Ikehara *et al*. 1996) e em (Yakamoto *et al*. 2000), a estrutura de dados base à obtenção da *frequência* é um *suffix-array*. Esta estrutura foi pela primeira vez descrita por Udi Manber e Gene Myers no seu artigo (Manber *et al.* 1991). Os *suffix-arrays* são os irmãos mais novos das *suffix-trees*, apresentando níveis de eficiência idênticos na maioria das operações. São muito mais eficientes do ponto de vista espacial, especialmente em aplicações em que o alfabeto[19] é extenso ou não limitado, ou em que o espaço ocupado em memória pela estrutura de dados é um factor a ter em conta. São aplicáveis quando o texto é fixo e utilizado em inúmeras buscas consecutivas (Gusfield 1999). Um *suffix-array* necessita em média de 3 a 5 vezes menos espaço que a correspondente *suffix-tree* e é completamente independente da dimensão do alfabeto (Manber *et al.* 1991). O principal obstáculo é a sua construção de forma eficiente.

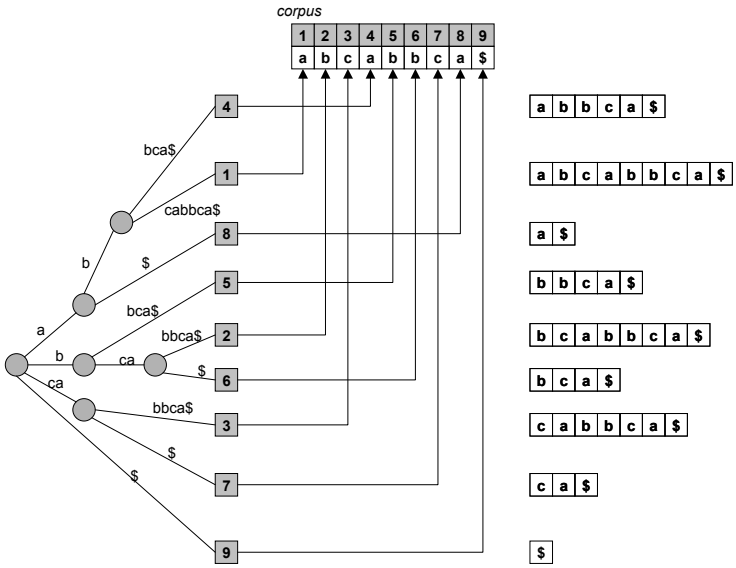


Figura 2.1 – *Suffix-tree* para "abcabbca$"

Tanto as *suffix-trees* como os *suffix-arrays* são formas alternativas de criação de índices sobre textos que permitem a formulação de consultas complexas. Uma *suffix-tree* é uma *trie* construída para o universo de todos, ou de parte, dos *sufixos* dum texto onde os apontadores para os diferentes *sufixos* são guardados nas folhas. Para melhorar o espaço utilizado a *trie* é compactada numa *Patricia tree*

[19] No contexto da extracção de padrões textuais o *alfabeto* é o conjunto de todas as *unidades lexicográficas* diferentes existentes num *corpus*. Também designado por *dicionário* nos restantes capítulos.

(Baeza-Yates *et al*. 1999), (Anderson *et al*. 1996), (Gusfield 1999). Na Figura 2.1 é apresentado um exemplo duma *suffix-tree* para o texto "abcabbca$", onde o caracter "$" é a marca de fim de texto necessária para que todos os *sufixos* do texto sejam diferentes. De realçar que percorrendo as folhas da árvore de cima para baixo, obtém-se a lista ordenada de todos os *sufixos* como é mostrado na parte direita da Figura 2.1.

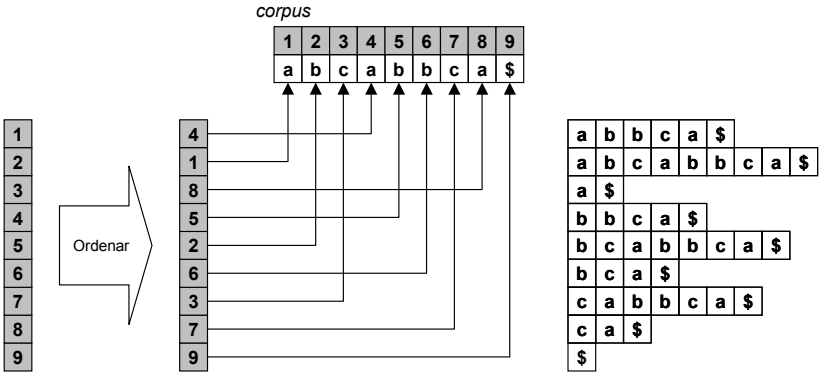


Figura 2.2 – *Suffix-array* equivalente à s*uffix-tree* para "abcabbca$"

Um *suffix-array* é uma implementação mais económica, em termos de espaço, duma *suffix-tree*. Resumidamente, é um vector de apontadores para todas as posições do texto, ordenados lexicograficamente de acordo com os *sufixos* iniciados em cada uma das posições apontadas. A construção do *suffix-array* é extremamente simples de explicar. No entanto, se não forem tomadas as opções correctas, pode ser uma operação muito pouco eficiente, dado envolver comparações entre *strings* de dimensão elevada. O método mais simples de construção necessita somente de dois passos (Manber *et al.* 1991). Num primeiro passo o vector é preenchido sequencialmente com apontadores para todas as posições do *corpus*. O segundo passo consiste em ordenar o vector lexicograficamente de acordo com os *sufixos* apontados por cada elemento do vector. A ordenação pode ser efectuada com base em qualquer algoritmo eficiente de ordenação de *strings*. Vários autores descrevem métodos alternativos, mais eficientes, para a construção dum *suffix-array* (Manber *et al.* 1991), (Gusfield 1999), (Sakadane 1997), (Sakadane *et* al. 1998), (Larsson *et* al. 1999), (McIloy *et al*. 1997). De entre estas referências destaca-se o livro de Gusfield (1999), soberbo em termos da descrição teórica rigorosa das duas estruturas e na apresentação de exemplos da sua aplicabilidade. Em termos algorítmicos é de destacar o trabalho pioneiro de Manber e Myers e o trabalho posterior de McIlroy e McIlroy e de Larsson e Sakadane em termos de síntese e de inovação sobre os algoritmos inicialmente propostos. Parte dos algoritmos propostos em (Larsson *et* al. 1999) foram utilizados nesta dissertação.

As operações de pesquisa e de contagem do número de ocorrências dum *sufixo* num *suffix-array* ficam extremamente mais eficientes se for acrescentado, a cada apontador, um campo contendo a dimensão do maior prefixo comum, abreviadamente *lcp*[20], entre cada dois *sufixos* consecutivos (Gusfiel 1997). Tendo, após o *suffix*-array estar ordenado, conhecimento sobre o número de caracteres iniciais comuns entre cada dois *sufixos* consecutivos, é possível simplificar drasticamente os algoritmos de pesquisa de *strings* (Gusfiel 1997) e os algoritmos de determinação da *frequência* duma qualquer *string* existente dentro do texto em análise (Yakamoto *et al*. 2000), (Nagao *et al*. 1994).

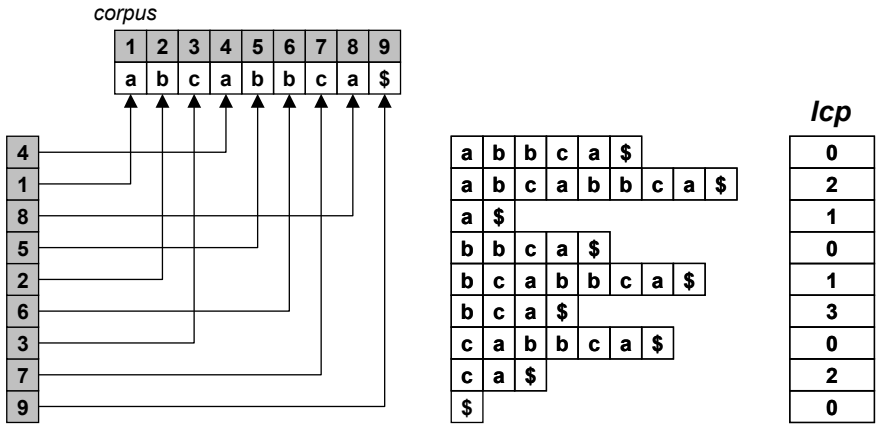


Figura 2.3 – Valores de *lcp* para o *suffix-array* para "abcabbca$"

Tendo os *lcps* calculados, é possível, percorrendo o *suffix-array*, do primeiro para o último índice, obter o número de ocorrências de cada *string* presente no texto, simplesmente utilizando os valores dos vários *lcps*, sem voltar a comparar *sufixos*. No exemplo, mostrado na Figura 2.3, a *string* "a" aparece três vezes porque o *lcp* entre o segundo e o primeiro elemento é maior que 1, o *lcp* entre o terceiro e o segundo é precisamente 1, o caracter 'a', e o *lcp* entre o quarto e terceiro elemento é nulo, ou seja, o 'a' inicial já não é comum entre estas duas posições (Yakamoto *et al*. 2000), (Nagao *et al*. 1994).

Existem múltiplas referências a estruturas de dados derivadas das *suffix-trees* e dos *suffix-arrays*. Irving & Love (2000 e 2001) descrevem duas outras estruturas, designadas por *Suffix Binary Search Tree* e *Suffix AVL Tree*, que foram analisadas como possíveis alternativas de implementação da estrutura de dados necessária à contagem do número de ocorrências de cada *n-grama* mas abandonadas pelo espaço excessivo que necessitam para o seu armazenamento.

[20] *Longest Common Prefix* (Gusfield 1997)

Em termos de sistemas reais, além do *SENTA* (Dias *et al*. 2000a), já referido na introdução, foi superficialmente analisado, já na etapa final da dissertação, o *NSC*[21] (Pedersen 2002), sem ter sido, no entanto, efectuada qualquer comparação directa. O *NSC* é um conjunto de ferramentas, escritas em *PERL*, que permitem a contagem e análise de *n-gramas*, contíguos e não contíguos, em textos, permitindo a aplicação de diferentes tipos de medidas de associação (e.g. *dice*, *log-likelihood*, *mutual information, chi-square test*, *left-fisher test*). É flexível em termos de configuração e de adaptabilidade a diferentes sistemas de regras para a identificação de *unidades lexicográficas* e permite a criação e utilização de outras medidas de associação.

[21] *N-Gram Statistic Package (v0.5)*

3 EXPECTATIVA MÚTUA E GENLOCALMAXS

O ponto de partida para esta dissertação foi o trabalho teórico e experimental iniciado por Dias *et al.* (1999a) em torno da procura duma medida e algoritmo adequados à extracção do conjunto de *MWUs* associado a um *corpus*. Este capítulo apresenta a medida e o algoritmo propostos por estes autores, a *Expectativa Mútua* (*ME*) e o *GenLocalMaxs* respectivamente, começando pelas definições de *n-grama* e de *frequência*, fundamentais à descrição dos primeiros. Finalmente é enunciado um conjunto de propriedades associadas à *frequência*, *ME* e *GenLocalMaxs* fundamentais para a justificação das opções tomadas em termos da solução computacional proposta nesta dissertação.

A *ME* e o *GenLocalMaxs* são apresentados em vários documentos dos autores atrás referenciados, nomeadamente em (Dias *et al.* 1999a), (Dias *et al.* 1999b), (Silva *et al.* 1999), (Dias *et al.* 2000a), (Dias *et al.* 2000b), (Dias *et al.* 2000c) e (Dias *et al.* 2000d). Neste trabalho recorre-se essencialmente à informação contida nos documentos mais recentes por estes serem mais actuais na apresentação da *ME* e do *GenLocalMaxs.*

Combinando a *ME* com o *GenLocalMaxs*, os autores atrás referidos demonstram que é possível construir-se um sistema com um elevado grau de eficácia (*precisão* e *cobertura*) para a extracção de *MWUs*. Não existindo um *corpus* de referência e não sendo consensual a definição do que é uma *MWU* correcta, os autores avançam com uma medida alternativa de avaliação da *cobertura* dos resultados com base unicamente no número de *MWUs* extraídas consideradas correctas. De acordo com a avaliação efectuada em (Dias *et al*. 1999a)[22], os autores defendem que com o seu método é possível obterem-se melhores resultados que com métodos anteriores baseados quer em assunções linguísticas quer em métodos numéricos associados a limiares a partir dos quais um *n-grama* é reconhecido como *MWU*.

[22] Gaël Dias na sua Tese de Doutoramento (Dias 2002), ainda não publicada, tem uma avaliação completa do método proposto.

3.1 *CORPUS*

Um *corpus* é composto, normalmente, por vários documentos e está associado a uma língua, a um alfabeto ou conjunto de símbolos, a um formato e a um conjunto de regras para agrupar os diferentes símbolos em *unidades lexicográficas*. É este conjunto de regras, designado por gramática, que permite retirar o significado lexicográfico, sintáctico e semântico dum texto. No âmbito desta dissertação são importantes, unicamente, as regras básicas que permitem a identificação das diferentes *unidades lexicográficas*.

Neste e no próximo capítulo, sem perca de qualquer generalidade, vamos tomar o *corpus* como uma sequência de *tokens* dividida em documentos. Cada *token* é assumido como sendo um número inteiro positivo maior que zero associado univocamente a uma *unidade lexicográfica*. O valor zero fica reservado para a representação dos *gaps*. Tomando o *corpus* como uma sequência de números inteiros, garante-se uma total independência das estruturas de dados e dos algoritmos em relação à língua e ao formato em que o *corpus* está baseado. Permite ainda a extensão de todas as conclusões tiradas, a *corpus* constituídos por sequências de elementos sem qualquer relação com línguas naturais, como sejam, sequências de ADN (Gusfield 1999), (Burkhardt *et al*. 1999). Esta restrição à definição de *corpus* não resulta em qualquer limitação em relação ao tipo de *corpus* que pode ser utilizado. No entanto, obriga ao tratamento prévio do *corpus* por forma a transformá-lo numa sequência de inteiros. Esta obrigação, mais uma vez, não é uma restrição mas sim uma oportunidade para que diferentes métodos de transformação sejam utilizados e logo para que diferentes tipos de análises possam ser postas em prática.

3.2 *N-GRAMAS*

Neste capítulo e nos próximos, entenda-se um *n-grama* como uma sequência, contígua ou não contígua, de *tokens* retirada dum *corpus*, mantendo a ordem e as posições pelas quais os diferentes *tokens* surgem no *corpus*. Concretizemos com um exemplo prático. Se tomarmos um *corpus* constituído pela sequência de *tokens*, [A B C D E F G H I J K L M][23], então [A B C], [A _ C] e [G _ I _ K L] são exemplos de *n-gramas* retirados deste *corpus*. A sequência [A J] não é um *n-grama* válido neste *corpus* dado que "J" não segue a "A" no *corpus*.

No exemplo anterior, recorreu-se a uma representação intuitiva dos *n-gramas* como sequências de *tokens* e *gaps*. Nesta representação, o caracter "_" simboliza a ausência dum *token* numa sequência, ou seja um *gap*. Esta representação, apesar de graficamente ser a mais natural, não é a mais adequada para ser utilizada em expressões algébricas que envolvam *n-gramas*. Em (Dias *et al.* 1999a) e nos outros documentos atrás citados, é proposta uma representação alternativa, mais adequada ao

[23] Nos exemplos utilizam-se letras para representar *tokens* somente para facilitar a compreensão dos mesmos.

tratamento algébrico, onde somente são incluídos os *tokens* e as respectivas distâncias em relação a um elemento pivô. Assim, o *n-grama* [G _ I _ K L], dado atrás como exemplo, passa a ser representado por [G +2I +4K +5L], tal como ilustrado na Figura 3.1.

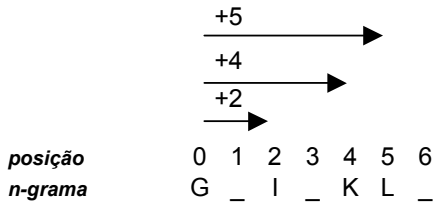


Figura 3.1 – Distâncias em relação ao elemento pivô

Duma forma genérica, um *n-grama* é representado por [$p_{11}w_1$, $p_{12}w_2$, $p_{13}w_3$, ... , $p_{1i}w_i$, ... , $p_{1m}w_m$] onde w_i representa um *token* e p_{1i} denota a distância, em número de posições, entre o *token* w_i e o *token* w_1. A primeira distância, p_{11}, é sempre zero, pelo que é normalmente omitida. Nesta representação foi escolhido o primeiro elemento como elemento pivô, ou seja, aquele a partir do qual se medem as distâncias. No entanto, qualquer dos *tokens* do *n-grama* pode ser escolhido como elemento pivô (Silva *et al.* 1999). Por exemplo, o *n-grama* [G _ I _ K L], anteriormente apresentado, poderia ser representado de forma completamente equivalente por [I -2G +2K +3L], alterando somente o elemento pivô e as distâncias.

3.2.1 DIMENSÃO DUM N-GRAMA

Definição: A dimensão dum *n-grama w*, representado pela função *Lenght(w)*, é a distância em número de posições, ocupadas ou não por um *token*, entre o primeiro e o último *token* do *n-grama*, incluindo os extremos. Para o cálculo da dimensão não são incluídas posições vazias para a esquerda do primeiro *token* ou para a direita do último *token*.

Por exemplo, o *n-grama* [A _ _ D E F] tem dimensão 6.

3.2.2 JANELA DE ANÁLISE

Um *n-grama*, apesar de poder ser utilizado de forma independente, só faz sentido, para esta dissertação, no contexto dum *corpus*.

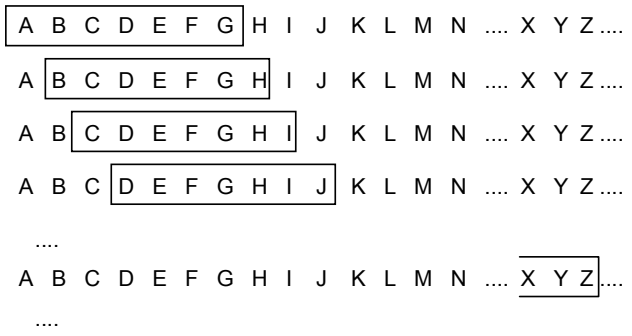


Figura 3.2 – Janela de análise deslizante. Geração dos *n-gramas* dum corpus para $m = 7$.

O conjunto ☚ de todos os *n-gramas* associados a um *corpus* é gerado fazendo deslizar sobre o *corpus* uma janela de análise de dimensão *m* e para cada posição da janela gerar os *n-gramas* respectivos de acordo com as regras adiante enunciadas.

3.2.3 NÚMERO DE TOKENS E DE GAPS DUM N-GRAMA

Definição: A função *NumOfTokens(w)* representa o número de *tokens* contidos num *n-grama w*, ou seja, o número de posições ocupadas por um *token*.

Por vezes é igualmente interessante determinar o número de *gaps* associados a um *n-grama*. Neste particular existem duas medidas importantes:

- O número de *gaps* existentes entre o primeiro e o último *token* do *n-grama*. Este valor é dado pela diferença entre *Lenght(w)* e *NumOfTokens(w)*.

  Por exemplo, o *n-grama* [A _ _ D E F], tem 6 – 4 = 2 *gaps* entre o primeiro e o último *token*;

- O número de *gaps* incluídos na dimensão da janela de análise. Neste caso o número de *gaps* é dado por *m* - *NumOfTokens(w)*.

  O mesmo *n-grama* [A _ _ D E F], dado atrás como exemplo, tem 7 – 4 = 3 *gaps* no espaço de uma janela de análise de $m = 7$ posições.

3.2.4 ESCOLHA DO ELEMENTO PIVÔ

Não colocando qualquer restrição às regras de criação de *n-gramas*, então cada sequência contígua de *m tokens* retirada dum *corpus* tem associada um conjunto de *n-gramas* correspondente a todas as combinações possíveis dos vários *tokens* dentro da janela de análise.

Definição: O número total de *n-gramas*, M_T, que pode ser gerado a partir de uma sequência de *m tokens* é dado por:

$$M_T = \sum_{i=1}^{m} C_i^m = \sum_{i=1}^{m} \frac{m!}{i!(m-i)!} = m! \sum_{i=1}^{m} \frac{1}{i!(m-i)!} \qquad \text{(Erro! Estilo não definido.}.1)$$

Para $m = 7$ existe um total de 127 combinações ou *n-gramas* associados.

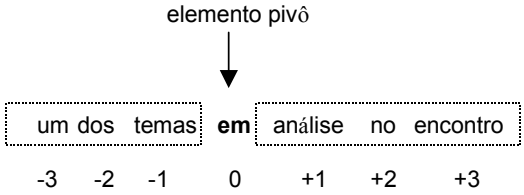


Figura 3.3 – Escolha do elemento central como pivô

Dias *et al.* (2000a) defendem, com base no trabalho desenvolvido por outros autores, que o elemento pivô deve ser o elemento central duma sequência de *tokens*. Em consequência, passam a existir distâncias positivas e distâncias negativas em relação ao elemento pivô central, como mostrado na Figura 3.3. Adicionalmente defendem um máximo de 3 *tokens* para a esquerda do elemento central e igual número de *tokens* para a direita, ou seja, uma janela de análise de 7 elementos.

Definição: A dimensão da janela de análise, *m*, é obrigatoriamente um número inteiro ímpar.

Esta definição é consequência do parágrafo anterior e é uma restrição fundamental para garantir, sempre, a existência dum elemento pivô central. Leva às seguintes definições importantes para os algoritmos implementados.

Definição: Para uma janela de dimensão *m,* o número de posições à direita e à esquerda da posição central é igual a:

$$\frac{(m-1)}{2} \qquad \text{(Erro! Estilo não definido.}.2)$$

Adicionalmente é imposto que o elemento pivô pertença a todos os *n-gramas* associados a cada sequência, conduzindo à seguinte definição para a distância máxima entre dois *tokens* num *n-grama*.

Definição: A distância máxima, d_{max}, ou número máximo de *gaps* consecutivos entre dois *tokens* num *n-grama* é dada por:

$$d_{\max} = \frac{(m-1)}{2} - 1 \qquad \text{(Erro! Estilo não definido.}.3)$$

Assim, a cardinalidade do conjunto de *n-gramas* válidos gerados a partir de cada posição da janela de análise é reduzida e passa a ser dada pela seguinte definição.

Definição: O número de *n-gramas* válidos, *M*, gerados a partir duma sequência de *tokens* de dimensão *m* é dado por:

$$M = \sum_{i=0}^{m-1} C_i^{m-1} = \sum_{i=0}^{m-1} \frac{(m-1)!}{i!(m-1-i)!} = (m-1)! \sum_{i=0}^{m-1} \frac{1}{i!(m-1-i)!} \text{ (Erro! Estilo não definido.}.4)$$

Como um dos elementos é sempre fixo, então o número de elementos que varia é igual a $m - 1$. Desta forma o número total de *n-gramas* gerados é dado pelo total de combinações possíveis de m - 1 elementos.

Tomando como exemplo o mesmo *corpus* utilizado no início deste capítulo e tomando uma janela de análise de dimensão 7 é apresentado no Anexo A, primeira coluna, o conjunto dos 64 *n-gramas* gerados para a primeira posição da janela de análise.

3.2.5 GERAÇÃO DOS N-GRAMAS DUM CORPUS

Se a geração de *n*-gramas estiver baseada num elemento pivô central verifica-se que existe repetição, de alguns dos *n-gramas*, para as várias posições consecutivas da janela de análise. No Anexo A é evidenciada esta repetição através dum exemplo baseado no *corpus* [A B C D E F G H I J K] e para uma janela de análise de dimensão 7. As repetições estão assinaladas a cinzento.

A repetição de *n-gramas* é desnecessária, leva a uma menor eficiência e a uma maior complexidade dos algoritmos de geração dos *n-gramas* dado que estes terão que se preocupar em detectar essas mesmas repetições. Por forma a eliminar as repetições, garantindo o mesmo resultado final em termos do conjunto de *n-gramas* gerado, propõe-se nesta dissertação que a geração seja baseada no primeiro *token* de cada janela de análise, obrigando, com base numa restrição adicional, que o conjunto de *n-gramas* final associado ao *corpus*, seja precisamente o mesmo, depois de eliminadas as repetições, que o obtido tomando o *token* central como pivô.

A restrição adicional obriga a que todos os *n-gramas* gerados possam ser "alinhados" dentro da janela de análise, deslizando para a direita, excluindo posições vazias à direita e inserindo posições vazias à esquerda, por forma a que na posição central, (m / 2) + 1, fique obrigatoriamente um *token*. Por exemplo, para $m = 7$, o *n-grama* [A _ _ _ _ F G] é inválido dado que não é possivel "alinhar" um *token* com a posição central, enquanto que o *n-grama* [A _ C _ E F _] é válido dado que, por uma simples translação para a direita, é possível colocar o *token* "C" como elemento central, ou seja, obter o *n-grama* [_ A _ C _ E F][24].

[24] A representação do *gap* à esquerda do primeiro *token* é desnecessária e pouco usual. É aqui utilizada para realçar a operação de translação dada como exemplo.

Voltando ao exemplo baseado no *corpus* [A B C D E F G H I J K] e $m = 7$, verifica-se que o número de *tokens* gerados em cada posição da janela de análise tomando o elemento central como pivô é, como vimos anteriormente, de $M = 64$ *n-gramas*. Utilizando o primeiro elemento como base geram-se apenas $M' = 43$ *n-gramas*[25] por cada posição da janela de análise, coincidindo 32 *n-gramas* com os anteriormente gerados e sendo os restantes 32 *n-gramas* gerados nas seguintes 3 posições da janela. Todos os *n-gramas* assim gerados terão o mesmo *token* na primeira posição[26]. Este exemplo está ilustrado no Anexo B.

Definição: O número total de *n-gramas* gerados a partir dum *corpus* de dimensão N é dada por:

$$T = M^{'} \times N - Fc \qquad \text{(Erro! Estilo não definido.}.5)$$

Onde $M^{'}$ é o valor revisto, sem repetições, para o número de *n-gramas* válidos gerados em cada posição da janela de análise, obtido pela utilização do primeiro elemento da janela como elemento base e Fc um factor de correcção igual ao número de *n-gramas* não gerados para as últimas posições de cada documento.

Sendo que a última janela de análise coincide com os últimos m tokens de cada documento e dado que se consideram como não adjacentes os últimos *tokens* dum documento em relação aos primeiros *tokens* do documento seguinte, é necessário, para sermos precisos, subtrair o número de *n-gramas* não gerados no final de cada documento. O factor de correcção é dado por:

$$Fc = nd \times \sum_{i=1}^{m-1} \sum_{d=i+1}^{m} \#(d) \qquad \text{(Erro! Estilo não definido.}.6)$$

onde nd corresponde ao número de documentos que compõem o *corpus,* d corresponde às várias dimensões que os *n-gramas* podem ter para uma janela de análise de dimensão m e $\#(d)$ uma função que dá o número de *n-gramas* válidos para cada dimensão d. O valor de i corresponde ao número de caracteres que faltam para o final do documento.

Para $m = 7$ o factor de correcção é de 201 *n-gramas* não gerados por cada documento, tal como é mostrado, esquematicamente, na Figura 3.4. O quadro deve ser lido de cima para baixo e da direita para a esquerda. Por exemplo, quando faltam 6 posições para o final do ficheiro (i.e. $i = 6$) não é possível aplicar máscaras de dimensão 7 (i.e. $d = 7$) que, para $m = 7$, são 16. Quando faltam 5 posições para o final do ficheiro têm que ser excluídas as máscaras de dimensão 6 e 7. O mesmo raciocínio

---

[25] Sai fora do âmbito desta dissertação o estabelecimento duma fórmula para o cálculo do número *M'* revisto de *n-gramas* gerados tendo por base a primeira posição. Esta fórmula tem alguma complexidade dadas as restrições impostas. É apresentada na Tese de Doutoramento de Gaël Dias, ainda não publicada (Dias 2002).

[26] Como efeito lateral são gerados mais 10 *n-gramas* correspondentes a combinações dos três primeiros *tokens* do *corpus* que antes não eram abrangidos dado o primeiro elemento central ser o quarto *token*. Estes conjunto de *n-gramas* adicionais somente enriquece o conjunto final em análise alterando ligeiramente os resultados.

deve ser aplicado para os restantes valores de i, obtendo-se o total de *n-gramas* não gerados para as posições finais de cada ficheiro.

d	1	2	3	4	5	6	7
#(*d*)	1	1	2	4	7	12	16

i								**Total**
6							16	**16**
5						12	16	**28**
4					7	12	16	**35**
3				4	7	12	16	**39**
2			2	4	7	12	16	**41**
1		1	2	4	7	12	16	**42**
								201

Figura 3.4 – Exemplo de cálculo do número de *n-gramas* não gerados no final de cada documento

3.3 SUB-GRAMAS E SUPER-GRAMAS

Essenciais para a *ME* e principalmente para o algoritmo *GenLocalMaxs* são os conceitos de *sub-grama* e de *super-grama*. Em conjunto definem a envolvente de cada *n-grama* ou seja o conjunto de *n-gramas* que lhe estão próximos e sobre os quais actua o algoritmo *GenLocalMaxs* na procura dum máximo local.

3.3.1 SUB-GRAMAS

Para cada *n-grama w* existe um conjunto de *sub-gramas*, $\oslash_{n-1}(w)$, formado por todos os *n-gramas* gerados, a partir de *w*, substituindo, à vez, cada um dos seus *tokens* por um *gap*.

Tomando como exemplo o *n-grama* [I -2G +2K +3L] e $m = 7$, o conjunto dos seus *sub-gramas* é {[G +4K +5L], [I +2K +3L], [I -2G +3L], [I -2G +2K]} (ver Figura 3.5).

posição	0	1	2	3	4	5	6	
n-grama	G	_	I	_	K	L	_	
sub-grama 1	_	_	I	_	K	L	_	***válido***
sub-grama 2	G	_	_	_	K	L	_	***inválido***
sub-grama 3	G	_	I	_	_	L	_	***válido***
sub-grama 4	G	_	I	_	K	_	_	***válido***

Figura 3.5 – Geração do conjunto de *sub-gramas*

Tendo em atenção as regras de validade, atrás enunciadas, que definem o valor máximo admissível (d_{max}) para a distância entre dois *tokens* num *n-grama*, conclui-se que o *n-grama* [G +4K +5L] não é válido dado que entre os *tokens* "G" e "K" existem 3 *gaps* consecutivos quando no máximo poderiam

existir (7 – 1) / 2 –1 = 2 *gaps*. Assim, o conjunto dos *sub-gramas* válidos do *n-grama* [I -2G +2K +3L] reduz-se a {[I +2K +3L], [I -2G +3L], [I -2G +2K]}.

3.3.2 SUPER-GRAMAS

Enquanto que um *sub-grama* é formado pela substituição dum *token* por um *gap*, um *super-grama* é formado pela operação inversa, ou seja, pela substituição dum *gap* por um *token*. Os *n-gramas* assim formados deverão, obrigatoriamente, pertencer ao conjunto de *n-gramas* válidos gerados desde o *corpus* que estiver a ser utilizado pelo que o conjunto de *tokens* que pode ser utilizado para substituir cada *gap* limita-se ao formado por todos os *tokens* que ocorrem simultaneamente, na posição em análise, com o *n-grama* a partir do qual está a ser formado o *super-grama*.

Tomando como exemplo o *n-grama* w = [D E _ G] e m = 7 (ver Figura 3.6), o conjunto de *super-gramas*, $\oslash_{n+1}(w)$ associados é o seguinte: {[C D E _ G]; [B _ D E _ G]; [A _ _ D E _ G]; [D E F G]; [D E _ G H]; [D E _ G _ I]; [D E _ G _ _ J]}. Onde A, B, C, F, H, I e J são *tokens* que ocorrem em simultâneo com o *n-grama* [D E _ G] nas várias ocorrências desse *n-grama* no *corpus*.

Se a dimensão do *n-grama* coincidir com a dimensão m da janela de análise (e.g. A _ _ D _ _ G) então $\oslash_{n+1}(w)$ é somente composto pelos *super-gramas* formados pela substituição individual de cada *gap* interno por um *token* válido não havendo lugar para a formação de *super-gramas* por adição de *tokens* à esquerda ou à direita de w.

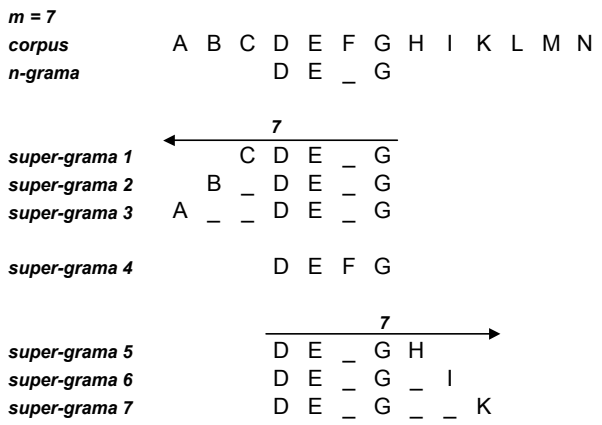


Figura 3.6 – Geração do conjunto de *super-gramas*

3.4 FREQUÊNCIA

Definição: No âmbito deste trabalho, *frequência* corresponde ao número de ocorrências ou repetições dum determinado *n-grama* num *corpus*.

Neste trabalho é igualmente utilizado o termo *frequência* como sendo o número de ocorrências dum determinado *token* num *corpus*.

3.5 EXPECTATIVA MÚTUA

A *ME* é uma medida normalizada de associação para *n-gramas*, que mede a força da ligação existente entre os vários *tokens* dum *n-grama*, avaliando o impacto da perca de cada um deles no valor do conjunto. Sequências de *tokens*, contíguas ou não contíguas, fortemente ligadas corresponderão a valores de *ME* elevados (Dias et al. 1999).

Definição: De acordo com (Dias *et al*. 2000d) a *ME* é dada por:

$$ME(w) = p(w) \times NE(w) \quad \textbf{(Erro! Estilo não definido.}.7)$$

onde *w* é um *n-grama*, *p*(*w*) é a probabilidade de ocorrência do *n-grama w* no *corpus* em análise e *NE*(*w*) é a medida normalizada da expectativa[27] associada a *w*.

Definição: A *NE* associada a uma sequência de *m tokens* é definida como sendo a expectativa média da ocorrência dum *token* numa determinada posição, conhecendo-se a ocorrência dos outros $m - 1$ *tokens*, igualmente restringidos às suas posições (Dias *et al.* 2000d), e é dada por:

$$NE(w) = p(w) / FPE(w) \quad \textbf{(Erro! Estilo não definido.}.8)$$

onde *FPE(w)*[28] é a média das probabilidades de ocorrência de cada *sub-grama de w* no *corpus*.

Definição: A média das probabilidades de ocorrência dos *sub-gramas* de w = $[w_1 p_{12} w_2 p_{13} w_3 \ldots p_{1t} w_t]$, FPE*(w)*, é dada por (Dias *et al.* 2000d):

$$FPE(w) = \frac{p([w_2 p_{23} w_3 p_{24} w_4 \ldots p_{2t} w_t]) + \sum_{i=2}^{t} p([w_1 p_{12} w_2 \ldots \hat{p}_{1i} \hat{w}_i \ldots p_{1t} w_t])}{t} \quad \textbf{(Erro! Estilo não definido.}.9)$$

onde o símbolo "^" representa o *token* a eliminar, em cada passo, para a formação de cada *sub-grama*. O valor de *t* corresponde ao número de *sub-gramas* válidos gerados a partir do *n-grama w*, ou seja, a cardinalidade de $\otimes_{n-1}(w)$, que será igual ao número de *tokens* de *w* se todos os *sub-gramas* de *w* forem válidos de acordo com as regras de validade atrás enunciadas.

---

[27] *Normalized Expectation Measure* (*NE*) no original

[28] *Fair Point of Expectation*

Definição: No contexto da análise que está a ser efectuada, a probabilidade de ocorrência do *n-grama* *w* no *corpus*, pode ser aproximada à divisão da *frequência* de *w* pelo número total de *n-gramas* de igual dimensão de *w*, ou seja, pela dimensão *N* do *corpus* (Dias *et al.* 2000d).

$$p(w) = f(w)/N \qquad \text{(Erro! Estilo não definido.}.10)$$

onde $f(w)$ é a *frequência* ou número de vezes que o *n-grama* *w* ocorre no *corpus*.

Tendo em atenção o que atrás foi enunciado e para $w = [w_1 p_{12} w_2 p_{13} w_3 \ldots p_{1t} w_t]$, a *ME*, é dada, de forma compacta, por:

$$ME(w) = \frac{f^2(w)}{N \times \frac{f([w_2 p_{23} w_3 p_{24} w_4 \ldots p_{2t} w_t]) + \sum_{i=2}^{t} f([w_1 p_{12} w_2 \ldots \hat{p}_{1i} \hat{w}_i \ldots p_{1t} w_t])}{t}} \quad \text{(Erro!}$$

Estilo não definido..11)

Um caso particular importante dá-se quando *w* tem somente dois *tokens*, ou seja, *NumOfTokens*(*w*)=2.

$$ME(w = [w_1 w_2]) = \frac{f^2(w)}{N \times \frac{f(w_1) + f(w_2)}{2}}$$

(**Erro! Estilo não**

$$para \quad NumOfTokens(w) = 2$$

definido..12)

onde $f(w_1)$ e $f(w_2)$ representam, respectivamente, a *frequência* do *token* w_1 e do *token* w_2 no *corpus*, o que é totalmente equivalente à *frequência* dos *n-gramas* de dimensão unitária iniciados por w_1 e por w_2. Esta constatação tem, como veremos, implicações importantes ao nível da implementação da *ME* e do *GenLocalMaxs*.

3.6 ALGORITMO GENLOCALMAXS

Como já foi referido, Dias *et al.* (1999a) propõem que o reconhecimento de *MWUs* seja baseado num método, puramente estatístico e totalmente independente de qualquer conhecimento prévio, seja ele de natureza linguística ou de outra natureza, relativamente ao *corpus* em análise. Designam este método por algoritmo *GenLocalMaxs*.

O *GenLocalMaxs* é um algoritmo simples mas eficaz. Baseia-se na procura do máximo local da função de associação quando esta é aplicada ao conjunto de *n-gramas* na envolvente dum dado *n-grama*,

incluindo o próprio. Se o valor da medida de associação do *n-grama* em análise for um máximo local então esse *n-grama* é uma *MWU* (Dias *et al.* 1999a).

Definição: De acordo com (Dias *et al*. 2000d), a condição para um *n-grama w* ser uma *MWU* é dada por:

w é uma MWU se e só se …x ? $\oslash_{n-1}(w)$, …y ? $\oslash_{n+1}(w)$ (**Erro! Estilo não definido.**.13)

$g(w) > g(y)$ para $NumOfTokens(w) = 2$;

$g(w)$ µ $g(x)$ ß $g(w) > g(y)$ para $NumOfTokens(w) > 2$;

onde $g(w)$[29] é uma medida de associação para *n-gramas*. Esta é a condição base do algoritmo *GenLocalMaxs*.

Sendo um algoritmo genérico pode ser utilizado em conjunto com qualquer medida de associação que seja crescente, ou seja, onde valores mais altos correspondam a casos mais relevantes. Esta dissertação limitou-se unicamente à procura duma implementação eficiente do *GenLocalMaxs* em conjunção com a *ME* não tendo sido abordada a utilização de outras medidas de associação.

Por definição, o *GenLocalMaxs* obriga a que as *MWUs,* seja qual for a sua dimensão, tenham um mínimo de duas posições ocupadas com um *token*, excluindo, deste modo, os *n-gramas* unitários da lista de potenciais *MWUs*. Esta restrição é utilizada para reduzir o espaço necessário pela solução apresentada no capítulo 4.

3.7 PROPRIEDADES DA FREQUÊNCIA DOS N-GRAMAS

Propriedade 1: Sejam $w = [w_1\ w_2\ ...w_k]$[30] e $u = [w_1\ w_2\ ..\ .w_k\ w_{k+1}]$, dois *n-gramas* gerados a partir dum mesmo *corpus*. Genericamente o *n-grama u* conterá o *n-grama w* podendo w_{k+1} substituir qualquer *gap* de *w* ou ser adicionado à esquerda ou à direita do mesmo.

Nestas condições, verifica-se que:

[29] A utilização da letra "g" para a função genérica de medida de associação prende-se com o facto da medida de associação ser normalmente associada a uma espécie de "cola" existente entre os *tokens*. Cola em Inglês é *glue* razão da escolha (Silva *et al. 1999*).

[30] Para que o texto resulte menos denso utilizaremos daqui para a frente uma representação simplificada para os n-gramas. Em vez de $[w_1\ p_{12}w_2\ p_{13}w_3\ ...\ p_{1i}w_i,\ ...\ p_{1m}w_m]$ utilizaremos abreviadamente $[w_1\ w_2,\ w_3\ ...w_m]$ sempre que tal não gere dúvidas.

$$\forall w,u \in \Delta_d$$
$$w = [w_1 w_2 ... w_k] \quad u = [w_1 w_2 ... w_k w_{k+1}]$$
$$f(u) \leq f(w)$$

(**Erro! Estilo não definido.**.14)

Esta propriedade pode ser generalizada a quaisquer duas sequências de *tokens*, contíguas ou não contíguas, em que a segunda contenha a primeira.

Resumidamente, esta propriedade diz-nos que quando é acrescentado um *token* a uma sequência de *tokens*, a *frequência* da sequência resultante é sempre menor ou no limite igual à *frequência* da sequência base.

A veracidade desta propriedade pode facilmente ser compreendida se tomarmos *w* como uma sub-parte de *u*. Nesta situação *u* não pode ocorrer mais vezes que *w*. No limite poderá ocorrer o mesmo número de vezes. Por outro lado, *w* pode ocorrer como sub-parte de outras sequências que não *u*, logo a sua *frequência* poderá ser maior que a de *u*.

Podemos igualmente concluir que a *frequência* de uma sub-sequência é sempre igual ou superior à *frequência* das sequências em que está inserida. Por outras palavras, quando se retira um ou mais *tokens* a uma sequência, a *frequência* da sequência resultante tem tendência para ser maior ou no mínimo igual à da sequência original. Nunca poderá ser menor.

Como consequência da propriedade 1, se um determinado *n-grama* ocorre uma única vez em ☚, então todos os *n-gramas* que contenham o primeiro só podem ocorrer uma única vez.

$$\forall w,u \in \Delta_d$$
$$w = [w_1 w_2 ... w_k] \quad u = [w_1 w_2 ... w_k w_{k+1}]$$
$$se \quad f(w) = 1 \quad então \quad f(u) = 1$$

(**Erro! Estilo não definido.**.15)

Exclui-se a possibilidade da *frequência* ser zero (menor que um) dado que estamos a considerar como condição base que *u* existe no conjunto ☚$_d$ de *n-gramas* associados ao *corpus*.

Esta observação pode ser generalizada a quaisquer duas sequências de *tokens*, contíguas ou não contíguas, em que a segunda contenha a primeira e esta ocorra uma única vez em ☚.

3.8 *PROPRIEDADES DA EXPECTATIVA MÚTUA*

Uma das consequências mais importantes da propriedade 1, tem impacto directo no cálculo da *ME*. A *frequência* associada aos *sub-gramas* de *w* é sempre igual ou superior à *frequência* de *w*, dado que os *sub-gramas* de *w* são sub-partes de *w*.

$$\begin{array}{l}\forall w \in \Delta_d \quad \forall u \in \Omega_{n-1}(w) \\ f(u) \geq f(w)\end{array} \qquad \textbf{(Erro! Estilo não definido.}.16)$$

Por outras palavras, no cálculo da *ME* de qualquer *n-grama* só participam *sub-gramas* cuja *frequência* seja igual ou superior à *frequência* do próprio *n-grama*.

3.9 *PROPRIEDADES DO GENLOCALMAXS* EM CONJUNTO COM A *ME*

Propriedade 2: Seja $w = [w_1\ w_2\ ...w_k]$ χ ☚. Se $f(w) = 1$ então *w* nunca será uma *MWU*.

Esta propriedade é uma importante constatação porque indica que somente *n-gramas* que ocorram mais que uma vez no conjunto dos *n-gramas* gerados a partir do *corpus* podem ser eleitos como *MWU*.

Prova: A prova desta propriedade não é evidente pelo que vamos primeiro demonstrar para um *n-grama* de dimensão 2 e só depois generalizar a prova a *n-gramas* de qualquer dimensão.

Sabemos que as *ME* de $w = [w_1\ w_2]$ χ ☚ e de $[w_1\ w_2\ w_3]$ χ $\oslash_{n+1}(w)$, são dadas respectivamente por:

$$ME([w_1 w_2]) = \frac{f^2([w_1 w_2])}{N \times \frac{f(w_1) + f(w_2)}{2}} \quad \textbf{(Erro! Estilo não definido.}.17)$$

$$ME([w_1 w_2 w_3]) = \frac{f^2([w_1 w_2 w_3])}{N \times \frac{f([w_1 w_2]) + f([w_1 w_3]) + f([w_2 w_3])}{3}} \quad \textbf{(Erro! Estilo não definido.}.18)$$

Por hipótese, $f([w_1\ w_2]) = 1$, logo, pela observação **Erro! Estilo não definido.**.15, $f([w_1\ w_2\ w_3]) = 1$, qualquer que seja $[w_1\ w_2\ w_3]$ χ $\oslash_{n+1}(w)$. Assim, podemos rescrever **Erro! Estilo não definido.**.17 e **Erro! Estilo não definido.**.18 como:

$$ME([w_1 w_2]) = \frac{1}{N \times \frac{f(w_1) + f(w_2)}{2}} \quad \textbf{(Erro! Estilo não definido.}.19)$$

$$ME([w_1 w_2 w_3]) = \frac{1}{N \times \frac{1 + f([w_1 w_3]) + f([w_2 w_3])}{3}}$$ (**Erro! Estilo não definido.**.20)

Queremos provar que:

$$\forall w = [w_1 w_2] \in \Delta, \forall [w_1 w_2 w_3] \in \Omega_{n+1}(w)$$

$$se \quad f([w_1 w_2]) = 1 \quad então \quad ME([w_1 w_2]) \leq ME([w_1 w_2 w_3])$$ (**Erro! Estilo não definido.**.21)

o que é contrário às condições base do algoritmo *GenLocalMaxs* para que um *n-grama* possa ser eleito como *MWU*.

Temos então que demonstrar que:

$$\frac{f(w_1) + f(w_2)}{2} \geq \frac{1 + f([w_1 w_3]) + f([w_2 w_3])}{3}$$ (**Erro! Estilo não definido.**.22)

Seja:

$$f(w_1) = x_1 \quad e \quad f(w_2) = x_2$$ (**Erro! Estilo não definido.**.23)

então, pela observação **Erro! Estilo não definido.**.15:

$$\begin{aligned} 1 \leq f([w_1 w_3]) \leq x_1 \\ 1 \leq f([w_2 w_3]) \leq x_2 \end{aligned}$$ (**Erro! Estilo não definido.**.24)

Considerando os valores máximos, temos que provar que:

$$\frac{x_1 + x_2}{2} \geq \frac{1 + x_1 + x_2}{3}$$ (**Erro! Estilo não definido.**.25)

Fazendo $x_1 + x_2 = a$, com a positivo maior que 2, podemos rescrever **Erro! Estilo não definido.**.25 como:

$$\frac{a}{2} \geq \frac{1 + a}{3}$$ (**Erro! Estilo não definido.**.26)

Dividindo a / 2 por (1 + a) / 3, facilmente verifica-se que a primeira fracção é sempre maior ou igual à segunda.

$$\frac{\frac{a}{2}}{\frac{(1+a)}{3}} = \frac{3a}{2+2a} = \frac{3}{\frac{2}{a}+2} \geq 1 \text{ dado a } f\ 2$$ (**Erro! Estilo não definido.**.27)

Por consequência, para $f([w_1\ w_2])$ = 1, $ME([w_1\ w_2])$ [$ME([w_1\ w_2\ w_3])$, como queríamos provar.

Generalizemos agora a prova para um *n-grama* de qualquer dimensão. Tomemos $w=[w_1\ w_2\ ...w_{k-1}]$ χ ☚ e $f([w_1\ w_2\ ...w_{k-1}])$ = 1. Igualmente, pela observação **Erro! Estilo não definido.**.15, se $f([w_1\ w_2\ ...w_{k-1}])$ = 1 então $f([w_1\ w_2\ ...w_{k-1}\ w_k])$ = 1 para qualquer $[w_1\ w_2\ ...w_{k-1}\ w_k]$ χ $\oslash_{n+1}(w)$.

Queremos então provar que:

$$\forall w = [w_1 w_2 ... w_k] \in \Delta, \forall [w_1 w_2 .... w_k w_{k+1}] \in \Omega_{n+1}(w)$$

$$se \quad f([w_1 w_2 ... w_k]) = 1 \quad então \quad ME([w_1 w_2 ... w_k]) \leq ME([w_1 w_2 .... w_k w_{k+1}])$$ (**Erro! Estilo não definido.**.28)

Os valores para a *ME* de $[w_1\ w_2\ ...w_{k-1}]$ e de $[w_1\ w_2\ ...w_{k-1}\ w_k]$ são dadas respectivamente por:

$$ME([w_1 w_2 ... w_{k-1}]) = \frac{1}{N \times \frac{\sum_{i=1}^{k-1} f([w_1 w_2 ... \hat{w}_i ... w_{k-1}])}{k-1}}$$ (**Erro! Estilo não definido.**.29)

$$ME([w_1 w_2 ... w_{k-1} w_k]) = \frac{1}{N \times \frac{\sum_{i=1}^{k} f([w_1 w_2 ... \hat{w}_i ... w_{k-1} w_k])}{k}}$$ (**Erro! Estilo não definido.**.30)

Podemos transformar a equação **Erro! Estilo não definido.**.29 dando-lhe o seguinte aspecto:

$$ME([w_1 w_2 ... w_{k-1}]) = \frac{1}{N \times \frac{\sum_{i=1}^{k-1} x_i}{k-1}}$$ (**Erro! Estilo não definido.**.31)

$$onde \quad x_i = f([w_1 w_2 ... \hat{w}_i ... w_{k-1}])$$

Pela observação **Erro! Estilo não definido.**.15:

$$se \quad x_i = f(\left[w_1 w_2 ... \hat{w_i} ... w_{k-1} \right])$$
$$então \quad 1 \le f(\left[w_1 w_2 ... \hat{w_i} ... w_{k-1} w_k \right]) \le x_i$$

(**Erro! Estilo não definido.**.32)

Considerando os valores máximos, temos que provar que:

$$\frac{\sum_{i=1}^{k-1} x_i}{k-1} \ge \frac{1+\sum_{i=1}^{k-1} x_i}{k}$$

(**Erro! Estilo não definido.**.33)

O 1 no numerador do segundo elemento da expressão **Erro! Estilo não definido.**.33 advém do facto de, quando o *token* eliminado é w_k, a frequência a avaliar é a do *n-grama* [w_1 w_2 ...w_{k-1}] que, por premissa, é 1.

Fazendo:

$$a = \sum_{i=1}^{k-1} x_i$$

(**Erro! Estilo não definido.**.34)

podemos rescrever **Erro! Estilo não definido.**.33 como:

$$\frac{a}{k-1} \ge \frac{1+a}{k}$$

(**Erro! Estilo não definido.**.35)

Voltando a determinar a relação de ordem dos dois lados da inequação através duma divisão obtemos:

$$\frac{a/(k-1)}{(1+a)/k} = \frac{ka}{(k-1)(1+a)} = \frac{ka}{k+ka-1-a} = \frac{k}{\frac{k}{a}+k-\frac{1}{a}-1} =$$
$$= \frac{k}{\frac{k-1}{a}+k-1} \in \left[1, \frac{k}{k-1}\right[\ge 1 \qquad a \ge k-1$$

(**Erro! Estilo não definido.**.36)

donde podemos concluir que o denominador de **Erro! Estilo não definido.**.29 é sempre superior ou igual ao denominador de **Erro! Estilo não definido.**.30 donde *ME*([w_1 w_2 ...w_{k-1}]) [*ME*([w_1 w_2 ...w_{k-1} w_k]), para *f*([w_1 w_2 ...w_{k-1}]) = 1, como queríamos provar.

Como veremos, a aplicação desta propriedade, conduz a reduções drásticas no universo de procura de *MWUs*, com um impacto positivo, muito importante, ao nível da eficiência, espacial e temporal, do sistema. Somente *n-gramas* com frequência não unitária, em número muito inferior aos *n-gramas* de frequência unitária, são potenciais *MWUs*.

4 SOLUÇÃO PROPOSTA

Enunciado o problema, enquadrado dentro do domínio da *IR*, e exposta a base teórica que lhe está subjacente, chegou o momento de apresentar a solução proposta para a sua resolução em termos duma implementação eficiente, em espaço e tempo[31], na forma dum programa de computador.

Em termos sumários o que é requerido é a procura duma implementação eficiente do algoritmo *GenLocalMaxs* com vista à extracção do conjunto de *MWUs*, associado a um determinado *corpus*, sabendo que existe uma íntima dependência, no caso em estudo, do *GenLocalMaxs* em relação ao cálculo da *ME* associada a cada *n-grama* gerado a partir do *corpus*. De acordo com o proposto, a solução está comprometida com as propriedades próprias da *ME*, maximizando a utilização das mesmas, na procura duma solução, o mais eficiente possível, para a conjugação do *GenLocalMaxs* com a *ME*. Neste contexto, não foi analisada a generalização da solução a outras medidas de associação.

São igualmente premissas do problema as duas seguintes:

- A solução deve suportar *corpus* baseados em *dicionário*s Σ extensos;
- A solução deve ser eficiente, em termos de espaço de memória ocupado, para permitir que sejam manipulados *corpus* com um número elevado de *tokens*. Tanto melhor quantos menos dados forem guardados para um mesmo nível de informação armazenada e para um mesmo grau de eficiência temporal dos algoritmos.

Neste contexto e tendo em atenção as condições **Erro! Estilo não definido.**.13, que resumem o algoritmo *GenLocalMaxs*, e a equação **Erro! Estilo não definido.**.11, para o cálculo da função *ME*, podemos decompor o problema nos seguintes sub-problemas:

1. Cálculo eficiente da *frequência* de cada *n-grama* w ? $\rightarrow_d$, essencial para o cálculo da respectiva *ME*(w);

[31] Quando for referida "eficiência" ao longo deste capítulo deve ser sempre tomada nas suas duas vertentes, a saber, espaço e tempo, não esquecendo a forte dependência normalmente existente entre estas duas vertentes.

2. Busca eficiente do conjunto $\oslash_{n-1}(w)$ associado a cada *n-grama* w ? ☚$_d$, base ao cálculo da respectiva $ME(w)$;

3. Cálculo eficiente da $ME(w)$ de cada *n-grama* w ? ☚$_d$, necessário ao *GenLocalMaxs*;

4. Busca eficiente dos conjuntos $\oslash_{n-1}(w)$ e $\oslash_{n+1}(w)$ para cada *n-grama* w ? ☚$_d$, necessários ao *GenLocalMaxs*.

Estes são os quatro sub-problemas que necessitam ser atacados, em termos da estrutura de dados e dos algoritmos, para a construção duma solução. Adicionalmente é necessário ter em conta a premissa inicial segundo a qual o sistema deve funcionar em memória, maximizando a utilização da mesma. Esta é uma restrição adicional importante dado que os *corpus* são, por natureza, de grande dimensão, da ordem dos muitos milhões de *tokens*. Mesmo para os computadores actuais, em que as quantidades de memória central se medem em *gigabytes*, estes valores são importantes.

O terceiro sub-problema, cálculo eficiente da *ME*, depende fortemente dos dois primeiros, ou seja, resolvendo-se os problemas associados ao cálculo da *frequência* de cada *n-grama* e da busca do conjunto de *sub-gramas* associado a cada *n-grama*, é simples calcular, eficientemente, a respectiva *ME*. A questão principal associada a este sub-problema deriva do facto do valor da *ME* de cada *n-grama* ser potencialmente utilizado várias vezes pelo *GenLocalMaxs*. Sendo assim, para este sub-problema, a questão principal é decidir guardar ou não guardar a *ME*, ou seja, "gastar" ou não "gastar" memória com o armazenamento da *ME* de cada *n-grama*.

A eficiência do cálculo da *ME* de cada *n-grama* poderá depender somente da eficiência do algoritmo de pesquisa de cada *sub-grama* desde que exista um cálculo prévio da *frequência* associada a cada *n-grama*.

A eficiência do *GenLocalMaxs* depende da eficiência do cálculo da *ME* e dos algoritmos de pesquisa dos elementos associados aos conjuntos $\oslash_{n-1}(w)$ e $\oslash_{n+1}(w)$, com especial dificuldade para este último dado que obriga a que o sistema tenha conhecimento sobre a vizinhança ou envolvente de cada *n-grama* no que respeita aos *tokens* válidos para substitutos de cada *gap* nas diversas ocorrências do *n-grama*.

A apresentação da solução é efectuada de forma progressiva, passo a passo, até se atingir a solução proposta. Começa-se por mostrar a solução mais directa, naturalmente menos eficiente e, depois, por refinamentos sucessivos, chega-se à solução final proposta.

Fazendo parte duma solução, construída essencialmente com base na experimentação, as sucessivas aproximações correspondem a soluções, encontradas nas várias etapas do desenvolvimento do protótipo, para os problemas práticos que a cada momento foram sendo colocados. O desenho e a implementação do protótipo, na sua globalidade, são os temas do capítulo 5. No presente capítulo é

detalhada a solução proposta para o módulo responsável pela identificação do conjunto de *MWUs* dum *corpus*.

4.1 PRIMEIRA APROXIMAÇÃO

Como ficou provado pelas várias experiências efectuadas, o principal problema em termos da eficiência global da solução escolhida prende-se com a eficiência do algoritmo subjacente ao cálculo da *frequência* associada a cada *n-grama*. Este problema tem por base a procura duma estrutura adequada para o armazenamento temporário do conjunto ☚ de todos os *n*-gramas associados ao *corpus*, base ao cálculo da *frequência*.

O cálculo da *frequência* não é mais que uma simples contagem do número de ocorrências de cada *n-grama*. Para tal, é necessário ter-se conhecimento sobre o universo completo de todos os diferentes *n-gramas* associados ao *corpus* e, depois, por um método eficiente de contagem, associar-se a cada um deles a respectiva *frequência*.

Tendo em conta o método de criação do conjunto ☚, descrito na secção 3.2.5, baseado na justaposição sucessiva de uma janela de *m* posições sobre a sequência de *tokens,* gerando-se para cada posição os respectivos *n-gramas*, não se conhecendo à partida o número $|☚_d|$ de *n-gramas* diferentes existentes e partindo-se do princípio que os *n*-gramas são independentes entre si, isto é, que não existe forma de os relacionar e assim simplificar o método de contagem, restam-nos basicamente dois métodos para obter-se o número de ocorrências de cada *n-grama*:

1. Gerar e inserir cada *n-grama* numa estrutura ordenada e por efeito quase lateral obter-se o número de repetições de cada um deles;
2. Gerar e guardar o conjunto de *n-gramas*, ordenando à posteriori e obtendo-se, igualmente, por efeito lateral, a *frequência* de cada um deles.

Dado o potencialmente elevado número de *n-gramas*, fica desde logo posta de parte a utilização de estruturas dinâmicas baseadas em apontadores[32], por definição grandes consumidoras de memória devido à sobrecarga imposta pelo armazenamento dos próprios apontadores. Desta forma, a estrutura de dados mais directa para armazenar o conjunto ☚ de todos os *n-gramas* associados ao *corpus* é a formada por um vector[33] de dimensão

$$|\Delta| = M' \times N \qquad \textbf{(Erro! Estilo não definido.}.1)$$

[32] Conhecidos por *pointers* em inglês

[33] Preferimos utilizar, dado existir consenso nesta tradução, o termo vector em vez do termo original a*rray*, em inglês.

de vectores de inteiros de dimensão *m* representando, cada um deles, um *n-grama* como uma sequência de *tokens* e *gaps*.

Numa estrutura tipo vector, operações de inserção ordenada são muito pouco eficientes do ponto de vista temporal. Juntando a esta limitação o facto da estrutura ser criada e não mais modificada, leva a que a opção correcta seja a que aponta para uma ordenação depois de todos os *n-gramas* estarem inseridos no vector. Depois de ordenado, é simples, com uma só passagem, criar o conjunto de todos os diferentes *n-gramas*, $\blacktriangleleft_d$, e associar a cada um deles a respectiva *frequência*. Tendo a frequência e uma função de busca de *n-gramas* sobre o vector ordenado dos diferentes *n-gramas* é possível calcular-se a *ME* associada a cada um deles.

Os algoritmos de ordenação adoptados devem tomar simplesmente em conta o valor relativo de cada *token* no universo dos números inteiros e ordenar cada *n-grama* da esquerda para a direita como se duma ordenação de palavras se tratasse.

A estrutura resultante e os passos a dar são mostrados resumidamente na Figura 4.1.

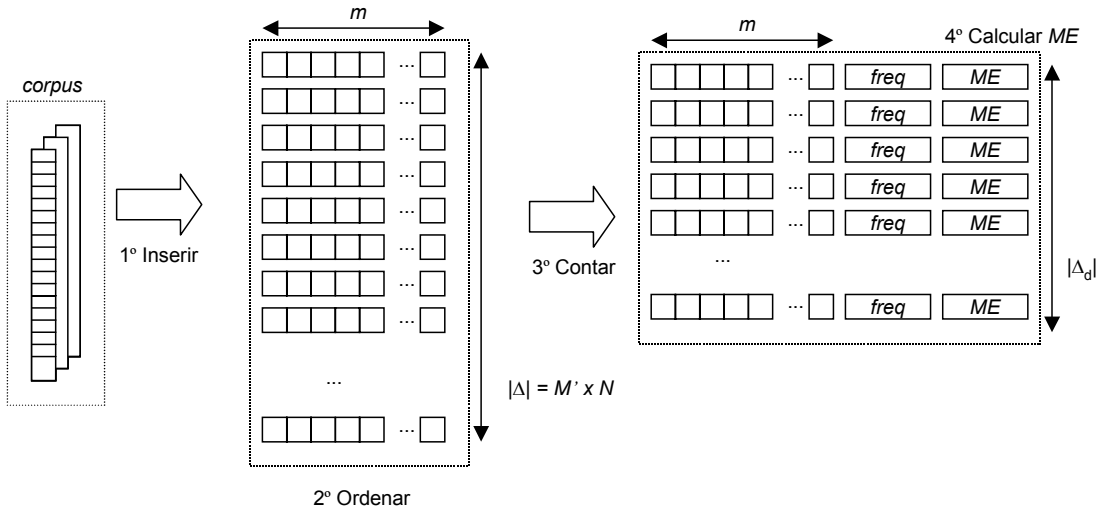


Figura 4.1 – Primeira aproximação

Na Figura 4.1 são mostrados, para tornar mais clara a explicação, dois vectores, um base para armazenar e ordenar todos os *n-gramas* e um segundo para armazenar o conjunto dos diferentes *n-gramas* conjuntamente com os valores da *frequência* e da *ME*. Na prática todas as operações podem ser executadas sobre um mesmo vector reduzindo-se assim o espaço necessário. No entanto, tratando-se, basicamente, dum vector de *strings* o espaço necessário é muito elevado e longe de ser óptimo.

Com esta estrutura é dada resposta aos três primeiros sub-problemas enumerados no início deste capítulo, ficando a eficiência temporal dependente da eficiência das operações de ordenação e de

busca necessárias aos passos 2 e 4 respectivamente. Basear a eficiência do sistema na eficiência dum qualquer algoritmo de pesquisa de *strings* num vector ordenado não é certamente a solução ideal.

Adicionalmente, esta estrutura dá uma resposta muito pouco eficiente a parte do quarto sub-problema no que se refere à determinação do conjunto $\oslash_{n+1}(w)$ para cada w ? $\leftarrow_d$. Como é sabido, a criação do conjunto $\oslash_{n+1}(w)$ depende do conhecimento da vizinhança de cada *n-grama* no *corpus* por forma a que se conheça o conjunto de *tokens* candidatos a preencherem cada um dos seus *gaps*. Com esta estrutura, a única maneira de conhecer a vizinhança de cada *n-grama w* é através de comparações sucessivas de *w* com todos os restantes *n-gramas*, até que se encontrem todos os *n-gramas* cujo número de *tokens* seja superior em uma unidade ao número de *tokens* de *w* (i.e. *NumOfTokens*(*w*) + 1) e nos quais *w* esteja incluído.

4.2 UTILIZAÇÃO DE REFERÊNCIAS

A solução atrás apresentada dá uma resposta muito pouco eficiente, em termos de espaço e de tempo, ao problema apresentado. No entanto, é um bom ponto de referência em termos dos problemas algorítmicos e de eficiência que é necessário resolver.

Procurando soluções alternativas, mais eficientes, em termos do armazenamento dos conjuntos $\leftarrow$ e $\leftarrow_d$ e para o cálculo eficiente da *frequência* e da *ME*, a aposta inicial centrou-se na análise da possível expansão dos resultados obtidos por Yamamoto e Church (2000), apresentados no capítulo 2, para o conjunto dos *sufixos* dum *corpus*[34], ou seja, na adaptação a *n-gramas* não contíguos da estrutura baseada num *suffix-array* proposta por estes autores. Como já foi referido, uma aproximação idêntica é igualmente descrita em (Nagao *et al.* 1994) e em (Ikehara *et al*. 1996). Estes artigos descrevem estruturas que tiram partido da eficiência espacial e temporal das soluções construídas em torno dum *suffix-array*. Yamamoto *et al*. levam mais além a estrutura proposta, apresentando simplificações importantes no método de cálculo da *frequência* e de medidas de associação restringindo-o ao que designam por classes de *substrings*[35]. No entanto, a extensão destas simplificações a *n-gramas* não contíguos não se mostrou viável. Verificou-se não ser igualmente possível estender directamente a *n-gramas* não contíguos os algoritmos optimizados de ordenação de *sufixos*, dado estas optimizações terem por base a exploração de propriedades dos próprios *sufixos* não extensíveis a sequências não contíguas, ou seja, a *n-gramas*.

Do estudo efectuado em torno do trabalho de Yamamoto *et al*., ficou essencialmente a constatação que a adopção duma estrutura baseada em referências, em substituição da representação explícita dos

---

[34] No contexto deste capítulo um *sufixo* pode ser visto como uma sequência de *tokens* iniciada em qualquer posição do *corpus* e que se prolonga até ao final do mesmo.

[35] Yamamoto *et al.* designam por *substring* uma parte dum *sufixo* .

conjuntos ☚ e ☚$_d$, resultaria em ganhos muito importantes em termos do espaço ocupado, ou seja, da eficiência espacial da solução. Como mostraram Manber e Myers (1991), representar o conjunto de todos os *sufixos* é simples e directo, basta um vector de apontadores para todas as posições do *corpus*. Cada posição indica a posição inicial de cada *sufixo* suficiente para a representação do mesmo (ver Figura 2.2).

Será igualmente possível representar duma forma compacta o conjunto de todos os *n-gramas*, contíguos e não contíguos, associados ao mesmo *corpus*?

Antes de apresentarmos a solução proposta nesta dissertação, para dar resposta à questão anterior, convém lembrar que, no âmbito desta dissertação, um *n-grama* é uma sequência, contígua ou não contígua, de *tokens* de dimensão máxima *m*. Apresenta assim, duas diferenças importantes em relação a um *sufixo*: 1º pode ser uma sequência não contígua; 2º tem dimensão máxima limitada por *m*. É igualmente importante realçar que a geração do conjunto de *n-gramas* associado a cada posição da janela de análise é uma operação repetitiva com um padrão de geração bem definido e resultante das regras apresentadas na secção 3.2.5. De salientar que, como foi demonstrado, a melhor solução passa pela adopção dum método de geração baseado na utilização, como elemento base, do *token* correspondente à primeira posição da janela de análise. Este *token* irá constar como primeiro *token* de todos os *n-gramas* gerados a partir de cada posição da janela de análise.

A constatação de que existe um padrão repetitivo associado à geração do conjunto de *n-gramas*, para cada posição da janela de análise, é o ponto de partida para a criação duma estrutura auxiliar que armazene este padrão, ou seja, o conjunto que iremos, de ora em diante, designar por conjunto de *máscaras* e que permita simplificar radicalmente a geração dos *n-gramas*. Cada *máscara* não é mais que uma sequência de dois símbolos que marcam as posições dum *n-grama* ocupadas por um *token* ou por um *gap*. O símbolo "1" foi adoptado como representando uma posição a ser ocupada por um *token* e o símbolo "0" como representante dos *gaps*.

1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	0	1	0	0	1
1	0	0	1	0	1	0
1	0	0	1	0	1	1
1	0	0	1	1	0	0
1	0	0	1	1	0	1
1	0	0	1	1	1	0
1	0	0	1	1	1	1
1	0	1	0	0	0	0
1	0	1	0	0	1	0
1	0	1	0	1	0	0
1	0	1	0	1	1	0
1	0	1	1	0	0	0

...

Figura 4.2 – Extracto da tabela de *máscaras*

Tomemos como exemplo o *corpus* [A B C D E F G H I J K L M], já antes utilizado, e *m* = 7. Se, para a primeira posição da janela de análise, aplicarmos a *máscara* [1 0 0 1 1 1 0], obtemos o *n-grama* [A _ _ D E F _]. A mesma *máscara* aplicada na terceira posição da janela de análise resulta no *n-grama* [C _ _ F G H _] (ver Figura 4.3).

Podemos então passar a representar cada *n-grama* ⁉ como um binómio, na forma {*pos*, *máscara*}, constituído por uma posição dentro do *corpus*, indicando o primeiro *token*, e por uma *máscara*.

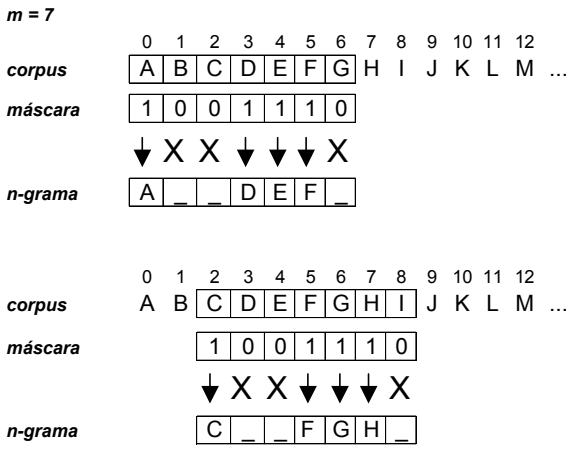


Figura 4.3 – Exemplo de aplicação duma *máscara*

Desta forma o *n-grama* [C _ _ F G H _], do exemplo anterior, poderá ser representado, dado o *corpus*, por {2, [1 0 0 1 1 1 0]}[36].

O conjunto de *máscaras* depende unicamente do valor *m* adoptado para a dimensão da janela de análise, sendo que as regras enunciadas relativamente à geração de *n-gramas* são totalmente transportas para o algoritmo de geração do conjunto de *máscaras*. Esta geração pode ser implementada por um algoritmo recursivo que toma como base o algoritmo de geração de todas as combinações possíveis de *gaps* e *tokens* dentro da dimensão duma janela de análise, condicionado às três restrições enunciadas nas secções 3.2.4 e 3.2.5 :

1. O primeiro *token* é fixo;
2. O número de *gaps* consecutivos deve ser menor ou igual a d_{max};
3. Deve ser sempre possível alinhar a *máscara* por forma a que na posição central da janela de análise fique um *token*.

[36] A primeira posição do *corpus* é a posição 0.

O código[37] do algoritmo de geração do conjunto de *máscaras* é apresentado na Figura 4.4, tomando cada *máscara* como um vector de *bytes*. O argumento *level* indica a posição em análise dentro da *máscara*. O argumento *depth* dá o número de *gaps* consecutivos inseridos desde a última posição reservada para um *token*. A primeira chamada a *GenMask* leva como argumentos uma *máscara* completamente preenchida com *gaps*, *level* a zero e *depth* a zero. Este último argumento obriga a que a primeira posição da *máscara* seja forçosamente reservada para um *token* e que em futuras chamadas recursivas esta condição se mantenha.

```
const int m = 7;

typedef unsigned char Byte;

const Byte GAP = 0;
const Byte FILLED = 1;

void GenMask(const Byte* mk, const int level, const int depth)
{
  if(level < m)
  {
    Byte mk1[m];
    Copy(mk1, mk);
    if(depth == 0)
    {
      mk1[level] = FILLED;
      if(ValidMask(mk1))                //testa alinhamento
        Store(mk1);
    }
    if(depth < ((m - 1)/2 -1))          // testa dmax
       GenMask(mk1,level+1,depth+1);    // Preenche com +1 GAP
    GenMask(mk1,level+1,0);             // Vai para a direita +1 FILLED
  }
}

void main()
{
  Byte ng[m];
  for(short int i=0; i<m; i++)
    ng[i] = GAP;
  GenMask(ng,0,0);                      //reserva primeira posição para
                                        //um token
}
```

Figura 4.4 – Algoritmo de geração do conjunto de *máscaras*

A função gera as *máscaras* numa estratégia de primeiro avançar em profundidade, colocando mais posições reservadas para *gaps*, recuando quando a condição de distância máxima entre *tokens* for violada.

[37] **Nota do autor:** O código apresentado ao longo deste texto corresponde a versões simplificadas em termos de instruções e de sintaxe relativamente ao código real implementado. Na sua maioria a sintaxe respeita as regras da linguagem C++ mas em alguns casos estas regras não são respeitadas totalmente por forma a tornar mais simples a apresentação. Igualmente são eliminados todos os testes não essenciais para a compreensão dos algoritmos ou aqueles que estejam ligados a particularidades da implementação concreta escolhida.

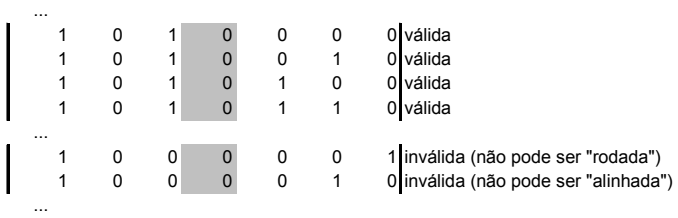


Figura 4.5 – Exemplos de *máscaras* válidas e não válidas

A função booleana *ValidMask*(*Byte * mk*) verifica se é possível alinhar a *máscara* por forma a que a posição central fique reservada para um *token*. Esta função, apresentada na Figura 4.6, verifica se uma determinada combinação é uma *máscara* válida, condicionando o seu armazenamento. Considera válidas todas as *máscaras* que tenham dimensão unitária (primeiro *if*), que tenham a posição central reservada para um *token* (segundo *if*) ou que possam ser "alinhados" por forma a terem a posição central reservada para um *token* (restante código). Todos as combinações que não tenham a posição central reservada para um *token* e que a última posição esteja reservada para um *token* (terceiro *if*) são rejeitadas, dado que não podem "deslizar" para a direita. Caso contrário o código verifica se a *máscara* pode ser "alinhada".

```
bool ValidMask(const Byte * value)
{
  bool result;

  short int meio = (m-1)/2;

  if(Lenght(value) == 1)result = true;
  else if(value[meio] != GAP)result = true;
  else if(value[m-1] != GAP) result = false;
  else
  {
    int i;
    int c2 = 0;
    for(i = m-1; i>meio;i--)
    {
      if(value[i] != GAP) break;
      c2++;
    }
    int c1 = 0;
    for(i = meio-1; i>=0;i--)
    {
      if(value[i] != GAP) break;
      c1++;
    }
    result = (c2>c1);
  }
  return result;
}
```

Figura 4.6 – Teste ao alinhamento duma *máscara*

No Anexo C é listado o conjunto das *máscaras* válidas para $m = 7$. É igualmente apresentado o conjunto de *máscaras* rejeitadas ou consideradas inválidas pelo algoritmo atrás mostrado. Do conjunto

de *máscaras* válidas faz parte a *máscara* [1][38] que, como referido anteriormente, é importante pelo facto de representar todas as sequências de dimensão unitária importantes para o cálculo da *ME* de *n-gramas* cujo número de *tokens* seja dois, de acordo com a equação **Erro! Estilo não definido.**.12.

Tal como é mostrado no Anexo C, a cada *máscara* é atribuído um identificador na forma dum número inteiro maior que um, recebendo a *máscara* [1] o identificador 0. Assim, retomando o último exemplo dado, o *n-grama* [C _ _ F G H _], poderá agora ser representado pelo binómio {2, 7} dado que se inicia na terceira posição do *corpus* e a *máscara* [1 0 0 1 1 1 0] tem o identificador 7.

Genericamente um *n-grama* passa a ser representado pelo binómio {posição no *corpus*, identificador da *máscara*} ou abreviadamente {*pos*, *mask*}. A posição no *corpus* é constituída, igualmente, por um binómio, o número do documento[39] e a posição dentro do documento, abreviadamente, {id_{doc}, pos_{doc}}. Juntando tudo obtemos a representação abreviada dum *n-grama*, dado um qualquer *corpus*, como dada por {{id_{doc}, pos_{doc}}, *mask*}.

Na representação completa, o *n-grama* [C _ _ F G H _], dado atrás como exemplo, fica representado por { {0,2}, 7 }, se considerarmos que a porção de *corpus* mostrada pertence ao primeiro documento do *corpus*. Este exemplo está mostrado de forma gráfica na Figura 4.7.

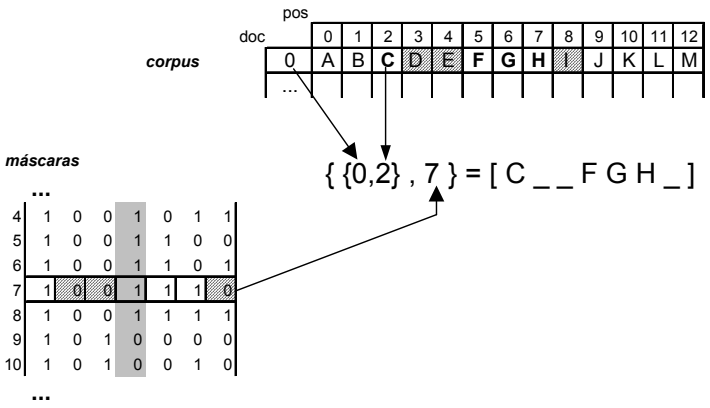


Figura 4.7 – Representação de *máscara* por um binómio {*pos*, *mask*}

A solução proposta passa então pela adopção duma estrutura de duplo apontador do tipo {{id_{doc}, pos_{doc}}, *mask*}. O primeiro indica a posição no *corpus*, na forma {id_{doc}, pos_{doc}}, que contém o primeiro *token* do *n-grama*. O segundo aponta para a *máscara* dentro dum vector contendo o conjunto de todas as *máscaras* válidas.

[38] Corresponde a [1 0 0 0 0 0 0]. Os *gaps* à direita ou à esquerda são usualmente omitidos.

[39] Pela mesma lógica o primeiro documento recebe o id 0.

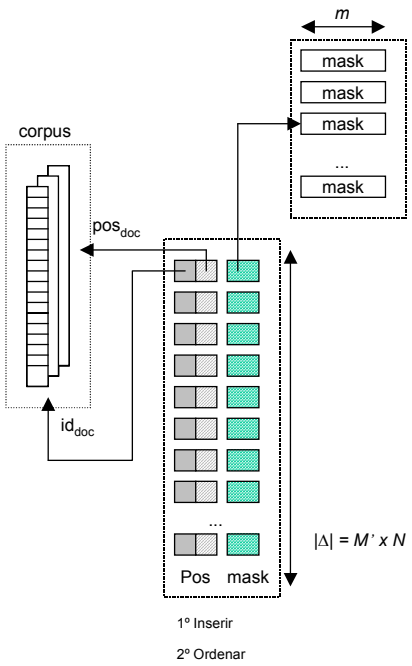


Figura 4.8 – Estrutura baseada em apontadores

Como resultado obtém-se uma estrutura extremamente mais leve que a primeira aproximação apresentada, contendo toda a informação existente na primeira e tendo como vantagem adicional o facto de ter memória da posição, dentro do *corpus*, associada a cada *n-grama*, essencial para o cálculo eficiente do conjunto $\oslash_{n+1}(w)$ dos *super-gramas* associados a cada *n-grama* w ? ☚$_{\mathrm{d}}$.

Se consideramos que um *token* necessita de 4 *bytes* para ser representado, então necessitamos de 4*m bytes* para cada *n-grama* na anterior representação. Para $m = 7$, corresponde a 28 *bytes*. Na nova representação necessitamos de 2 *bytes* para o identificador do documento, 4 *bytes* para a posição dentro do documento e 2 *bytes* para o identificador da *máscara*, num total de 8 *bytes*. Para $m = 7$ corresponde a uma redução de 70% no espaço ocupado. Com a vantagem adicional, muito importante, desta dimensão passar a ser sempre a mesma, ou seja, independente de *m*.

Apesar da forma indirecta como a informação passa a estar armazenada, os dois primeiros passos são precisamente os mesmos, inserir todos os *n-gramas* no vector ☚ de *n-gramas* e ordenar o vector de acordo com as regras de ordenação já enunciadas.

O primeiro passo fica extremamente simplificado porque é totalmente independente do conteúdo do *corpus*. Basta conhecer a dimensão do *corpus* e a dimensão do vector de *máscaras*. O vector é assim preenchido com dois ciclos encadeados. O primeiro percorre todas as posições do *corpus*. O segundo associa cada uma das *máscaras* a todas as posições do *corpus*. Fica assim gerado o conjunto dos binómios {*pos*, *mask*} representativo do conjunto ☚ de todos os *n-gramas* associados ao *corpus*.

O segundo passo, ordenação, fica um pouco menos eficiente dado que passa a ser necessário com base no binómio {*pos*, *mask*} obter-se o *n-grama* respectivo antes de efectuar-se qualquer operação. No entanto esta desreferenciação não apresenta dificuldades de maior, somente consome tempo. Para que este tempo seja o menor possível é importante que o *corpus* seja lido previamente para memória antes que seja efectuado qualquer tratamento posterior. Como veremos adiante esta condição é perfeitamente aceitável, pelo que a podemos tomar como adquirida, dado que o peso, em termos de espaço, do *corpus* em relação às restantes estruturas é muito pouco significativo.

Após o segundo passo, o vector fica ordenado por *n-grama*, tomando cada um deles como um cadeia de inteiros onde a posição de menor índice, representada normalmente à esquerda, tem maior significado e a posição mais à direita tem menor significado. A ordenação, como atrás foi dito, segue regras idênticas à ordenação lexicográfica de cadeias de caracteres.

Após a ordenação, as várias posições em que um dado *n-grama* se repete no *corpus* ficam armazenadas de forma consecutiva no vector. É possível então, novamente com uma só passagem, comprimir a informação e construir um segundo vector[40] com o conjunto $\blacktriangleleft_d$ de todos os diferentes *n-gramas*. É possível agora associar a cada um dos *n-gramas* o conjunto das posições onde cada um deles aparece repetido no *corpus*. Como vimos esta informação é basilar para a construção do conjunto $\oslash_{n+1}(w)$ para cada w ? $\blacktriangleleft_d$ necessário ao algoritmo *GenLocalMaxs*. O número de posições armazenadas corresponde à *frequência* associada a cada *n-grama*.

[40] Mais uma vez por simplificação de representação é apresentado um segundo vector. No entanto todo o algoritmo pode perfeitamente trabalhar sobre um mesmo vector.

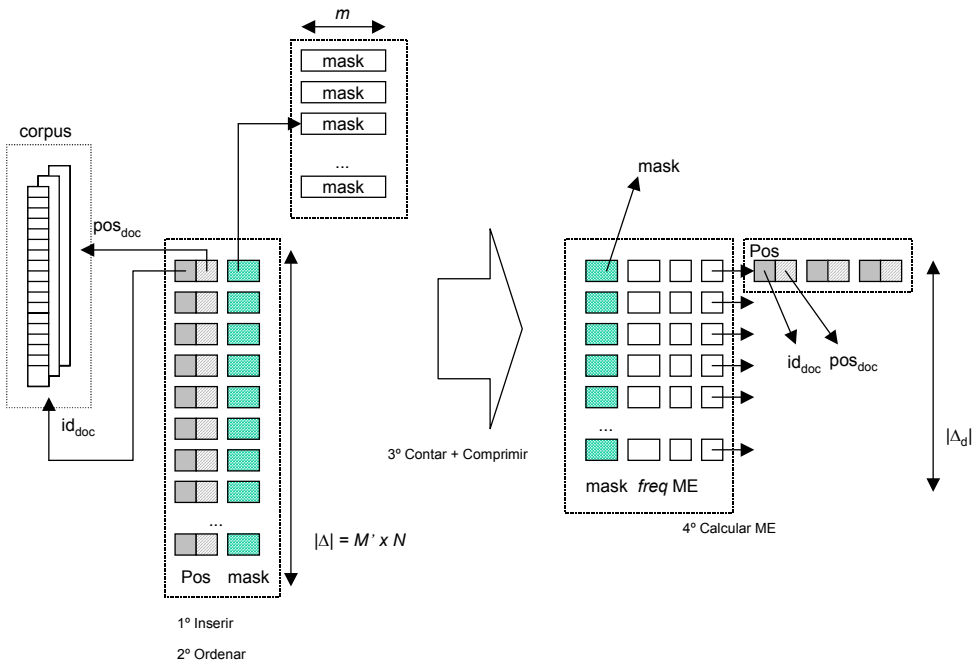


Figura 4.9 – Estrutura baseada em apontadores. Construção do conjunto Δ_d.

O quarto passo, cálculo da *ME* para cada *n-grama* $w \in \Delta_d$, continua a ser efectuado do mesmo modo e com o mesmo grau de complexidade. Depende essencialmente da complexidade associada ao algoritmo de pesquisa, dentro do conjunto Δ_d, dos *n-gramas* pertencentes ao conjunto $\Diamond_{n-1}(w)$, para obtenção do valor da respectiva *frequência*.

Com a nova estrutura é possível avançar-se para o quinto passo, ou seja, para a identificação eficiente do conjunto de *MWUs* existentes em Δ_d, objectivo primeiro do sistema, por aplicação do algoritmo *GenLocalMaxs*. Este passo passou a ser realizável dado ter passado a existir memória das posições onde cada *n-grama* $w \in \Delta_d$ ocorre no *corpus*, ou seja, é possível agora determinar-se eficientemente a vizinhança de cada *n-grama* $w \in \Delta_d$ e logo a lista de *tokens* válidos como substitutos de cada *gap*. A complexidade do algoritmo *GenLocalMaxs* fica dependente da complexidade do algoritmo de determinação da vizinhança de cada *n-grama* e do algoritmo de pesquisa de cada *n-grama* pertencente a $\Diamond_{n+1}(w)$ para obtenção da respectiva *ME*.

4.3 SOLUÇÃO "QUASE" IDEAL

Com a nova estrutura de dados houve um salto significativo em termos da redução do espaço necessário, ou seja, um aumento significativo da eficiência espacial. Este salto, quantitativo e qualitativo, foi possível mantendo níveis idênticos de eficiência temporal na maioria dos algoritmos. No caso da determinação do conjunto $\Diamond_{n+1}(w)$ existe uma melhoria da eficiência temporal.

Deixando o cálculo de complexidades para mais tarde podemos, no entanto, afirmar que a eficiência temporal do sistema depende essencialmente da complexidade associada aos seguintes algoritmos:

1. Algoritmo de ordenação do vector ☚, base do cálculo da *frequência*;
2. Algoritmo de pesquisa de *n-gramas*, sobre o conjunto $☚_{d,}$ para obtenção dos valores da respectiva *frequência* e *ME*;
3. Algoritmo de obtenção do conjunto $\oslash_{n-1}(w)$ para cada w ? $☚_{d;}$
4. Algoritmo de obtenção do conjunto $\oslash_{n+1}(w)$ para cada w ? $☚_{d;}$
5. Algoritmo de cálculo da *ME*;
6. Algoritmo *GenLocalMaxs*;

Esta lista é uma versão refinada e mais concreta da lista de sub-problemas enunciada no início deste capítulo vista agora à luz da estrutura de dados entretanto desenhada.

A complexidade do algoritmo de cálculo da *ME* depende totalmente do algoritmo de obtenção do conjunto $\oslash_{n-1}(w)$ e do algoritmo de pesquisa de *n-gramas* sobre o vector $☚_{d}$. A complexidade do algoritmo *GenLocalMaxs* está dependente dos algoritmos de obtenção dos conjuntos $\oslash_{n-1}(w)$ e $\oslash_{n+1}(w)$ e do algoritmo de pesquisa de *n-gramas* sobre o conjunto $☚_{d}$.

Podemos então afirmar que a eficiência temporal do sistema, no seu todo, depende essencialmente do algoritmo de ordenação do vector ☚, do algoritmo de pesquisa de *n-gramas* sobre o conjunto $☚_{d}$ e dos algoritmos de obtenção dos conjuntos $\oslash_{n-1}(w)$ e $\oslash_{n+1}(w)$. Existe um conjunto importante de melhorias, passíveis de incrementar a eficiência global do sistema, que endereçam cada um destes três problemas. A maioria destas optimizações centra-se na estrutura de suporte ao conjunto de *máscaras* utilizando-a como pólo concentrador da informação que permite a simplificação dos algoritmos e por consequência o aumento da sua eficiência.

4.3.1 AGRUPAR VERSUS ORDENAR

Ataquemos o primeiro problema, ou seja, a melhoria da eficiência do algoritmo de ordenação do vector ☚. Para tal necessitamos previamente de enunciar as condições para que dois *n-gramas* sejam considerados iguais. Estas condições são derivadas e aplicáveis directamente à nova representação adoptada, ou seja, ao binómio $\{pos, mask\}$.

Definição: Seja w e u dois *n-gramas* pertencentes ao conjunto ☚ de todos os *n-gramas* associados a um *corpus*. Tomemos w e u como sendo representados por binómios na forma $\{pos, mask\}$, onde *pos* indica a posição dentro do *corpus* do primeiro *token* do *n-grama* e *mask* é o identificador da *máscara* associada ao *n-grama* dentro do vector de *máscaras*. Nestas condições w e u são iguais se:

1º estiverem associados à mesma *máscara*, ou seja, $mask_w = mask_u$;

2º se o primeiro *token* de *w* for igual ao primeiro *token* de *u*, ou seja, *corpus*[pos_w]= *corpus*[pos_u], tomando o *corpus* como representado por um vector;

3º se os restantes *tokens* de *w* forem iguais aos restantes *tokens* de *u*.

A primeira condição já tinha sido utilizada na operação de compressão do vector ☚ para a criação do vector ☚$_d$ e a criação dos conjuntos de posições em que cada *n-grama* ocorre. A construção adoptada só é válida dado que cada conjunto de *n-gramas* iguais partilha a mesma *máscara*. Nesta construção cada diferente *n-grama* é representado por um binómio {*pos*, *mask*} em que *mask* tem sempre o mesmo valor e *pos* pode ser qualquer das diferentes posições em que o *n-grama* ocorre no *corpus*. Em qualquer dos casos está-se a representar o mesmo *n-grama*.

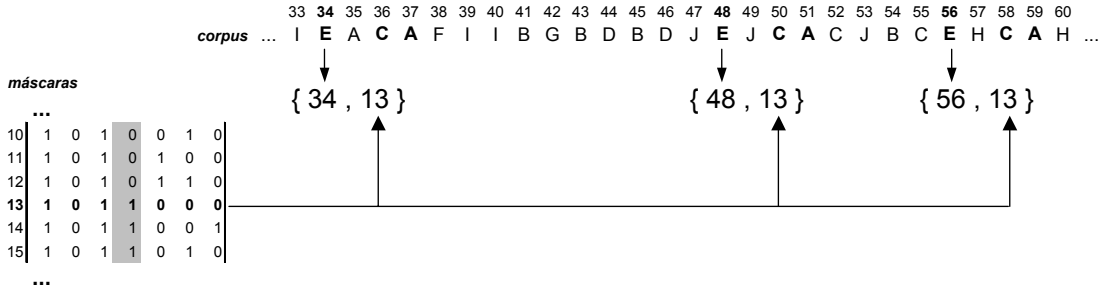


Figura 4.10 – Várias ocorrências dum mesmo *n-grama* (e.g. [E _ C A]) num *corpus*

No exemplo mostrado na Figura 3.1, o *n-grama* [E _ C A] ocorre três vezes no *corpus* podendo ser representado por quaisquer dos seguintes pares: {34, 13}, {48, 13} e {56, 13}. Notar que a *máscara* é sempre a mesma, variando unicamente a posição do primeiro *token*.

Conjugando a primeira e a segunda condições podemos eliminar a necessidade de ordenação total do vector ☚ de todos os *n-gramas*, substituindo-o por um máximo de *M'* % |☒| ordenações parciais. Esta simplificação obriga a uma ordenação prévia do *corpus* e é válida dado que o que procuramos não é ordenar o vector mas tão somente descobrir o número de ocorrências de cada diferente *n-grama*. Não serve de nada ordenar a totalidade do vector se sabemos à partida que dois *n*-gramas só serão iguais se tiverem a mesma *máscara* e o mesmo *token* inicial. Basta que o ciclo de geração do conjunto ☚ esteja construído de tal modo que estas duas condições estejam logo à partida cumpridas. O conjunto ☚ de *n-gramas* passa então a ser gerado parcialmente para cada uma das *máscaras* e para cada um dos *tokens* tomando o *corpus* ordenado[41]. Todos os conjuntos parciais de binómios {*pos*, *mask*}, que

[41] Não é obrigatório que o *corpus* seja ordenado. É uma condição suficiente mas não necessária para que seja válido o restante raciocínio. Na verdade, basta que seja encontrado um método que agrupe as posições em que cada um dos diferentes *tokens* ocorre. Não tendo sido encontrado outro método recorreu-se à ordenação.

partilhem a mesma *máscara* e cujas posições no *corpus* correspondam a um mesmo *token*, são então ordenados para que seja cumprida a terceira condição.

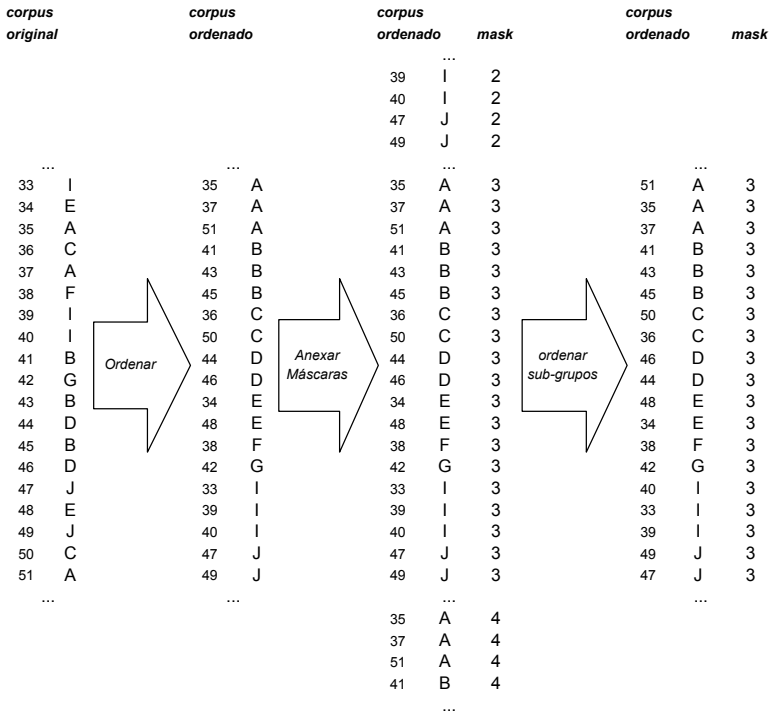


Figura 4.11 – Ciclo de "ordenação" do conjunto de *n-gramas*

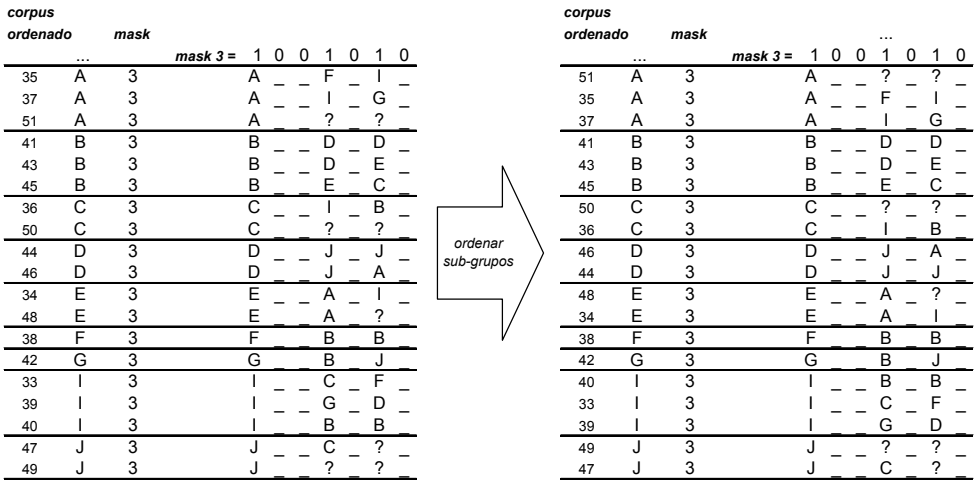


Figura 4.12 – Ordenações parciais

Na Figura 4.11 é mostrado esquematicamente o ciclo de "ordenação" do conjunto de *n-gramas* para obtenção da *frequência*. O último passo implica a desreferenciação de cada *n-grama* aplicando a *máscara* a cada uma das posições apontadas. A Figura 4.12 mostra, através dum exemplo, este passo. Nestas figura, os "?" correspondem às posições do *corpus* após o excerto mostrado como exemplo.

O vector resultante poderá não ficar completamente ordenado mas os *n-gramas* iguais estão agrupados, condição suficiente para a obtenção do conjunto Δ_d, para que possa ser determinada a *frequência* associada a cada *n-grama* e para que seja possível associar-lhe o conjunto de posições onde o mesmo ocorre no *corpus*.

```
class NGram {
    Corpus::Pos pos;
    short int pMask;
  ...
}

Corpus c;                                       // Corpus
NGramMasks masks;                               // Vector de máscaras

void GenΔd()
{
  ISVector<NGram> vectΔ(c.Dim());               // vector auxiliar com a dimensão do corpus

  Corpus::Pos p = c.FirstPos();                 // primeira posicao do corpus
  for(unsigned long j = 0; j < cDim; j++)       // preencher com todas as posicoes do corpus
  {
    vectΔ[j]  = NGram(p, 0);                    // mascara preenchida a zero
    p.Next();                                   // proxima posicao
  }
  vectΔ.Sort();                                 // ordenar posicoes corpus por token

  FillΔd(0, vectΔ);                             // preencher Δd para mascara zero

  Token actual, previous;
  unsigned long j, ini;

  for(short int z = 1; z < nMasks; z++)         // tratar cada mascara
  {
    ini = 0;                                    // inicio de cada ciclo
    previous = c[vectΔ[0].pos];
    for(j = 1; j < c.Filled(); j++)             // tratar todas as posicoes do corpus
    {                                           // com freq > 1
      vectΔ[j].pMask++;                         // actualiza valor da máscara
      actual = c[vectΔ[j].pos];
      if (actual != previous)                   // se mudou token ordenar
      {
        if((j - ini) > 1) vectΔ.Sort(ini, j-1); // ordenar se numero elementos maior que 1
        ini = j;
      }
      previous = actual;
    }
    if((j - ini) > 1) vectΔ.Sort(ini, j-1);     // ordenar ultimo grupo
    FillΔd(z, vectΔ);                           // preencher Δd presente mascara
  }
}
```

Figura 4.13 – Geração do conjunto Δ_d

De notar no algoritmo de geração do conjunto Δ_d, mostrado na Figura 4.13, o facto da dimensão do vector Δ estar, agora, limitada à dimensão do *corpus* e o facto de bastar uma ordenação inicial para garantir a geração ordenada dos diferentes *n-gramas* para as diferentes *máscaras*. A geração é efectuada *máscara* a *máscara* e dentro de cada *máscara*, *token* a *token*. A passagem duma *máscara* à seguinte faz-se pelo simples incremento do valor da *máscara* antes de cada posição do vector Δ ser tratada. A *máscara* 0, correspondente aos *n-gramas* de dimensão unitária, é tratada antes de entrar-se no ciclo de tratamento das restantes *máscaras*, dado que não necessita de nenhuma ordenação parcial posterior. Com estas simplificações foi possível reduzir drasticamente a dimensão do vector Δ e aumentar a eficiência temporal do algoritmo.

A classe *ISVector<T>* é um *template* C++ para vectores de apontadores para objectos que contenham sequências de *tokens*. É uma simplificação e adaptação, ao caso presente, da classe *VectorIndirectSortable<T>* apresentada em (Guerreiro 2000). A função de ordenação, como veremos na secção 4.5.1, foi melhorada para suportar ordenações eficientes de sequências de *tokens*. A utilização de vectores de apontadores em vez da utilização de vectores de objectos prende-se com a eficiência das operações de troca[42] de posições sempre presentes em qualquer algoritmo de ordenação.

A eficiência desta função está condicionada à eficiência do algoritmo que for utilizado para ordenação de cada um dos *M'* % |□| sub-conjuntos de *n-gramas*. Este algoritmo está no entanto extremamente simplificado pelo facto de estarmos a ordenar conjuntos de *n-gramas* que partilham uma mesma *máscara*, ou seja, que têm uma mesma dimensão e um conjunto de *tokens* e *gaps* precisamente nas mesmas posições. Existindo associado a cada *máscara* o conjunto de posições correspondentes a *tokens* basta então que o algoritmo de comparação percorra os dois *n-gramas*, da esquerda para a direita, e compare somente essas posições. A problemática da escolha do algoritmo de ordenação mais eficiente é o tema da secção 4.5.

A complexidade do algoritmo de geração do conjunto ☚$_d$ depende ainda da função *Fill*☚$_d$, dependendo esta da estrutura de dados adoptada para suportar o conjunto ☚$_d$. Mais adiante voltaremos a este tema apresentando a implementação desta função, completando então a função de geração do conjunto ☚$_d$.

4.3.2 VECTOR DE MÁSCARAS

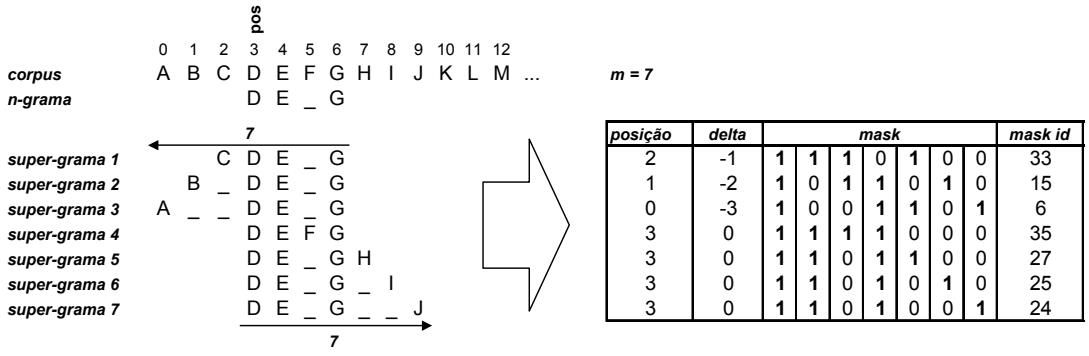
A solução para a melhoria da eficiência dos algoritmos de obtenção dos conjuntos $\oslash_{n-1}(w)$ e $\oslash_{n+1}(w)$ para cada w ? ☚$_d$, passa, novamente, pela constatação que existe um padrão associado à geração de cada um destes conjuntos. Este padrão depende somente da *máscara* que estiver associada a cada *n-grama*, não dependendo do conteúdo do *n-grama*, ou seja, dos *tokens*.

Tomemos um exemplo já anteriormente referido. O *n-grama* w = [D E _ G] tem associado o conjunto $\oslash_{n+1}(w)$ = {[C D E _ G]; [B _ D E _ G]; [A _ _ D E _ G]; [D E F G]; [D E _ G H]; [D E _ G _ I]; [D E _ G _ _ J]} de *super-gramas*, tomando m = 7 e onde A, B, C, F, H, I e J são *tokens* que ocorrem em simultâneo com o *n-grama* [D E _ G] nas várias ocorrências desse *n-grama* no *corpus*.

A Figura 4.14 mostra graficamente a geração do conjunto $\oslash_{n+1}(w$=[D E _ G]) para uma das posições em que w ocorre no *corpus*. Neste exemplo, o mais importante é a associação de cada *super-grama* a uma *máscara* e a um *delta*, ou seja, a um deslocamento em relação à posição base do *n-grama* em análise. É esta associação que nos permite generalizar o mecanismo de geração dos *super-gramas*,

[42] *Swap* em inglês

utilizado no exemplo, a todos os *n-gramas* que tenham a mesma *máscara* base associada. Por outras palavras, se alterarmos somente os *tokens* o conjunto resultante terá um padrão totalmente idêntico, ou seja, o padrão do conjunto $\oslash_{n+1}(w)$ depende somente do formato, isto é, da *máscara* associada a *w*. Assim, tomando *w* como o binómio {*pos*, [1 1 0 1]}, o conjunto dos seus *super-gramas* $\oslash_{n+1}(w)$ é { {*pos*-1, [1 1 1 0 1]}, {*pos*-2, [1 0 1 1 0 1]}, {*pos*-3, [1 0 0 1 1 0 1]}, {*pos*, [1 1 1 1]}, {*pos*, [1 1 0 1 1]}, {*pos*, [1 1 0 1 0 1]}, {*pos*, [1 1 0 1 0 0 1]} } ou de forma indirecta, recorrendo ao vector de *máscaras* mostrado no Anexo C, é igual a {{*pos*-1, 33}, {*pos*-2, 15}, {*pos*-3, 6}, {*pos*, 35}, {*pos*, 27}, {*pos*, 25}, {*pos*, 24}}.



posição	delta	mask							mask id
2	-1	**1**	**1**	**1**	0	**1**	0	0	33
1	-2	**1**	0	**1**	**1**	0	**1**	0	15
0	-3	**1**	0	0	**1**	**1**	0	**1**	6
3	0	**1**	**1**	**1**	**1**	0	0	0	35
3	0	**1**	**1**	0	**1**	**1**	0	0	27
3	0	**1**	**1**	0	**1**	0	**1**	0	25
3	0	**1**	**1**	0	**1**	0	0	**1**	24

Figura 4.14 – Exemplo de geração do conjunto $\oslash_{n+1}(w)$

Esta nova representação tem somente a posição inicial do *n-grama*, *pos*, como variável. O valor de *pos* deve ser substituído por todas as posições em que o *n-grama* ocorre no *corpus* para então obter-se o conjunto completo de todos os *super-gramas* de *w*. No exemplo anterior utilizou-se expressões do tipo "*pos-x* " para indicar que a posição inicial do novo *n-grama* é calculada subtraindo-se à posição base *pos* um *delta* dado por *x* considerando que as várias posições dum *corpus* são contíguas dentro de cada documento[43].

Existindo um padrão podemos então associar a cada *máscara* um conjunto de pares [*mask, delta*] com todas as configurações que permitam a geração do conjunto $\oslash_{n+1}(w)$ para qualquer *w* associado a essa *máscara*, simplificando e aumentando a eficiência do algoritmo de geração destes conjuntos, com um custo completamente insignificante em termos de espaço.

A lista de pares [*mask, delta*] associado a cada *máscara* é designado por conjunto de *super-masks*. A Figura 4.15 mostra dois exemplos de conjuntos de *super-masks*. Por exemplo, à *máscara* 23 ([1 1 0 1]), corresponde o conjunto de *super-masks* formado pelos pares {[33, -1], [15, -2], [6, -3], [35, 0],

[43] Todos os *super-gramas* assim gerados são válidos à luz das regras de validade atrás enunciadas. No entanto deverão ser eliminados todos os *super-gramas* que correspondam a posições negativas, obviamente inexistentes, ou cuja *máscara* ultrapasse o fim dum qualquer documento.

[27, 0], [25, 0], [24, 0]}. Desta forma, gerar o conjunto $\oslash_{n+1}(w)$ para qualquer w ? $\blacktriangleleft_d$ passa a ser uma tarefa repetitiva e extremamente simples. Basta percorrer o conjunto de *super-masks* associado à *máscara* de *w* e aplicar os *deltas* e as *máscaras* respectivas, repetindo a operação para todas as posições em que *w* ocorre no *corpus*. Estas posições são associadas a cada *n-grama* aquando do cálculo da respectiva *frequência*.

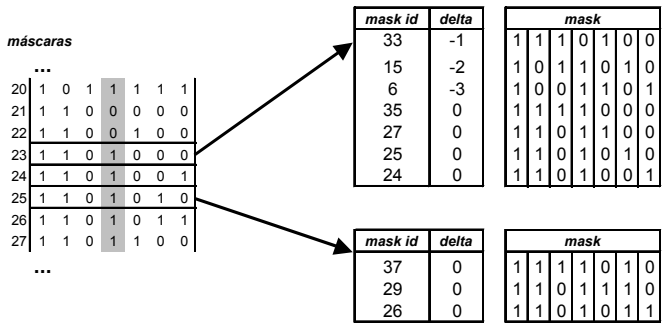


Figura 4.15 – Exemplos de conjuntos de *super-masks*

Raciocínio idêntico pode ser aplicado para a geração do conjunto $\oslash_{n-1}(w)$ de *sub-gramas* associados a qualquer *w* pertencente a $\blacktriangleleft_d$. Tomando, como exemplo, o *n-grama* [G _ _ I _ K L] e *m* = 7, o conjunto dos seus *sub-gramas* é {[I _ K L], [G _ _ I _ _ L], [G _ _ I _ K]}[44] ou na nova representação {{*pos*+2, [1 0 1 1]}, {*pos*, [1 0 0 1 0 0 1]}, {*pos*, [1 0 0 1 0 1]}}, o que é equivalente a {{*pos*+2, 13}, {*pos*, 2}, {*pos*, 3}} tendo em atenção a tabela de *máscaras* apresentada no Anexo C.

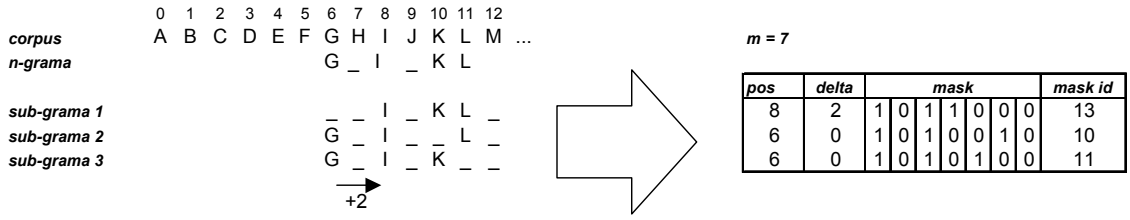


Figura 4.16 – Exemplo de geração do conjunto $\oslash_{n-1}(w)$

O conjunto de pares {*mask, delta*} associado a cada *máscara* é designado por conjunto de *sub-masks*. Por exemplo, a *máscara* 12 = [1 0 1 0 1 1 0] tem associado o conjunto de *sub-masks* {{13, +2}, {10, 0}, {11, 0}}, conforme exemplificado na Figura 4.16.

[44] Como vimos anteriormente, o *n-grama* [G _ _ _ K L] não faz parte do conjunto de *sub-gramas* porque viola a regra da distância máxima entre *tokens* num *n-grama*.

No Anexo D é apresentada a tabela completa dos atributos associados a cada *máscara* para o caso m = 7. Os atributos apresentados são a dimensão da *máscara*, o seu número de *tokens* e os conjuntos de *super-masks* e de *sub-masks* que lhe estão associados. Todos estes atributos são gerados automaticamente pelo sistema com base no valor de m e só dependendo deste valor. Outro factor importante associado à existência duma tabela de *máscaras* é o facto de somente a geração das *máscaras* depender directamente de m. Todos os outros algoritmos ficam independentes do valor de m, utilizando este valor, indirectamente, através da aplicação das *máscaras*.

4.3.3 MATRIZ INVERTIDA

Com a solução atrás apresentada, baseada em atributos associados à tabela de *máscaras*, passamos duma situação em que era necessário trabalhar directamente com as sequências de *tokens* e *gaps*, para uma situação em que passamos a manipular somente posições, *deltas* e identificadores de *máscaras*.

Após a ordenação inicial do *corpus* e o preenchimento do vector $\pointright_d$ com a *frequência* de cada *n-grama*, tarefas que dependem do conteúdo de cada posição do *corpus*, todo o trabalho posterior pode e deve ser efectuado, com evidentes vantagens, independentemente do conteúdo do *corpus*, ou seja, tendo somente em atenção os valores das posições, os *deltas*, as *máscaras* e as *frequências* entretanto armazenadas. Esta é a grande vantagem da solução proposta. No entanto, para que tal seja completamente verdade falta apresentar a estrutura que permite encontrar eficientemente cada *sub-grama* ou *super-grama* u de w, sem recorrer ao *corpus*, sabendo unicamente que u tem a forma dum binómio {*pos*, *mask*} calculado com base na posição base e nos atributos da *máscara* associada a w.

O cálculo da *frequência* de cada *n-grama* w ? $\pointright_d$ tem como objectivo suportar o cálculo da *ME* associada aos diferentes *n-gramas* e identificar as várias posições em que cada *n-grama* ocorre no *corpus*, necessárias ao *GenLocalMaxs*. Por outro lado, somente nos interessa procurar valores de *frequência* associados a *n-gramas* pertencentes ao conjunto $\oslash_{n-1}(w)$, ou valores de *ME* e posicionamento associados a elementos dos conjuntos $\oslash_{n-1}(w)$ e $\oslash_{n+1}(w)$. Sabemos já encontrar todos os elementos destes conjuntos, na forma dum binómio $\{pos=pos_{base}+delta, mask\}$ recorrendo aos atributos associados a cada *máscara*. Desta forma a estrutura lógica será aquela que indexe os *n-gramas* numa tabela de duas entradas indexada por posição e por *máscara*. Com esta matriz, daqui para a frente designada por *matrix*, é possível encontrar com complexidade constante, *O(1)*, cada *n-grama* sabendo o binómio {*pos*, *mask*} que lhe está associado. Encontrado o *n-grama* fica-se a saber a *frequência* e o conjunto de posições em que o mesmo ocorre no *corpus*.

Com um esquema de indexação, assim criado, é possível construir uma solução, muito eficiente, para a implementação do cálculo da *ME* e para o *GenLocalMaxs*.

A nova estrutura, mostrada na Figura 4.17, dá uma resposta completa ao conjunto de problemas enumerado no início desta secção, dado que permite a implementação eficiente dos algoritmos de

ordenação do vector ☚, de obtenção do conjunto $\oslash_{n-1}(w)$, de obtenção do conjunto $\oslash_{n+1}(w)$ e de pesquisa de *n-gramas* sobre o conjunto ☚$_d$.

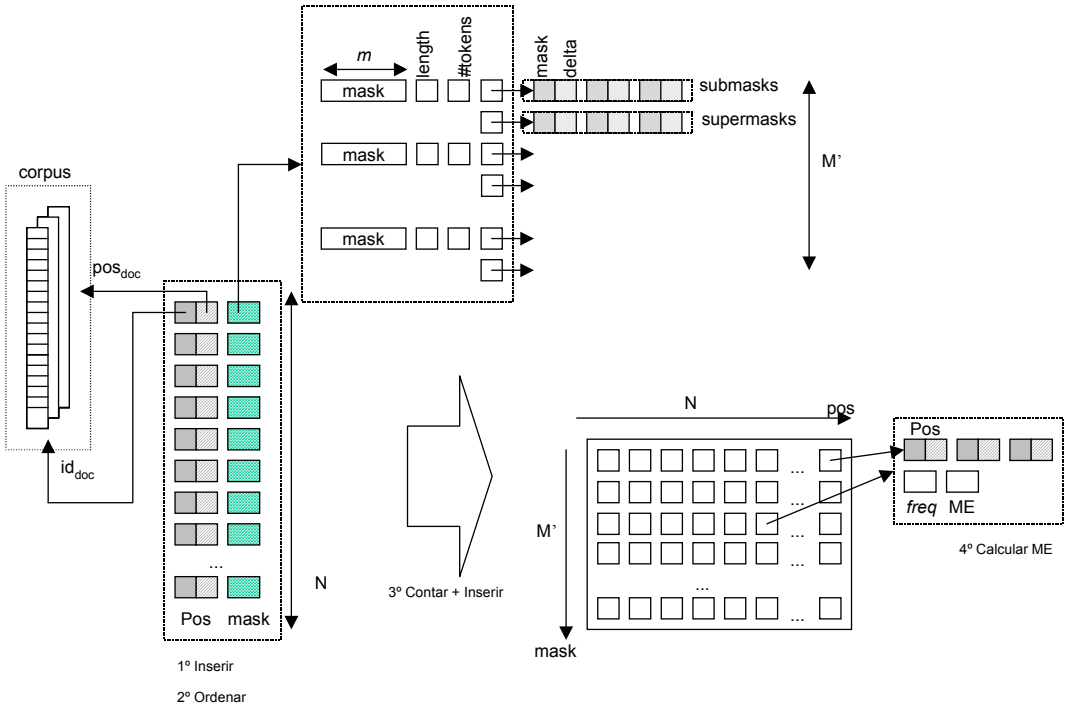


Figura 4.17 – *Matrix*

A *matrix* contém uma célula por cada *n-grama* pertencente ao conjunto ☚ de todos os *n-gramas* gerados a partir do *corpus*. Comporta-se, como antes afirmado, como um índice que permite obter directamente os valores associados a um determinado *n-grama* conhecendo a sua posição no *corpus* e o identificador da *máscara* que lhe esteja associada. As células da *matrix* são apontadores para estruturas, daqui para a frente designadas por *MNodes*[45], onde reside a informação relevante sobre cada diferente *n-grama* pertencente ao conjunto ☚$_d$, pelo que várias células poderão apontar para um mesmo *MNode*. Cada diferente *n-grama* ? ☚$_d$ terá então associado um *MNode* contendo a respectiva *frequência*, *ME*[46]e o conjunto das posições onde o mesmo *n-grama* se repete no *corpus*. Todas as células correspondentes a este conjunto de posições, para a *máscara* associada ao *n-grama*, apontam para um mesmo *MNode*.

[45] De *Matrix Node*

[46] Como será explicado mais adiante, na implementação real, optou-se por não armazenar o valor da *ME*.

O conjunto $\blacktriangleleft_d$ passa a estar representado, duma forma um pouco difusa[47], não por um vector mas pelo conjunto de *MNodes* associados à *matrix*. O conjunto $\blacktriangleleft_d$ pode ser obtido percorrendo o conjunto de células da *matrix* recolhendo os pares {*pos*, *mask*} correspondentes àquelas que apontem para um *MNode.* Ao longo do caminho é necessário ter o cuidado de marcar os *MNodes* já visitados para evitar repetições dum mesmo *MNode*, ou seja, dum mesmo *n-grama*.

A conjunção da *matrix* com o conjunto de *super-máscaras* associada a cada *máscara* permite resolver de forma muito eficiente o problema que oferecia maior dificuldade, ou seja, a obtenção do conjunto $\oslash_{n+1}(w)$ para cada $w \in \blacktriangleleft_d$. Este é fundamental para se encontrar uma implementação eficiente do *GenLocalMaxs*. Com a conjugação destas duas estruturas obtém-se igualmente de forma eficiente o valor da *ME* associada a cada um dos elementos do conjunto $\oslash_{n+1}(w)$ base ao algoritmo *GenLocalMaxs.*

A estrutura apresentada esquematicamente na Figura 4.17 é a base para a solução de todos os sub-problemas enunciados no início deste capítulo, ou seja, é a base para a implementação eficiente do algoritmo *GenLocalMaxs* conjugado com a implementação eficiente do cálculo da *ME*.

4.3.4 CÁLCULO EFICIENTE DA ME

É apresentado na Figura 4.18 o código respeitante ao algoritmo de cálculo da *ME* para um *n-grama* representado por um binómio {*pos*, *mask*} de acordo com a definição dada em **Erro! Estilo não definido.**.11.

[47] As razões para o classificador "difuso" são dadas na secção 4.3.6

```
class NGram {
    Corpus::Pos pos;
    short int pMask;
  ...
}

Corpus c;
NGramMasks masks;
Matrix matrix;

float CalcME(const Ngram& ng)
{
  float me = 0;
  unsigned long soma, freq;
  Byte nSub;
  Corpus::Pos pAux;

  Corpus::Pos pos = ng.pos;
  short int nMask = ng.pMask;

  freq = matrix[pos, nMask]->freq;        // Frequência do n-grama em análise

  if(masks[nMask].Lenght() > 1)           // Só valido para n-gramas com mais que um token
  {
    nSub = masks[nMask].NSub();           // cardinalidade do conjunto de subgramas
    soma = 0;
    for(Byte mk = 0; mk < nSub; mk++)     // calcular somatorio das frequencias dos subgramas
    {
      pAux = pos + masks[nMask].SubDelta(mk);               // aplicar delta
      soma += matrix[pAux, masks[nMask].SubMask(mk)]->freq;  // obter frequencia
    }
    me = (freq * freq) / ((c.Dim() / nSub) * soma);        // eliminados static_casts
                                                           // para melhor legibilidade
  }
  return me;
}
```

Figura 4.18 – Cálculo da *ME*

O código tem subjacente a existência duma estrutura tipo *matrix*, tal como atrás foi explicado, que possibilite que a *frequência* associada a cada *sub-grama* seja obtida em tempo constante e a existência dum vector de *máscaras* com um conjunto de *sub-masks* associado a cada *máscara,* que permita obter eficientemente o conjunto $\oslash_{n-1}(w)$.

A função de cálculo da *ME* pressupõe que os *n-gramas* de dimensão unitária (*mask* = 0) estão igualmente incluídos na *matrix* para efeitos de obtenção da sua *frequência*, necessária para o cálculo da *ME* dos *n-gramas* cujo número de *tokens* seja 2, de acordo com a equação **Erro! Estilo não definido.**.12. Esta generalização simplifica o código evitando a criação de situações especiais para tratamento destes *n-gramas*. Esta é a razão da existência do primeiro *if* onde é testada a dimensão do *n-grama* passado como argumento. Só *n-gramas* de dimensão não unitária têm *ME* associada.

O algoritmo é basicamente um ciclo sobre o conjunto de *sub-masks* associado à *máscara* do *n-grama* para o qual queremos conhecer a *ME*. Dado que o número de *sub-masks* é normalmente pequeno e o acesso à *frequência* de cada *sub-grama* é uma operação de complexidade constante, o cálculo da *ME* é extremamente eficiente.

4.3.5 IMPLEMENTAÇÃO EFICIENTE DO GENLOCALMAXS

À semelhança do código que implementa o cálculo da *ME*, o algoritmo *GenLocalMaxs* tem por base a *matrix* e o vector de *máscaras*. Tem como pressuposto que cada *MNode* tem o valor da *ME* previamente calculado através da função apresentada na Figura 4.18.

Em termos práticos o *GenLocalMaxs* são duas funções. A primeira, uma função booleana, recebe um *n-grama* e verifica se o mesmo é uma *MWU* de acordo com as condições enunciadas na definição **Erro! Estilo não definido.**.13. A segunda é um ciclo sobre o conjunto $\leadsto_d$ para identificação do conjunto de todas as *MWUs*.

A primeira função, designada por *IsLocalMax* é apresentada na Figura 4.19, testa, para um dado *n-grama*, o conjunto de condições impostas pelo algoritmo. Após obter o valor da *ME* do *n-grama* em avaliação, executa dois ciclos consecutivos. O primeiro para obter o valor máximo da *ME* dos seus *sub-gramas* e o segundo ciclo para obter o valor máximo da *ME* dos respectivos *super-gramas*. Ambos os ciclos utilizam a informação associada à *máscara* do *n-grama* em análise e acedem à *ME* em operações de complexidade constante baseadas na *matrix*. O segundo ciclo utiliza a informação armazenada relativa ao conjunto de posições em que cada *n-grama* se repete no *corpus*. Um *n-grama* é uma *MWU* se o valor da sua *ME* for maior ou igual que o valor máximo da *ME* dos seus *sub-gramas* e maior que o valor máximo da *ME* dos seus *super-gramas* de acordo com a definição **Erro! Estilo não definido.**.13.

A função, mostrada mais adiante na secção 4.4.2, que permite a identificação do conjunto de *MWUs* associado a um *corpus*, necessita percorrer toda a *matrix* verificando, para cada *n-grama*, se este é ou não uma *MWU*, utilizando para tal uma chamada à função *IsLocalMax*.

```
class NGram {
    Corpus::Pos pos;
    short int pMask;
  ...
}

Corpus c;
NGramMasks masks;
Matrix matrix;

bool IsLocalMax(const Ngram& ng) const
{
  Corpus::Pos pAux, pAux1;
  Byte mk;
  float maxSub, maxSuper;

  Corpus::Pos pos = ng.pos;
  short int nMask = ng.pMask;

  bool res = false;
  if(masks[nMask].NumOfTokens() >= 2)
  {
    float me = matrix[pos, nMask]->ME;                                  //obtém ME do n-grama

    maxSub = 0;
    if(masks.At(nMask)->NumOfTokens() > 2)
    {
      pAux = pos;
      for(mk = 0; mk < masks[nMask].NSub(); mk++)                       // percorre sub-masks
      {
        pAux = pos + masks[nMask].SubDelta(mk);                         // aplica delta
        maxSub = MAX(maxSub, matrix[pAux, masks[nMask].SubMask(mk)].ME);   // maximo ME sub-gramas
      }
    }

    maxSuper = 0;
    unsigned long freq = Matrix[pos, nMask].freq;
    for(unsigned long f = 0; f < freq; f++)                         // percorre ocorrencias do n-grama
    {
      pAux1 = Matrix[pos, nMask]->Pos(f);
      for(mk = 0; mk < masks[nMask].NSuper(); mk++)                 // percorre super-masks
      {
        pAux = pAux1 + masks[nMask].SuperDelta(mk);                             // aplica delta
        maxSuper = MAX(maxSuper, matrix[pAux, masks[nMask].SuperMask(mk)]->ME);  // maximo ME super-gramas
      }
    }
    res = (maxSub <= me) && (me > maxSuper);                        // verifica se é MWU
   }
   return res;
}
```

Figura 4.19 – Função *IsLocalMax*

4.3.6 ESTRUTURA IDEAL?

Expliquemos agora a razão do qualificador "quase" no título dado à secção 4.3. A *matrix*, na implementação apresentada, é uma estrutura que permite aceder ao valor da *frequência*, do posicionamento e da *ME* de cada *n-grama* numa operação de complexidade constante. Permite igualmente, em conjunção com o vector de *máscaras* e com o vector das posições em que cada *n-grama* ocorre no *corpus*, a identificação muito eficiente dos conjuntos $\oslash_{n-1}(w)$ e $\oslash_{n+1}(w)$ necessários ao cálculo da *ME* e ao *GenLocalMaxs*. Não podia ser melhor pelo que poderia ser considerada como ideal. No entanto, peca, razão porque não é ideal, em quatro aspectos:

1. Necessita de muito espaço;
2. O cálculo da *ME* tem que ser efectuado percorrendo toda a *matrix,* tendo o cuidado de não repetir o mesmo cálculo duas vezes;

3. A geração do conjunto de todas as *MWUs* necessita de percorrer toda a *matrix* e de marcar os *n-gramas* (os *MNodes*) já analisados para evitar repetições;
4. Será que existem características dos *n-gramas*, do *GenLocalMaxs* e/ou da *ME* não exploradas e que possam conduzir a maiores simplificações e a ganhos de eficiência?

Os segundos e terceiros pontos podem ser resolvidos pela inclusão de mais uma estrutura que guarde o conjunto $\blacktriangleright_d$ de todos os diferentes *n-gramas*. Esta estrutura pode ser formada simplesmente por um vector de binómios {*pos*, *mask*} e poderá ser preenchida em paralelo com a criação dos *MNodes*. Passa então a existir uma correspondência bi-unívoca entre cada *MNode* e uma entrada neste novo vector dado que existe um *MNode* para cada diferente *n-grama* existente no *corpus*. O valor de *pos* poderá corresponder a qualquer das posições em que o *n-grama* se repete. Com esta estrutura simplificam-se os ciclos para o cálculo da *ME* e para identificação das *MWUs* evitando a necessidade de marcar os *MNodes* para que não haja repetições. O ganho obtido versus o maior consumo de memória levou a que esta opção não fosse considerada em termos práticos. No entanto, existindo esta estrutura auxiliar, a eficiência temporal obtida seria dificilmente superada salvo no que diz respeito aos algoritmos de ordenação ou à exploração de características não identificadas até agora, áreas onde será sempre potencialmente possível encontrar novos caminhos mais eficientes.

O tema do excessivo espaço utilizado por esta implementação da *matrix* é o primeiro tema da próxima secção.

O quarto ponto é um tema, como veremos até ao final deste capítulo, difícil de encerrar, dado que existirão sempre possibilidades não analisadas e como tal não exploradas. Algumas das áreas estudadas são abordadas em seguida e outras são deixadas para trabalhos futuros.

4.4 REFINAMENTO DA SOLUÇÃO

Com já foi afirmado, apesar da solução apresentada ao longo deste capítulo demonstrar um grau de eficiência muito elevado do ponto de vista temporal, este é obtido sacrificando a eficiência espacial do conjunto. A *matrix* é uma solução muito eficiente para resolver o problema da pesquisa dos atributos associados a cada *n-grama* e por consequência para o cálculo da *ME* e para a implementação do *GenLocalMaxs*, mas consome uma grande quantidade de memória. Este desequilíbrio é tanto mais evidente se tomarmos em linha de conta que em *corpus* baseados em documentos escritos numa língua natural, a percentagem de *n-gramas* com *frequência* unitária poderá atingir percentagens superiores a 80% do total de *n-gramas* associados ao *corpus*. Este valor é confirmado pelas experiências efectuadas (ver capítulo 6). Esta observação, importante para a solução apresentada, aplica-se somente a *corpus* cuja fonte sejam documentos escritos numa língua natural. Nestes casos é normal que exista um importante número de *n-gramas*, gerados a partir do *corpus*, que só ocorram uma única vez, ou seja, cuja *frequência* seja unitária. No entanto, é extensível a todos os *corpus* em que não exista uma

tendência para uma qualquer distribuição comum ao conjunto de *tokens* ao longo dos vários documentos. Nestes casos é normal que exista um elevado número de *n-gramas* associados com uma única ocorrência.

Deste modo, uma percentagem muito elevada da *matrix* aponta para *MNodes* não partilhados por outras células, ou seja, corresponde a *n-gramas* que ocorrem uma única vez no *corpus*. Os *MNodes* não partilhados têm todos frequência unitária e uma lista de repetições reduzida a uma única posição, ou seja, têm pouca informação relevante associada.

É essencial procurar-se soluções de implementação alternativas que permitam reduzir o espaço de memória requerido.

4.4.1 REDUÇÃO DO ESPAÇO OCUPADO PELA MATRIX

A procura duma estrutura alternativa mais eficiente do ponto de vista de ocupação de memória mas que mantenha níveis próximos de eficiência temporal, para o armazenamento da informação relevante, ou seja, somente os *n-gramas* com frequência maior que 1, parece ser uma área com potenciais ganhos, principalmente ao nível da eficiência espacial. A redução do espaço ocupado em memória é fundamental para que possam ser trabalhados, de acordo com as premissas, *corpus* de dimensão elevada.

A opção tomada foi no sentido da criação de alternativas de implementação que permitam ter níveis de performance elevados para *corpus* de dimensão adequada à memória disponível ou, em opção, sacrificar de algum modo a eficiência temporal mas permitir o tratamento de *corpus* de maior dimensão. Assim o problema conduziu na prática a dois tipos de abordagens: 1º O manter da estrutura em matriz mas reduzindo a informação associada aos *n-gramas* cuja *frequência* seja unitária; 2º A passagem a uma estrutura mais leve do ponto de vista espacial mas com sacrifício da eficiência temporal. Estas duas alternativas foram implementadas e analisadas experimentalmente (ver capítulo 6) e são apresentadas em seguida.

4.4.1.1 Simplificações para n-gramas com frequência unitária

Mantendo a estrutura matricial já descrita foi experimentado o seguinte conjunto de alternativas práticas no sentido de reduzir o espaço ocupado:

1. Todos os *n-gramas* com *frequência* unitária passam a não ter associado um *MNode* individual dado que o seu valor é sempre o mesmo. As células da *matrix* passam a conter um valor especial indicativo que se está na presença dum *n-grama* com *frequência* unitária. Esta medida tem pouca implicação do ponto de vista da eficiência temporal mas reduz de forma importante o espaço ocupado. Na verdade, torna o algoritmo de pesquisa ligeiramente mais rápido dado que os *n-gramas* de *frequência* unitária ficam de imediato identificados, mas obriga ao teste

de mais condições. Uma acção contraria a outra pelo que em termos globais a eficiência temporal fica na mesma;

2. Não guardar o valor da *ME* associada a cada *n-grama*. É uma medida que leva a uma importante redução do espaço necessário dado que estamos a falar dum valor do tipo *float*. No entanto, o impacto em termos da eficiência temporal é visível, apesar do cálculo da *ME* ter uma implementação extremamente eficiente quando suportada pela estrutura matricial atrás apresentada e tendo os valores de *frequência* previamente calculados e armazenados. A degradação deve-se essencialmente ao facto de cada *n-grama* participar no cálculo da *ME* e nas chamadas ao *GenLocalMaxs* de vários outros *n-gramas*. Deste modo, não estando armazenado, se repete várias vezes o mesmo cálculo;

3. Diferenciar a informação armazenada para a *máscara* 0 (i.e. [1]) dado que, de acordo com a definição e propriedades da *ME* e do *GenLocalMaxs*, para esta *máscara* só necessitamos conhecer a *frequência* de cada *n-grama*. Evita-se assim o armazenamento do conjunto de posições em que cada *n-grama*, baseado na *máscara* 0, ocorre no *corpus*;

4. Para além de reduções de espaço ao nível da *matrix* tentou-se reduzir a dimensão do vector < pela eliminação, à partida, de todos os *tokens* com *frequência* unitária da lista de potenciais primeiras posições de *n-grama*. Esta eliminação é válida dado que todos os *n-gramas*, de acordo com a observação **Erro! Estilo não definido.**.15, assim gerados, terão *frequência* unitária e não serão inseridos na *matrix*.

   O mesmo poderia ser generalizado para qualquer posição dos *n-gramas* gerados mas tal simplificação é de difícil aplicação prática contrariamente à respeitante à primeira posição. Esta simplificação foi efectuada sem um visível impacto do ponto de vista temporal dado que a percentagem de *tokens* com frequência unitária, como provado experimentalmente, tende para zero com o aumento da dimensão do *corpus*, quando o contexto dos vários documentos é o mesmo. Assim, sendo, só é importante para *corpus* de pequena dimensão.

Em termos de protótipo foi experimentada a implementação de todas estas alternativas tendo-se decidido pela sua utilização simultânea como meio de redução de espaço.

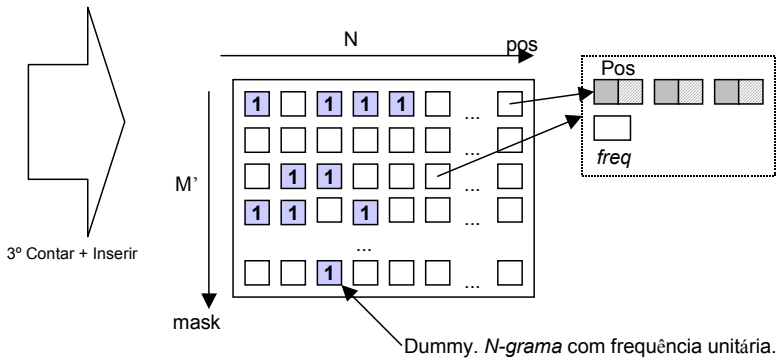


Figura 4.20 – *Matrix* com *n-gramas* unitários marcados de forma especial

Somente agora, após estarem enunciadas todas as restrições implementadas é possível apresentar o código relativo à função *Fill*☚$_d$ que faltava para completar o algoritmo mostrado na Figura 4.13. É, resumidamente, um ciclo que toma um vector de *n-gramas* ordenados e associados a uma mesma *máscara* e que conta as ocorrências de cada *n-grama* registando a informação necessária na *matrix*. Só são criados *MNodes* e são preenchidas células para *n-gramas* com *frequência* não unitária. As restantes células, correspondentes a *n-gramas* que ocorrem uma única vez no *corpus*, são marcadas, previamente, com um apontador para um *MNode* especial, designado por *dummy* no código.

```
...
class MNode {
  Corpus::Pos* pos;     // Vector de posicoes com igual n-grama
  long freq;            // frequencia
  ...
}

MNode * dummy;          // Identificador de n-grama com frequência unitária

void FillΔd(const short int mask, const Vector<NGram>& vectΔ)
{
  // Matrix previamente preenchida na totalidade com dummy
  unsigned long numberOf = vectΔ.Filled();
  unsigned long r;                                  // Numero de repeticoes em cada loop
  unsigned long j;
  MNode * pNode;                                    // Pointer para MNode

  r = 1;                                            // existe pelo menos um elemento
  for(unsigned long i = 1; i <= numberOf; i++)      // percorre vector ordenado
    if(i < numberOf && (vectΔ[i] == vectΔ[i-1]))    // enquanto mesmo elemento
      r++;                                          // conta numero de ocorrencias
    else                                            // fim dum ciclo
    {
      if(r > 1)                                     // se frequencia maior que 1
      {
        pNode = new MNode(r);                       // cria novo MNode com freq = r
        for(j = i-r; j < i; j++)                    // percorre r posicoes
        {
          matrix[vectΔ[j].pos, mask] = pNode;       // coloca toda as posicoes a apontar
                                                    // para a nova estrutura
          Node->Add(vectΔ[j].pos);                  // adiciona posicao à estrutura
        }
      }
      r = 1;                                        // reinicializa r para novo ciclo
    }
}
```

Figura 4.21 – Algoritmo de preenchimento da *matrix*

A função apresentada na Figura 4.21 implementa o preenchimento da *matrix,* a criação dos vários *MNodes* e o seu preenchimento em termos de *frequência* e do vector de posições em que cada *n-grama* se repete no *corpus*. É, no essencial, um ciclo que percorre o vector ☚ para uma determinada *máscara*. Vector e *máscara* são passados como argumento à função *Fill*☚$_d$. O ciclo tem dois comportamentos distintos mas complementares. Numa primeira fase (ramo verdadeiro do *if*) faz a contagem do número de ocorrências de cada *n-grama*. Numa segunda fase, quando a sequência passa para outro *n-grama*, cria um *MNode* e regista as informações respeitantes à *frequência* e ao conjunto de posições em que o *n-grama* se repete no *corpus*.

Como atrás foi dito está, indirectamente, a criar o conjunto ☚$_d$ dos diferentes *n-gramas* associados ao *corpus*, concretizado no conjunto de *MNodes* criados. É criado um novo *MNode* por cada *n-grama*.

4.4.1.2 Matrix compacta

A *matrix* é, como sabemos, uma estrutura a duas dimensões, posição no *corpus* (*pos*) e identificador de *máscara* (*mask*). A segunda dimensão (i.e. identificador da *máscara*) é muito inferior à primeira. Por outro lado cada *pos* é constituída pelo identificador dum documento (id_{doc}) e pela posição (pos_{doc}) dentro desse documento. Tendo este factos em mente é possível construir-se uma estrutura alternativa

à *matrix*, com uma mesma interface, com alguma perca em termos de eficiência temporal associada às operações de inserção e de busca mas com ganhos significativos em termos da eficiência espacial.

Uma potencial solução passa pela utilização duma matriz, indexada pelo binómio $\{id_{doc}$, *mask*$\}$ de apontadores para árvores indexadas por pos_{doc} cujos nós contêm apontadores para *MNodes*, somente respeitantes a *n-gramas* com *frequência* não unitária. Todos os *n-gramas* não representados têm, por omissão, *frequência* unitária. Na Figura 4.22 está esquematicamente representada esta estrutura.

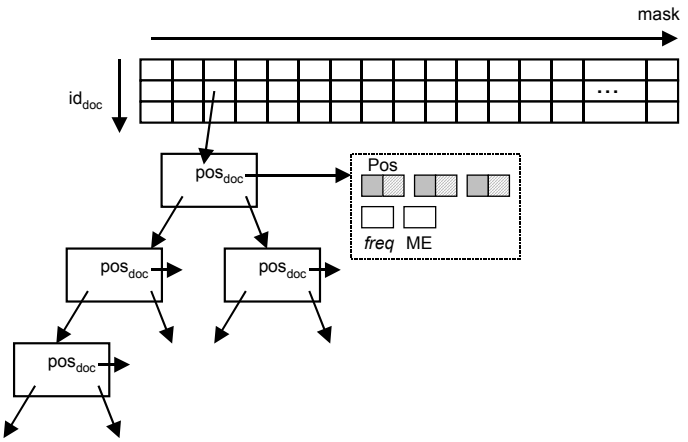


Figura 4.22 – Primeira aproximação a uma estrutura alternativa para implementação da *matrix*

Com esta estrutura reduz-se drasticamente o número de *n-gramas* representados mas aumenta-se o espaço necessário para representar cada *n-grama*. Em contas muitos simples, passa-se de um *pointer* para quatro. Passando a representar somente 10 a 20% dos *n-gramas* é possível obter-se reduções de espaço na ordem dos 50%. Este valor pode variar tendo em conta o tipo de árvore escolhida que depende fortemente do *corpus* e da necessidade de manter níveis elevados de eficiência nas operações de inserção e de busca. Existem várias alternativas de implementação, nomeadamente árvores binárias não-balanceadas, árvores binárias balanceadas, *digital search trees* (DST) ou *PAT tries* (Sedgewick 1998), mas todas elas apresentam a mesma desvantagem ao nível do espaço necessário para o seu armazenamento.

Uma segunda alternativa, em tudo idêntica à anterior do ponto de vista de funcionalidades, é a formada pelo mesmo vector duplo indexado por $\{id_{doc}$, *mask*$\}$ mas com as árvores substituídas por vectores ordenados de posições dentro de cada documento.

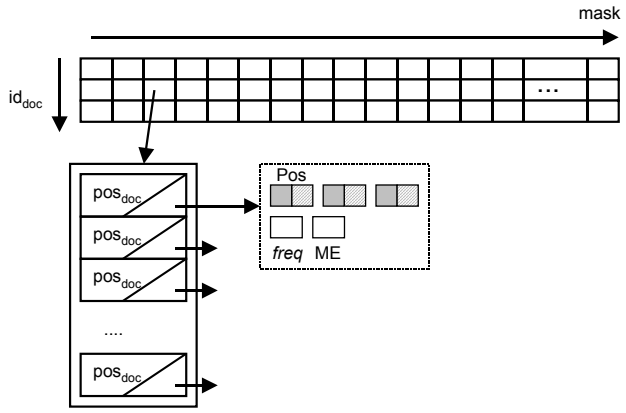


Figura 4.23 – Estrutura alternativa, compacta, para implementação da *matrix*

Com esta estrutura é possível atingir-se reduções superiores a 50% em espaço ocupado[48] em relação à primeira implementação da *matrix* apresentada, derivada da não representação dos *n-gramas* com frequência unitária. Em contrapartida existe um agravamento ao nível da eficiência das operações de inserção e busca dado que passamos de operações de complexidade constante para operações de complexidade tipicamente logarítmica. Na prática, verificou-se que, através duma codificação cuidada, é possível ultrapassar algumas destas limitações. A grande vantagem desta estrutura é que todo o espaço ocupado corresponde a informação útil para o sistema.

A pesquisa dum *n-grama* na *matrix* passa duma operação de complexidade $O(1)$ para uma operação de complexidade:

$$O\left(\frac{N}{nd}\times \%nu\right) \qquad \text{(Erro! Estilo não definido..2)}$$

onde *%nu* representa a percentagem de *n-gramas* com *frequência* não unitária.

A pesquisa passa a ser efectuada em dois passos: 1º Identificação do vector de *posições* correspondentes à *máscara* e ao *documento* em que o *n-grama* está inserido; 2º Pesquisa binária sobre o vector tendo como argumento a posição do *n-grama* dentro do documento. O logaritmo deriva desta segunda operação. O argumento do logaritmo deriva de considerarmos que o *corpus* é dividido irmãmente por todos os documentos, ou seja, que o número médio de *tokens* por documento é dada pela divisão do número total de *tokens* pelo número de documentos. Estamos a admitir que o número de *n-gramas* com *frequência* não unitária segue igual distribuição.

[48] Comprovadas pelas experiências efectuadas. Ver capítulo 6.

Sabendo que a percentagem de *n-gramas* de *frequência* não unitária presentes no *corpus* é baixa, consequentemente a eficiência temporal de cada pesquisa, apesar de logarítmica, resulta pouco agravada em relação à situação ideal de mantermos a implementação matricial da *matrix*.

Como efeito lateral temos igualmente a vantagem do espaço de pesquisa de *MWUs* ter sido reduzido. O percurso efectuado sobre a *matrix* passa agora a estar limitado aos *n-gramas* com frequência não unitária, ou seja, aqueles que de acordo com a propriedade 2, enunciada na secção 3.9, poderão ser eleitos como *MWU*. Não é o ideal mas é muito inferior ao anterior.

Nesta versão da estrutura de dados, por uma questão de eficiência, optou-se por calcular previamente e armazenar o valor da *ME* associada a todos os *n-gramas* com *frequência* não unitária. Verificou-se experimentalmente que, devido à complexidade acrescida do algoritmo de procura dum *n-grama* na *matrix*, este era a operação mais penalizada. Esta opção está associada à constatação, derivada directamente da observação **Erro! Estilo não definido.**.16, que no cálculo da *ME* de *n-gramas*, não unitários, aqueles que podem ser *MWUs*, só participam *n-gramas* com *frequência* igual ou superior à do *n-grama* base. O cálculo da *ME* associada aos *n-gramas* com frequência unitária, necessários durante o segundo ciclo do algoritmo *GenLocalMaxs*, continua a ser efectuado no momento em que é necessário.

Ao nível experimental, como mostrado no capítulo 6 dedicado à apresentação dos resultados, foi igualmente construída uma versão do sistema baseada nesta última alternativa de implementação da *matrix* por forma a que fosse possível comparar as duas opções atrás enunciadas.

4.4.2 EXTRACÇÃO DO CONJUNTO DE MWUS

Como já foi referido, a propriedade 2 aponta para uma simplificação importante ao nível do ciclo que é percorrido para determinar quais dos *n-gramas* pertencentes a ☚_d podem ser potenciais *MWUs*. Esta propriedade diz-nos que somente os *n-gramas* com *frequência* superior à unitária poderão vir a ser eleitos como *MWU*. Utilizando a implementação original da *matrix* mostrada na Figura 4.17, basta que no ciclo de geração das *MWUs* seja acrescentada uma condição de teste ao valor da *frequência*, eliminando os *n-gramas* que ocorram um única vez em ☚_d. Caso sejam utilizadas as medidas de redução de espaço apresentadas na secção 4.4.1.1, a identificação dos *n-gramas* com *frequência* unitária passa a ser directa. Utilizando a estrutura compacta alternativa, os *n-gramas* com *frequência* unitária estão eliminados à partida pelo que basta que o ciclo de identificação de *MWUs* somente tenha em conta os *n-gramas* armazenados. Esta é uma vantagem importante desta segunda opção de implementação. A aplicação desta propriedade tem um impacto significativo ao nível da eficiência temporal do sistema dado que uma parte muito significativa do conjunto de *n-gramas* passa a ser excluído logo à partida.

Foi adicionalmente acrescentada, no mesmo ciclo, uma condição de rejeição de todos os *n-gramas* cujo número de *tokens* coincida com a dimensão *m* da janela de análise. Estes *n-gramas*, contíguos, não são interessantes do ponto de vista prático. Têm grande probabilidade de serem eleitos como *MWUs*, pelo método exposto, dado não terem *super-gramas* associados. Eliminando-os melhora-se a *precisão* do sistema.

Igualmente, dado que, de acordo com a definição **Erro! Estilo não definido.**.13, só *n-gramas* com pelo menos dois *tokens* são potenciais *MWUs*, não são incluídos no ciclo todos os *n-gramas* formados com base na *máscara* 0 correspondente aos *n-gramas* unitários.

É agora possível apresentar o código para a função que gera todas as *MWUs.* É um ciclo que percorre todas as células da *matrix* à procura de *n-gramas* não unitários e que respeitem as condições para serem uma *MWU*. Durante o percurso há o cuidado de marcar os *n-gramas* já analisados por forma a não gerar repetições da mesma *MWU*.

```
class NGram {
    Corpus::Pos pos;
    short int pMask;
  ...
}

Corpus c;
NGramMasks masks;
Matrix matrix;
Byte m;

void WriteMWUs() const
{
  float me;
  Byte m;
  long freq;

  unsigned long i;
  Corpus::Pos p;

  matrix.ResetAll();

  for(short int y = 1; y < masks.TotalMasks(); y++)
    if(mask[y].NumOfTokens() < m)
      for(Corpus::Pos x = c.FirstPos(); x <= c.LastPos(); x.Next())
        if(matrix[x,y] != dummy && !matrix[x,y].IsSet())
        {
          if(IsLocalMax(NGram(x, y))
            Write(NGRam(x, y));
          matrix[x,y].Set();
        }
}
```

Figura 4.24 – Geração do conjunto das *MWUs*

Com a informação existente, é possível adicionar a cada *MWU,* informação sobre o valor da *ME* que lhe está associada e o conjunto de posições em que a mesma ocorre no *corpus*. Este tipo de informação pode ser importante para a posterior análise ou utilização dos resultados obtidos.

Caso fosse tomada a opção pela criação duma estrutura de suporte ao conjunto $\blacktriangleleft_d$ dos diferentes *n-gramas*, esta poderia conter somente os *n-gramas* com frequência não unitária e cujo número de *tokens* fosse superior a 1 e inferior a *m*. Desta forma obtinha-se uma importante simplificação do ciclo

de identificação das *MWUs*, obtendo-se ganhos visíveis do ponto de vista temporal. Como já foi afirmado esta opção não foi implementada.

4.5 ALGORITMOS DE ORDENAÇÃO

Existem três momentos em que a eficiência do sistema fica dependente da eficiência dos algoritmos de ordenação escolhidos:

1. A ordenação inicial do *corpus* por *token*;
2. As ordenações parciais do vector ☚ para o cálculo da *frequência*;
3. A ordenação de cada vector de posições no caso de se optar pela implementação compacta da *matrix* mostrada na Figura 4.23.

Os casos 1 e 3 correspondem a ordenações de vectores de inteiros, singulares, enquanto que o caso 2 implica a ordenação de cadeias de inteiros o que é em tudo idêntico à ordenação de *strings* de caracteres. Comecemos por este.

4.5.1 CASO 2 - ORDENAÇÃO DOS N-GRAMAS

Tendo todo o trabalho de pesquisa tido origem no artigo (Yamamoto *et al.* 2000), o natural foi tentar estender as conclusões destes autores ao caso presente. Estes autores defendem a utilização do algoritmo de ordenação de *sufixos* proposto por Manber e Myers, anos antes, no artigo (Manber *et al.* 1991), utilizando uma versão posteriormente refinada por Doug McIlroy (McIlroy *et al.* 1997). Jesper Larsson e Kunihiko Sadakane apresentam em (Sadakane *et al.* 1999) uma versão melhorada, mais eficiente, do mesmo algoritmo para ordenar *sufixos*. No entanto, as técnicas utilizadas são exclusivamente aplicáveis a ordenação de *sufixos* dado que se fundamentam num conjunto de propriedades particulares e exclusivas dos próprios *sufixos*. Apesar das técnicas não serem aplicáveis, os algoritmos base utilizados são totalmente vocacionados para a ordenação de *strings* independentes, ou seja, são transferíveis para a ordenação de *n-gramas*. Para a ordenação é utilizado um algoritmo semelhante ao *Multikey Quicksort*[49] proposto, nomeadamente, por Bentley e Sedgewick em (Bentley *et al.* 1997). Foi este último algoritmo o escolhido, neste trabalho, para a ordenação de *n-gramas.*

Algoritmos do tipo *Radix Sort* são os candidatos mais naturais para a ordenação lexicográfica de *strings*. São algoritmos que ordenam progressivamente o vector de *strings*, operando em cada ciclo somente sobre uma única posição da *string*, à semelhança do que fazemos quando pesquisamos entradas num *dicionário*. Começamos pela primeira letra e depois progressivamente vamos utilizando

[49] Também designado por *Ternary String Sorting* e por *Three-Way Radix Quicksort*

as restantes letras para chegarmos onde pretendemos. O *MSD*[50] *Radix Sort* e seus derivados são uma das alternativas existentes para algoritmos tipo *Radix Sort*. Neste caso as sequências são ordenadas da esquerda para a direita, ou seja, da posição mais significativa para a posição menos significativa (Sedgewick 1998), (McIlroy *et al.* 1993), (Anderson *et al.* 1994), (Dale 1999).

Sendo os *n-gramas* não mais que um caso especial de *strings*, se considerarmos somente as posições ocupadas por *tokens* e a comparação de *n-gramas* associados a uma mesma *máscara*, então algoritmos do tipo *Radix Sort* são indicados para a ordenação de vectores de *n-gramas*.

O *Quicksort*, originalmente proposto por Hoare em 1962, é talvez o algoritmo mais conhecido, estudado e utilizado na prática para ordenações de vectores. Tal deve-se em grande medida ao facto da sua complexidade média, para vectores de elementos distribuídos aleatoriamente, ser das melhores comparativamente com outras alternativas. Somente aplicações particulares, estudadas para distribuições de elementos bem conhecidas, conseguem na prática suplantar o *Quicksort*. Apesar dos largos anos de investigação em torno de outros algoritmos o *Quicksort* e seus derivados continuam a ser os escolhidos em muitas das implementações práticas (Jaja 2000). O *Quicksort* é basicamente um algoritmo baseado no método largamente conhecido de dividir para conquistar (Bentley *et al.* 1997), (Dale 1999). Duma forma simples o algoritmo processa recursivamente o vector, dividindo-o em duas metades, uma com os elementos maiores que um pivô, escolhido em cada etapa, e a outra com os valores menores que esse mesmo pivô. Nada é dito sobre onde colocar os elementos iguais ao pivô ficando dependente da implementação se ficam no conjunto dos maiores ou dos menores que o pivô (Bentley *et al.* 1997). Em 1993, Bentley e McIlroy no artigo "*Enginnering a sort function"* [51] propõem uma nova abordagem para o método de partição, baseando-o na existência de três partições: os maiores que o pivô, os menores que o pivô e os iguais ao pivô. A forma eficiente como manipulam esta terceira partição, já anteriormente estudada por outros autores, é a inovação que introduzem em relação ao algoritmo original (ver Figura 4.25). O algoritmo resultante, designado por *Ternary-Split Quicksort*, é particularmente vocacionado para a ordenação de conjuntos de dimensão muito elevada e onde haja muitos elementos repetidos.

---

[50] *MSD – Most Significant Digit*

[51] Não houve acesso directo a este artigo mas somente através de outros artigos em que o primeiro é referenciado e largamente estudado.

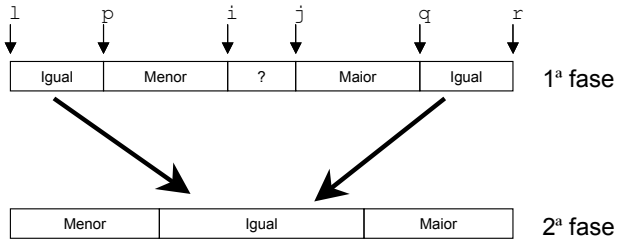


Figura 4.25 – *QuickSort* com três partições

O *Multikey Quicksort* é um misto entre *Ternary-Split Quicksort* e *MSD Radix Sort*. Tal como o *Ternary-Split Quicksort* divide, em cada etapa, o vector em três partições e tal como o *MSD Radix Sort* analisa cada *string* da esquerda para a direita, analisando uma só posição em cada etapa e só avançando para a seguinte na partição dos elementos iguais ao pivô (Sedgewick 1998), (Larsson *et al.* 1999), (Bentley *et al*. 1997).

```
template <class T>
class ISVector {
  private:
    typedef T * TPointer;
  private:
    TPointer *items;
  ...
  private:
    void MKQuickSort(unsigned long l, unsigned long r, unsigned short depth, unsigned short maxDepth);
    unsigned long ChoosePivot(unsigned long l, unsigned long r, unsigned short depth) const;
  ...
}
...
template<class T>
void ISVector<T>::MKQuickSort(unsigned long l, unsigned long r, unsigned short depth, unsigned short maxDepth)
{
  if((r-l)+1 == 2)                                // se dois elementos ordena directamente
  {
    T::SetFirst(depth);                           // só precisa ordenar a partir da posicao depth
    if(*items[l] > *items[r]) Swap(l,r);
  }
  else
  {
    unsigned long k, s, t, w;
    bool jNegativo = false;                        // flag necessária dado que estamos a trabalhar com unsigned
    unsigned long pivot = ChoosePivot(l,r,depth);  // indice do elemento pivô
    Token v = items[pivot]->At(depth);             // valor do elemento pivô
    Swap(l,pivot);                                 // pivo fica de imediato na posição mais à esquerda
    unsigned long i = l+1;                         // indice esquerdo
    unsigned long p = l+1;                         // limite esquerdo dos elementos iguais ao pivô
    unsigned long j = r;                           // indice direita
    unsigned long q = r;                           // limite direito dos elementos iguais ao pivô

    for(;;)
    {
      while(i <= j && items[i]->At(depth) <= v)               // percorre metade esquerda
      {
        if(items[i]->At(depth) == v) {Swap(p, i); p++;}       // se igual passa para a esquerda
        i++
      }
      while(!jNegativo && j >= i && items[j]->At(depth) >= v) // percorre metade direita
      {
        if(items[j]->At(depth) == v) {Swap(q, j); q--; }      // se igual passa para a direita
        if(j==0) jNegativo = true; else j--;
      }
      if (i > j || jNegativo)break;                           // se i ultrapassou j então acabou
      Swap(i, j);                                             // troca [i] com [j]
      i++;                                                    // incrementa i
      if(j==0) jNegativo = true; else j--;                    // decrementa j
    }
    s = MIN(p-l,i-p);
    for(k = l, w = i-s; s ; s--, k++, w++) Swap(k, w);        // move iguais da esquerda para o centro
    s = (jNegativo? MIN(q+1,r - q) : MIN(q - j,r - q));
    for(k = i, w = r-s+1; s; s--, k++, w++) Swap(k, w);       // move iguais da direita para o centro
    s=i-p;
    if(s>1) MKQuickSort(l, l+s-1, depth, maxDepth);           // partição esquerda
    if(depth < maxDepth && (p-l+r-q) > 1)                     // partição central
       MKQuickSort(i-(p-l), i+r-q-1, depth+1, maxDepth);      // avança uma posicao
    t= (jNegativo? q+1: q-j);
    if(t>1) MKQuickSort(r-t+1, r, depth, maxDepth);           // partição direita
  }
}
```

Figura 4.26 – Código do *Multikey QuickSort*

A implementação eficiente do método de criação de três partições, a dos elementos maiores que o pivô, a dos elementos iguais ao pivô e a dos elementos menores que o pivô, é devida a Bentley e a McIlroy (Sedgewick 1998). O algoritmo começa por deslocar para os extremos esquerdo e direito[52] os elementos iguais ao pivô, para a esquerda do pivô os elementos menores e para a direita do pivô os elementos maiores. No final reúne todos os elementos iguais ao pivô, no centro do vector. Importante

[52] Numa representação normalmente aceite em que o menor índice está à esquerda.

realçar que os elementos iguais ao pivô, ficam de imediato nas suas posições definitivas. Após a última troca, aplica recursivamente a cada uma das três partições o mesmo algoritmo, sendo que para a partição central incrementa de uma unidade a posição em análise (Sedgewick 1998).

Estando a trabalhar com *strings* e sendo o *Quicksort* baseado em sucessivas trocas[53] de elementos, optou-se, com algum sacrifício ao nível da eficiência espacial, pela utilização de vectores de apontadores para *n-gramas* em vez de vectores de *n-gramas*. Os ganhos em tempo, produzidos por esta opção, são visíveis experimentalmente e resultam da redução do peso das operações de troca de elementos, executadas durante o processo de ordenação. Em vez de copiar estruturas inteiras, passa-se somente a copiar apontadores. A classe *ISVector<T>* corresponde a este tipo de implementação (ver Figura 4.26).

```
template <class T>
class ISVector {
  private:
    typedef T * TPointer;
  private:
    TPointer *items;
  ...
  private:
    void MKQuickSort(unsigned long l, unsigned long r, unsigned short depth, unsigned short maxDepth);
    unsigned long ChoosePivot(unsigned long l, unsigned long r, unsigned short depth) const;
  ...
}
...
template<class T>
unsigned long ISVector<T>::ChoosePivot(unsigned long l, unsigned long r, unsigned short depth) const
{
  unsigned long nr = r - l + 1;       // numero de items
  unsigned long pm = l + nr/2;        // ponto medio
  if(nr>7)
  {
    unsigned long pl = l;
    unsigned long pn = r;
    if(nr>40)
    {
      unsigned long s = nr >> 3;
      pl = MED3(pl, pl+s, pl+s+s);
      pm = MED3(pm-s, pm, pm+s);
      pn = MED3(pn-s-s, pn-s, pn);
    }
    pm=MED3(pl, pm, pn);
  }
  return pm;
}
```

Figura 4.27 – Escolha do elemento *pivô* para o *Multikey QuickSort*

Na implementação do *Multikey Quicksort* optou-se por utilizar o método conhecido como *Median-of-Three* para a escolha do elemento pivô. Neste método o elemento pivô escolhido é um valor médio calculado com base no conjunto de elementos formado pelo elemento mais à esquerda, o elemento mais à direita e o elemento central do vector (Sedgewick 1998), (Larsson *et al.* 1999). Este método aumenta as probabilidades do elemento escolhido como pivô ser precisamente o elemento que melhor

[53] *Swaps no original*

divide o vector em duas partes iguais, ou seja, o valor médio de todos os elementos em ordenação. Da aplicação deste método podem resultar ganhos na ordem dos 20 a 25% em relação a uma implementação mais directa (Sedgewick 1998).

O código para a função *ChoosePivot* que implementa o método *Median-of-Three* é mostrado na Figura 4.27, existindo, na literatura referenciada, implementações alternativas mas em tudo semelhantes a esta. A macro *MED3*(a, b, c) escolhe entre os três índices, que recebe como argumento, aquele que corresponde ao valor médio dos elementos referenciados.

O *Multikey Quicksort* apresenta como principais vantagens as seguintes:

1. É independente da dimensão do alfabeto. No nosso caso é independente do número de diferentes *tokens* existentes no *corpus* (Sedgewick 1998);
2. Não necessita de analisar todas as posições de todas as *strings* bastando-lhe ordenar os vários prefixos que diferenciam as diferentes *strings*. Quando uma partição tem um só elemento não é necessário continuar a analisar as restantes posições dessa *string* (Sedgewick 1998).
3. Tem as características do *Radix Sort* sem necessitar de qualquer estrutura adicional para ser executado com excepção de espaço em *stack* para as chamadas recursivas que é, em média, proporcional a *log n*, onde *n* é o número de elementos a ordenar (McIlroy *et al.* 1993);
4. É mais eficiente que a versão base do *Quicksort*. A versão base utiliza um total de *m2n log n* comparações para ordenar *n* elementos de dimensão *m*, que diferem apenas na última posição da *string*, enquanto que o novo algoritmo necessita somente de *mn + 2n log n* comparações. A primeira parcela corresponde à separação dos *n* elementos em sub-conjuntos com iguais prefixos, diferenciadores dos restantes, e a segunda à ordenação dos vários sub-conjuntos (Sedgewick 1998). O novo algoritmo tem complexidade $O(mn + n \log n)$, que é equivalente a $O(n \log n)$ se considerarmos *log n* superior a *m* como será, normalmente, expectável;
5. O algoritmo trabalha bem, com a introdução dum *overhead* praticamente desprezível, mesmo para o caso em que não haja *strings* repetidas (Sedgewick 1998);
6. O algoritmo é linear no caso extremo de todos os elementos serem iguais (Sedgewick 1998).

O tempo disponível para a elaboração desta dissertação não permitiu a experimentação de outras alternativas em termos da ordenação do vector de *n-gramas*. O algoritmo *Forward Radix Sort*, proposto por Anderson e Nilson em (Anderson *et al*. 1994), (Nilson 1996) e (Anderson *et al*.1998), parece ser uma boa alternativa, merecendo que seja avaliado como trabalho futuro. Estes autores comparam as várias alternativas de *Radix Sort*, incluindo o proposto *Forward Radix Sort* e o *Multikey Quicksort*, mostrando, apesar do seu trabalho ser em defesa do *Forward Radix Sort*, que o *Multikey Quicksort* é uma das melhores opções para ordenação de *strings*.

Noutra perspectiva e dado que o objectivo principal da ordenação de *n-gramas* é a contagem do número de ocorrências faz todo o sentido ser igualmente estudada, em trabalho futuro, a utilização de algoritmos de contagem do tipo *Key-Indexed Counting* (Sedgewick 1998).

4.5.2 CASOS 1 E 3 - ORDENAÇÃO DO CORPUS

Para os casos 1 e 3 foi tomada a opção pela utilização do irmão mais velho do *Multikey Quicksort*, ou seja, o *Ternary Quicksort*. Este algoritmo apresenta as mesmas vantagens já apresentadas para o *Multikey Quicksort*, excepto aquelas que têm a ver directamente com a sua vertente *Radix Sort* (i.e. ordenação de *strings*), pelo que não vale a pena repeti-las (Sedgewick 1998).

No caso 3, onde o vector é constituído por valores (posições dentro do *corpus*) totalmente distintos e sem qualquer repetição, a opção pela versão base do *Quicksort* ou outro algoritmo similar seria igualmente adequada. Optou-se pela utilização dum algoritmo comum por razões meramente de comodidade na implementação e por esta opção não apresentar prejuízos significativos ao nível da eficiência global do sistema.

O *Ternary Quicksort* tem complexidade igual à do *Quicksort*, $O(N \log N)$, mas é na prática, quando existem elementos repetidos, mais eficiente por convergir mais depressa para partições com poucos elementos (Sedgewick 1998).

4.6 ESTRUTURA DE CLASSES

Na Figura 4.28 estão representadas, na forma dum diagrama UML (Rumbaught *et al*. 1999), as diferentes classes implementadas e as suas relações de dependência.

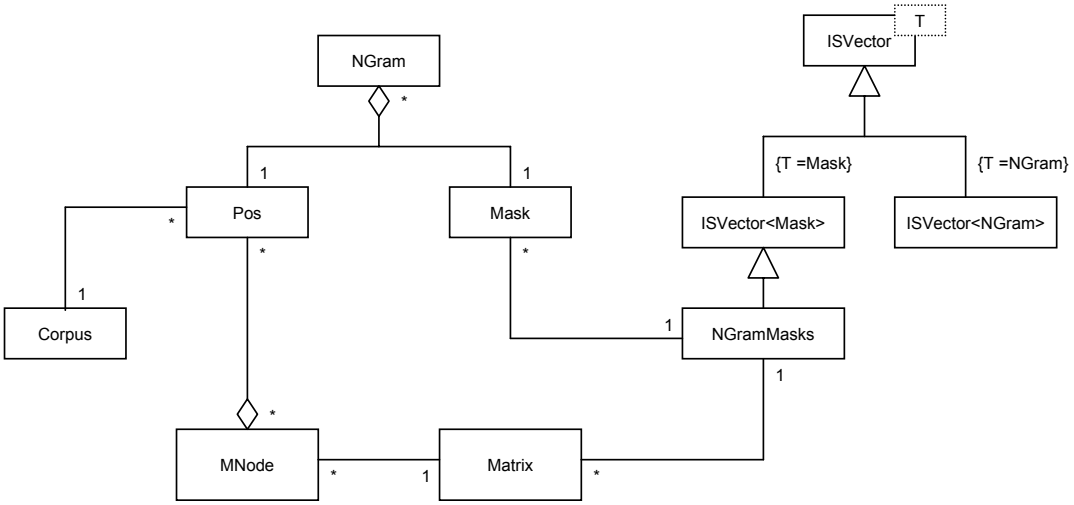


Figura 4.28 – Diagrama de classes

Necessariamente é um diagrama simples dado que a preocupação fundamental foi a de criação duma estrutura de dados leve e muito eficiente do ponto de vista espacial.

Reconhece-se facilmente o conjunto de classes referenciadas ao longo deste capítulo:

- *NGram*: Abstracção do conceito de *n-grama* agregando uma posição e uma *máscara*;
- *Pos*: Abstracção do conceito de apontador para uma posição do *corpus*;
- *Mask*: Abstracção do conceito de *máscara*;
- *NGramMasks*: Vector de *máscaras*. Herda os atributos e as operações dos vectores de apontadores para *máscaras* representados pela classe *ISVector<Mask>*. Dá origem a um único objecto;
- *Corpus*: Abstracção do conceito de *corpus*. É na sua essência um vector de *tokens* dividido por documentos. Dá origem a um único objecto;
- *MNode*: Abstracção do conceito de *n-grama* quando este pertence ao conjunto ☚$_d$ e está associado à *matrix*;
- *Matrix*: Implementação da estrutura matricial de indexação dos *n-gramas*. Dá origem a um único objecto;
- *IsVector<T>*: Template da classe base contendo os atributos e as operações necessárias à abstracção dum vector de apontadores para objectos, nomeadamente o método de ordenação. Dá origem a duas classes concretas, uma para o vector de apontadores para *máscaras* e outra para o vector de apontadores para *n-gramas*, *IsVector<NGram>*, da qual é instanciado o vector *vect* ☚, utilizado na função de contagem do número de ocorrências de cada *n-grama*.

Os detalhes de implementação de cada uma destas classes foram sendo dados ao longo deste capítulo.

Nenhuma das classes programadas contém membros virtuais, por não ser importante para o tipo de arquitectura de classes adoptada, muito simplificada, e dado ter-se verificado que este qualificador implicava o consumo de mais 4 *bytes* por objecto criado.

4.7 CÁLCULO DE COMPLEXIDADES

Nesta secção é efectuada a análise da complexidade global do sistema, complexidade esta que somente dá, por definição, a ordem de grandeza da dependência da eficiência temporal e espacial do sistema relativamente às três variáveis principais do sistema: a dimensão *N*, em *tokens*, do *corpus*; a dimensão *m* da janela de análise e a dimensão |☒| do *dicionário*, ou seja, o número de *tokens* diferentes existentes no *corpus*. O que é importante determinar é a forma como varia a eficiência do sistema em função destas três variáveis. É principalmente importante determinar a dependência da eficiência do

sistema relativamente à dimensão *N* do *corpus*. O valor de *m* é importante mas será na maioria dos casos constante. No entanto, é necessário ter presente que um incremento de *m* implica um incremento exponencial de *M'*, facto que deverá ser tomado em consideração na utilização do sistema. Mesmo pequenas variações de *m* têm influência significativa na eficiência global do sistema.

Todo o processamento inicial necessário para a criação do vector de *máscaras* é totalmente desprezível, quer em espaço como em tempo, quando comparado com o restante processamento pelo que não está incluído nesta análise. É interessante verificar que é precisamente este processamento desprezível que permite atingir os níveis de eficiência demonstrados.

4.7.1 COMPLEXIDADE TEMPORAL

A análise efectuada neste capítulo baseia-se na estrutura ideal, mostrada na Figura 4.17, correspondente à utilização da *matrix* como estrutura base para as operações de procura de *n-gramas*.

O cálculo dum valor para a complexidade temporal global do sistema não é simples dado que depende de factores não totalmente dependentes de *N* ou de *m*, como sejam, a cardinalidade do *dicionário* e o número de *n-gramas* com frequência não unitária existentes. Na Figura 4.29 são apresentados os componentes principais que contribuem para a complexidade temporal do sistema.

Etapa	Acções	Equações
Ordenação do Corpus	Ordenar N elementos	N x log N ***O*(N x log N)**
Cálculo da *frequência* para todos os *n-gramas*	h(*m*) x N *n-gramas* gerados + h(*m*) x |Σ| ordenações parciais de, em média, N / |Σ| *n-gramas* com, em média, m/2 *tokens*	h(*m*) x N + h(*m*) x |Σ| x (N / |Σ| x m/2 + N / |Σ| log N / |Σ|) = h(*m*) x N x (1 + m/2 + log(N / |Σ|)) ***O*(h(*m*) x N x log(N / |Σ|))** considerando m/2 <= log N / |Σ|
Criar *MNodes* e preencher *matrix*	Preencher h(*m*) x N células + *K(N,m)* novos *MNodes* + *U(N,m) novas Pos em MNodes*	h(*m*) x N + *K*(*N,m*) + *U*(*N,m*) ***O*(h(*m*) x N)**
Cálculo da *ME* para todos os *n-gramas*	h(*m*) x N comparações + *K*(*N,m*) calculos de *ME* com, em média, *m*/2 somas	h(*m*) x N + *K*(*N,m*) x *m*/2 ***O*(h(*m*) x N + *m x K*(*N,m*))**
WriteMWUs	h(*m*) x N comparações + *K*(*N,m*) chamadas a *IsLocalMax* com, em média, *m*/2 + *m*/2 x *U*(*N,m*) / *K*(*N,m*) chamadas a *ME*	h(*m*) x N + *K*(*N,m*) x *m*/2 x (1+ *U*(*N,m*) / *K*(*N,m*)) ≅ h(*m*) x N + *K*(*N,m*) x *m*/2 x *U*(*N,m*) / *K*(*N,m*) = h(*m*) x N + *m*/2 x *U*(*N,m*) ***O*(h(*m*) x N + *m x U*(*N,m*))**

•O número de máscaras *M'* é função de *m* pelo que está representado por *h(m)*
•*U(N,m)* é uma função de *N* e de *m* que dá o número de *n-gramas* com *frequência* não unitária
•*K(N,m)* é uma função de *N* e de *m* que dá o número de *n-gramas* diferentes existentes com frequência não unitária

Figura 4.29 – Esquema para a análise da complexidade temporal assintótica

A complexidade temporal baseia-se na soma das complexidades dos algoritmos apresentados na Figura 4.13, correspondente ao cálculo das *frequências,* na Figura 4.21, que contém o algoritmo de preenchimento da *matrix*, na Figura 4.18, respeitante ao algoritmo de cálculo da *ME*, na Figura 4.19, correspondente à função *IsLocalMax*, e na Figura 4.24, correspondente ao ciclo de identificação das *MWUs*. Adicionalmente, a complexidade do sistema está dependente do algoritmo de pesquisa de *n-gramas*, para obtenção da respectiva *frequência* e restantes atributos, durante o cálculo da *ME* e no *GenLocalMaxs*. Finalmente, é preciso ter em linha de conta tudo o que foi dito, na secção 4.5, relativamente ao algoritmo escolhido para ordenação dos *n-gramas* e do *corpus*. Somando todos os componentes obtemos a complexidade global do sistema (ver Figura 4.30).

$$
\begin{array}{lll}
 & O(\,N \times \log N\,) & \\
 & O(\,h(m) \times N \times \log(N / |\Sigma|)\,) & \\
 & O(\,h(m) \times N\,) & \\
 & O(\,h(m) \times N + K(N,m) \times m\,) & \\
+ & O(\,h(m) \times N + m \times U(N,m)\,) & \\
\hline
= & O(\,N \times \log N + h(m) \times N \times \log(N / |\Sigma|) + m \times (K(N,m) + U(N,m))\,) & \\
= & O(\,N \times \log N + h(m) \times N \times \log(N / |\Sigma|) + m \times U(N,m)\,) & \text{dado que } K(N,m) \text{ é sempre menor que } U(N,m) \\
= & O(\,N \times (\log N + h(m) \times \log(N / |\Sigma|)\,) & \text{porque } U(N,m) \text{ é sempre menor ou igual a } h(m) \times N \\
 & & \text{e considerando, mais uma vez, } m/2 <= \log(N / |\Sigma|)
\end{array}
$$

Figura 4.30 – Cálculo da complexidade temporal assintótica do sistema

O número |☒| de *tokens* diferentes presentes no *corpus*, ou seja, a cardinalidade do *dicionário*, é normalmente elevado dado que corresponde ao conjunto de palavras, com todas as derivações de género, número e conjugação, de sinais de pontuação, de números e de siglas existentes em textos escritos em qualquer língua natural. No entanto, é normalmente muito inferior ao valor de *N*, ou seja, ao número de *tokens* presentes no *corpus*. É neste contexto que o valor de |☒| é tomado em conta no cálculo da complexidade do sistema. A sua importância será cada vez menor quanto maior for o N dado que é normal que cresça muito mais devagar que N.

O número de *n-gramas* com *frequência* não unitária depende de características próprias de cada *corpus*, pelo que não é possível identificar uma curva de evolução típica para as funções $U(N,m)$ e $K(N,m)$ utilizadas na tabela da Figura 4.29. No entanto, para *corpus* baseados numa língua natural, corresponderão a valores baixos quando comparados com o total de *n-gramas* associados ao *corpus*.

Tomando em consideração o que foi dito relativamente a |☒|, $U(N,m)$ e $K(N,m)$, podemos afirmar que o peso do algoritmo, tomando *m* como constante, está nos dois primeiros passos: ordenação do *corpus* e obtenção da *frequência* de cada *n-grama* e que esta é quase totalmente dependente de *N log N*, dado

que a parte logarítmica *log*(*N* / |☒|) está muito atenuada pela existência do divisor |☒|. Esta constatação deriva do facto do algoritmo se basear na divisão prévia do conjunto de *n-gramas* em pequenos conjuntos que somente então são ordenados. Cada ordenação é uma operação de complexidade *O*(n *log n*) mas para um número *n* muito inferior de elementos. O factor logarítmico será cada vez mais preponderante com o incremento do *N*, a que não corresponderá, como já foi dito, um incremento directo de |☒|.

4.7.2 COMPLEXIDADE ESPACIAL

Em termos espaciais a complexidade é obtida pela análise da estrutura apresentada na figura 4.17. Dado que *N* é muito superior a *M'*, podemos desprezar o espaço ocupado pelo vector de *máscaras*. Assim sobram essencialmente 4 estruturas de dados importantes:

1. *Corpus*: É uma estrutura de dimensão *O*(*N*), ou seja, tem um crescimento linear em *N*;
2. Vector ☚: É um vector de dimensão *O*(*N*), mais uma vez, com crescimento linear em *N,* independente do valor de *m*;
3. *Matrix*: É um vector duplo de dimensão *O*(*N % M'*) = *O*(*N % h*(*m*)) em que *h*(*m*) é uma função não linear em *m*. Se tomarmos *m* constante então a *matrix* cresce linearmente em *N*. Para um mesmo *N* cresce não linearmente em *m*;
4. *MNodes*: O total de *MNodes* criado é uma percentagem baixa, em geral menor que 5%, para *corpus* baseados numa língua natural, do total de *n-gramas* diferentes gerados. Assim, o total de *MNodes* é, à parte duma constante, *O*(*N % M'*) , ou seja, *O*(*N % h*(*m*))

Resumidamente, temos uma parte da estrutura dependente linearmente do valor de *N*, o *corpus* e o vector ☚, e uma outra parte, a *matrix* e os *MNodes*, dependentes linearmente do valor de *N* e não linearmente de *m*.

Somando tudo temos *O*(*N*) + *O*(*N*) + *O*(*N % h*(*m*)) + *O*(*N % h*(*m*)), o que é o mesmo que *O*(*N % h*(*m*)). Em termos gerais a estrutura é linearmente dependente de *N* e não linearmente de *m*. Fica igualmente claro o peso preponderante da *matrix* no total da estrutura, justificando o investimento efectuado na procura de alternativas de redução da sua dimensão através das opções apresentadas na secção 4.4.

5 DESENVOLVIMENTO DO PROTÓTIPO

Esta dissertação correspondeu essencialmente a um trabalho experimental. Foram as sucessivas experiências que conduziram à solução apresentada no capítulo 4. Para tal foi sendo progressivamente desenvolvido e refinado um protótipo, totalmente funcional, correspondente às várias fases e opções atrás apresentadas.

Este capítulo descreve o protótipo final do ponto de vista dos vários módulos implementados fazendo, sempre que necessário, a ponte com a solução atrás descrita.

O protótipo foi integralmente desenvolvido em C++. O código inicial foi baseado nas ideias desenvolvidas pelo Professor Pedro Guerreiro no livro (Guerreiro 2000), que em conjunto com os livros (Rodrigues *et al.* 1998), (Rodrigues *et al.* 2000) e os manuais *on-line* do compilador utilizado para desenvolvimento do protótipo (*Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland*), constituíram as referências principais em termos da utilização da linguagem.

5.1 ARQUITECTURA GLOBAL DO PROTÓTIPO

O capítulo 4 foi dedicado à apresentação detalhada da solução encontrada para a implementação eficiente do algoritmo *GenLocalMaxs* e da medida de associação *ME*. Estes são os dois componentes fundamentais do módulo principal do protótipo desenvolvido, o *Extractor*. No entanto, este módulo necessita de dois outros módulos auxiliares, o *Tokenizer* e o *Translator*, para que possa trabalhar com *corpus* baseados em diferentes linguagens e formatos.

O *Tokenizer* filtra e transforma os documentos do *corpus* numa sequência de *tokens* passível de ser tratada pelo *Extractor*. Esta transformação é necessária para tornar o processo de extracção de *MWUs* independente da linguagem em que o *corpus* esteja expresso e para que seja cumprido o pressuposto, assumido na apresentação da solução, de que o *corpus* seria uma sequência de *tokens* (valores inteiros positivos) dividida em documentos.

O *Translator* faz a operação inversa, isto é, recebe o conjunto de *MWUs*, expressas como sequências de *Tokens*, traduzindo-o para a linguagem original do *corpus*.

A Figura 5.1 mostra, esquematicamente, os módulos constituintes do protótipo desenvolvido, as interfaces entre os módulos e as entradas e saídas do sistema.

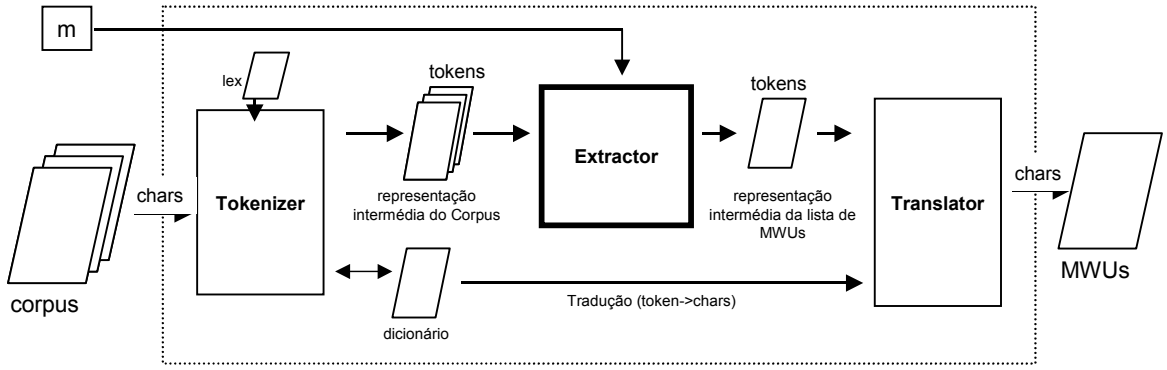


Figura 5.1 - Arquitectura global do sistema

O sistema recebe um *corpus* na forma dum ficheiro contendo a lista dos documentos que o compõem e um valor *m* para o tamanho da janela de análise. Cada documento é um ficheiro de texto. Após o processamento devolve um ficheiro de texto com o conjunto de *MWUs* identificadas.

Um extracto dum ficheiro do *corpus* do Público/MCT, utilizado nas experiências efectuadas, é mostrada na Figura 5.2.

```
....
<s>
E o dinheiro «não falta só às câmaras», lembra o secretário de Estado, que considera que a solução para as autarquias é «especializarem-se em fundos comunitários».
</s><ext n=1 sec=clt sem=92b>
<t>
Um revivalismo refrescante
</t>
<p>
O 7 e Meio é um ex-libris da noite algarvia.
</s>
<s>
É uma das mais antigas discotecas do Algarve, situada em Albufeira, que continua a manter os traços decorativos e as clientelas de sempre.
</s>
<s>
É um pouco a versão de uma espécie de «outro lado» da noite, a meio caminho entre os devaneios de uma fauna periférica, seja de Lisboa, Londres, Dublin ou Faro e Portimão, e a postura circunspecta dos fiéis da casa, que dela esperam a música «geracionista» dos 60 ou dos 70.
</s>
<s>
Não deixa de ser, nos tempos que correm, um certo «very typical» algarvio, cabeça de cartaz para os que querem fugir a algumas movimentações nocturnas já a caminho da ritualização de massas, do género «vamos todos ao Calypso e encontramo-nos na Locomia».
....
```

Figura 5.2 – Extracto do *corpus* do Público/MCT utilizado nas experiências

Os três módulos, *Tokenizer*, *Extractor* e *Translator*, são totalmente independentes entre si, comunicando através da troca de ficheiros (*representação intermédia do corpus*, *representação intermédia da lista de MWUs* e *dicionário*). Assim, as interfaces entre eles estão concretizadas ao nível do formato dos ficheiros trocados.

O *dicionário* não é utilizado, no processo de extracção das *MWUs*, dado que o significado de cada *token* e a real relação de ordem entre as diferentes *unidades lexicográficas* não é importante para este processo. Para o *Extractor*, o *corpus* é uma sequência de números inteiros naturais, conjunto este onde existe uma relação de ordem naturalmente estabelecida entre os vários elementos. O *dicionário* volta a ser importante na fase final do processo para a tradução e ordenação das *MWUs* identificadas.

A criação da *representação intermédia* poderá ser efectuada antecipadamente ou imediatamente antes do processo de extracção das *MWUs*. A *representação intermédia* é um conjunto de ficheiro binários, em igual número dos existentes no *corpus* original, contendo a sequência de números inteiros naturais correspondentes à sequência de *tokens.*

Os vários módulos são descritos em seguida, sendo que o *Extractor* teve já larga descrição ao longo da apresentação da solução no capítulo 4.

5.2 TOKENIZER

A função principal do *Tokenizer* é transformar a sequência de caracteres correspondente ao *corpus*, numa sequência equivalente de *tokens*, designada por *representação intermédia*. Esta transformação é efectuada de acordo com um conjunto de regras elementares de reconhecimento de *unidades lexicográficas*, derivadas das regras gramaticais subjacentes à língua ou linguagem a que o *corpus* pertence.

A representação intermédia corresponde a uma sequência de *tokens*, dividida em igual número de ficheiros que os do *corpus*, totalmente equivalente ao original, com excepção do que for eliminado pelo filtro associado às regras de reconhecimento de *unidades lexicográficas*. Em paralelo, cria o *dicionário*, associando a cada *unidade* lexicográfica um *token*. Este *dicionário* permitirá que, mais tarde, seja efectuada a operação inversa de tradução de *tokens* em *unidades lexicográficas*, expressas na linguagem original do *corpus*.

No protótipo do *Tokenizer*, desenvolvida para os testes do sistema, tomámos o conjunto total de caracteres como o determinado pela tabela ASCII estendida, ou seja, 256 caracteres. Esta limitação aplica-se somente às versões desenvolvidas para o *Tokenizer* e para o *Translator*. O *Extractor* é totalmente independente do conjunto de caracteres base ao *corpus*. Outras versões do *Tokenizer* e do

Translator poderão ser desenvolvidas para satisfazer necessidades concretas de outros tipos de *corpus*, sem que para tal seja necessário alterar o *Extractor*.

A estrutura interna do *Tokenizer* depende do formato e do conteúdo dos ficheiros do *corpus* que se está a analisar mas é genericamente uma variação da estrutura mostrada na Figura 5.3. A versão desenvolvida segue esta estrutura e tem por base o formato dos ficheiros do *corpus* do Público/MCT, utilizado como base de trabalho ao longo dos testes do protótipo desenvolvido, mas pode ser utilizada para outros *corpus* que sigam um formato e regras de construção semelhantes.

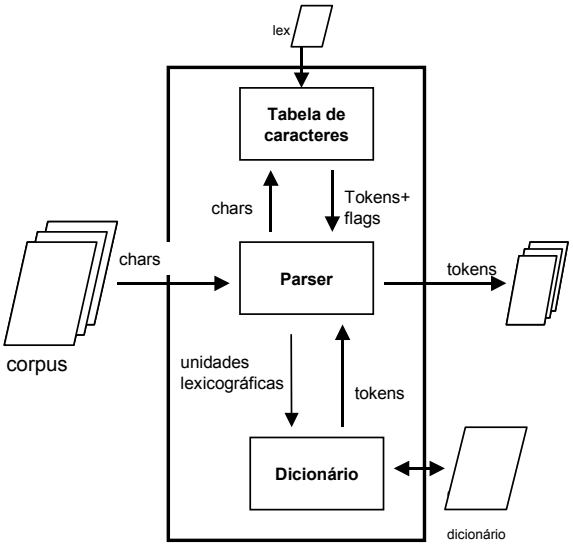


Figura 5.3 – Estrutura interna do *Tokenizer*

São reconhecíveis três objectos principais: a *tabela de caracteres*; o *parser*; o *dicionário*. A primeira é simplesmente um vector correspondente aos 256 caracteres da tabela ASCII, contendo, para cada um deles, informações que caracterizam e agrupam os caracteres e que guiam a extracção de informação pelo *parser*[54]. A estrutura desta tabela é mostrada na Figura 5.4, realçando-se a função de cada um dos campos constituintes.

[54] A noção de *parser* é aqui, um pouco abusivamente, estendida ao que normalmente é designado por analisador lexicográfico na literatura sobre compiladores.

```
struct {
  bool isTrash;              // para ignorar
  bool isLetter;             // letra
  bool isDigit;              // digito
  bool isDelimiter;          // delimitador
  bool isBeginTrash;         // inicio de sequência a ignorar
  bool isEndTrash;           // fim de sequência a ignorar
  bool isHiffen;             // hífen
  bool isDot;                // ponto
  bool isComma;              // virgula
  Token token;               // identificador atribuído
  char * identifier;         // string identificadora
} cTab[256];
```

Figura 5.4 – Estrutura da *tabela de caracteres*

Esta informação está, em parte, no ficheiro de regras passado como parâmetro. Este ficheiro é a forma encontrada para, duma forma simples, poder-se configurar o comportamento do *Tokenizer* na forma como os diferentes caracteres devem ser agrupados. Contém regras nas formas "*identificador = lista de códigos de caracteres*" ou "*função = lista de identificadores*", que permitem identificar a função de cada um dos caracteres (e.g. delimitadores, letras, dígitos).

A Figura 5.5 mostra exemplos reais de algumas regras implementadas. Este ficheiro deverá ser estendido, incorporando regras para o reconhecimento das *unidades lexicográficas*, para tornar ainda mais flexível a configuração do *Tokenizer*.

```
...
LINEFEED = 10
CARRIAGERETURN = 13
SPACE = 32
TAB = 9
MAIUSCULA = 65-90
MINUSCULA = 97-122
LETRAESPECIAL = 138, 140, 142, 154, 156, 158-159, 161-163, 188-190, 192-246, 248-254
DIGITO = 48-57

<DELIMITER> = LINEFEED, CARRIAGERETURN, SPACE, TAB
<LETTERS> = MAIUSCULA, MINUSCULA, LETRAESPECIAL
<DIGITS> = DIGITO

<PONTO> = PONTO
<BEGINTRASH> = MENOR
<ENDTRASH> = MAIOR
<HIFFEN> = MENOS
<DOT> = PONTO
<COMMA> = VIRGULA
....
```

Figura 5.5 – Extracto do ficheiro de regras

O filtro implementado ignora todas as *tags*, ou seja, todos os caracteres incluídos entre os símbolos "<" e ">". São considerados delimitadores os caracteres espaço, tab, fim-de-linha e fim-de-ficheiro e são reconhecidas como *unidades lexicográficas* as seguintes:

- Sequências de maiúsculas e minúsculas, podendo estar ligadas por hífens;
- Sequências de dígitos, podendo estar ligadas por virgulas e pontos;
- Caracteres isolados de pontuação, como seja, o ponto de interrogação.

O processo de transformação, papel do *parser*, assenta numa tabela de equivalência, designada por *dicionário*, entre cada *unidade lexicográfica* e o respectivo *token*. O *parser* é um autómato finito que faz a leitura sequencial dos caracteres do *corpus*, gerando os *tokens* de acordo com regras de reconhecimento de *unidades lexicográficas* residentes numa tabela de mudança de estado/acções. Apela ao *dicionário* para a atribuição dum *token* a cada *unidade lexicográfica* identificada.

O executável do *Tokenizer*, recebe como parâmetros o nome de dois ficheiros. O primeiro contendo a lista dos ficheiros de texto que constituem o *corpus*, contendo um nome de ficheiro por linha. O segundo contendo um conjunto de regras descritivas do conjunto de caracteres presente no *corpus* de acordo com o formato atrás enunciado. Cria um ficheiro, com extensão ".int", contendo a lista de ficheiros da representação intermédia do *corpus*, e um ficheiro intermédio por cada ficheiro do *corpus* original, acrescentando a extensão ".int". Os ficheiros da *representação intermédia* são ficheiros binários contendo uma sequência de *tokens* (valores inteiros a 4 *bytes*). Cria ainda o ficheiro do *dicionário*, com a extensão ".dic", na forma dum ficheiro de texto a duas colunas. A primeira coluna é o *token*, a segunda é a correspondente *unidade lexicográfica*. Na primeira linha é incluído o número total de *tokens* presentes.

5.2.1 IMPLEMENTAÇÃO DO DICIONÁRIO

A implementação da classe de suporte ao *dicionário* foi motivo de alguns cuidados dado que a eficiência da operação de atribuição dum *token* a cada *unidade lexicográfica* identificada pelo *parser* é a principal contribuinte para a eficiência global do módulo. Deste modo necessita ter uma implementação eficiente em termos das operações de procura e de inserção, essenciais durante a sua construção, ou seja, ao longo do processo de transformação do *corpus* original para a *representação intermédia*.

Para que a eficiência seja máxima e porque na maioria das aplicações, nomeadamente no tratamento de textos em língua natural, terá dimensão reduzida, deve estar totalmente em memória durante a sua construção. No entanto, é importante que tenha uma representação externa correspondente para que

possa ser reutilizado em diferentes sessões e para que possa servir, na operação inversa, como base ao processo de tradução da *representação intermédia* do conjunto de *MWUs* para uma representação final na mesma língua do *corpus* original. No protótipo desenvolvido não foi adicionada a hipótese do *dicionário* poder ser reutilizado entre sessões. No entanto o desenvolvimento desta funcionalidade é relativamente simples.

A construção do *dicionário* dá-se em simultâneo com a transformação do texto original para a *representação intermédia*. No protótipo desenvolvido, assenta numa estrutura em árvore designada por *hybrid ternary search trie (hTST)* descrita em (Bentley *et al.* 1998) e em (Sedgewick 1998). Uma *hTST* é uma estrutura muito interessante, simples e muito eficiente para a construção de tabelas de símbolos. É constituída por um vector ao primeiro nível contendo apontadores para *ternary search tries*, ou abreviadamente *TSTs*. O vector garante busca em tempo constante para o primeiro caracter e as *TSTs* suportam a representação dos restantes caracteres das diferentes *strings* com tempos de inserção e de pesquisa logarítmicos, mas muito baixos dada a altura média das árvores ser normalmente pequena (Sedgewick 1998).

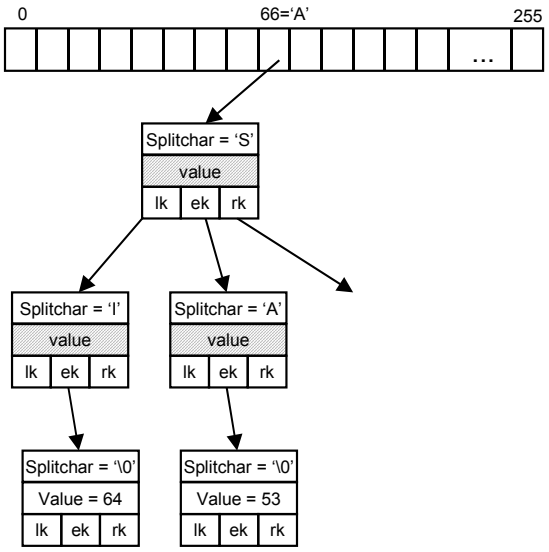


Figura 5.6 – Estrutura duma *hTST*

Na Figura 5.6 é apresentada a estrutura duma *hTST* com dois exemplos. O *token* 64 que corresponde à *string* "AI" e o *token* 53 que correponde à *string* "ASA". Foi adoptada a representação de *strings* tal como na linguagem C, ou seja, com o caracter '\0' como símbolo terminal.

A ideia de utilização dum vector ao primeiro nível advém da constatação que a estrutura ideal, em termos de eficiência temporal, para a representação de tabelas de símbolos é uma *trie*. A *trie* é uma árvore sempre balanceada pelo que a eficiência das operações de busca e de inserção é sempre a mesma. O grande senão das *tries* é a sua total ineficiência em termos espaciais. Cada nó é um vector

de apontadores indexado pelo caracter em cada posição das *strings* armazenadas. É fácil de perceber que os primeiros níveis estão geralmente completamente ocupados mas que a partir do terceiro nível existem muitas posições vazias nos vectores. Existem várias soluções para a resolução deste problema, tentando manter níveis idênticos de eficiência temporal, mas reduzindo drasticamente o espaço necessário. Uma das mais interessantes são as *PATRICIA Tries* (abreviadamente *PAT*) e outra são as *hTST*. Têm muitas vantagens e desvantagens em comum sendo que as PAT são mais difíceis de implementar e menos intuitivas (Sedgewick 1998). Esta é a principal razão da opção tomada ter recaído sobre as *hTST*.

O código da função de inserção duma nova *string* na *hTST* é mostrado na Figura 5.7. A função começa por procurar a raiz da árvore, no *array* existente ao primeiro nível da estrutura, com base no primeiro caracter da *string*, chamando então a função de inserção na *TST*. Esta função procura recursivamente, caracter a caracter, a *string* na árvore. Se existir devolve o valor do identificador respectivo. Caso contrário insere a nova *string* e gera um novo valor para a identificar.

```
...
template<class T>
struct TNode
{
  char splitchar;
  TNode* lokid, eqkid, hikid;
  T value;
};

template<class T>
class TSTree
{
  private:
    TNode<T>* root[256];
    ...
  private:
    virtual TNode<T>* _Insert(const char * s, TNode<T>* node, int& v);
  public:
    virtual T Insert(const char * s);
    ...
}

template<class T>
T TSTree<T>::Insert(const char * s)
{
  int res;
  unsigned char c = *s;
  root[c] = _Insert(++s, root[c], res);         // procura raiz com base no primeiro caracter
  return res;                                   // devolve identificador (token)
}

template<class T>
TNode<T>* TSTree<T>::_Insert(const char * s, TNode<T>* node, int& v)
{
  TNode<T>* p = node;

  if(p == 0)                                    // Não encontrou elemento
  {
    p = new TNode<T>;                           // Cria novo nó.
    p->lokid = p->eqkid = p->hikid = 0;
    if( (p->splitchar = *s) == '\0')            // Insere e stop se for fim de string
      v = p->value = count++;                   // Atribui valor para novo elemento
    else
      p->eqkid = _Insert(++s, p->eqkid, v);     // Insere proximo caracter
  }
  else if(*s < p->splitchar)                    // Elemento maior. Vai pela esquerda
    p->lokid = _Insert(s, p->lokid, v);
  else if(*s > p->splitchar)                    // Elemento menor. Vai pela direita
    p->hikid = _Insert(s, p->hikid, v);
  else if(*s == '\0')                           // Elemento igual.
    v = p->value;                               // Se '\0' elemento já existente. Devolve valor.
  else
    p->eqkid = _Insert(++s, p->eqkid, v);       // Avança pelo centro
  return p;
}
```

Figura 5.7 – Função de inserção numa *hTST*

Sedgewick mostra (Sedgewick 1998, *Property 15.8*) que a operação de busca/inserção dum elemento numa *hTST*, com o vector ao primeiro nível indexado pelo primeiro caracter de cada *string*, requer, em média, o acesso a um número de caracteres na ordem de $O(log\ |\boxtimes| - log\ R)$, onde $|\boxtimes|$ é o número total de diferentes *strings* armazenadas e R é o número de diferentes caracteres existentes, ou seja, no nosso caso, 256. O espaço ocupado corresponde a R apontadores para o vector inicial mais $|\boxtimes|$ vezes a média de caracteres de cada *string* vezes o espaço necessário para armazenar um nó de cada *TST*, correspondente a três apontadores mais o caracter de decisão.

A *hTST*, utilizada como base na construção do *dicionário*, poderá ser melhorada em termos de ocupação de memória utilizando técnicas idênticas às utilizadas na construção de *PATs*, eliminando o que é designado por "*one-way branching"* (Sedgewick 1998, exercicio 15.65). Igualmente podem ser utilizadas técnicas de balanceamento dos nós internos das *TSTs* (Sedgewick 1998, exercicio 15.67) para garantir os níveis de eficiência da estrutura. Por falta de tempo e porque experimentalmente a estrutura desenvolvida apresentou níveis de eficiência muito elevados, optou-se por não implementar nenhuma destas melhorias.

5.3 EXTRACTOR

Os componentes principais do *Extractor de MWUs* estão representados na figura 5.8.

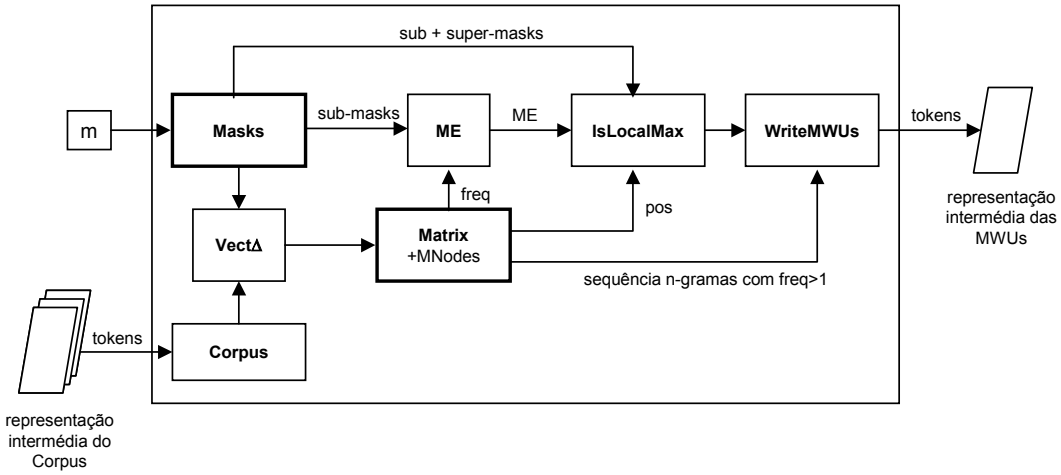


Figura 5.8 – Estrutura interna do *Extractor*

Os componentes representados têm uma correspondência directa com os de igual nome descritos ao longo do capítulo 4 dedicado à apresentação da solução, pelo que não iremos aqui repetir a finalidade e a função de cada um deles. A Figura 5.8 dá essencialmente uma visão global da interligação e interdependência entre os diversos componentes.

Importante realçar que o *Extractor* é totalmente independente do conjunto de caracteres, das regras sintáctico-semânticas e do formato em que o *corpus* está construído. As *representações intermédias*, puramente numéricas, de entrada e de saída, isolam-no completamente do *corpus* original pelo que é aplicável a qualquer língua e a qualquer processo de *tokenização*.

O executável recebe, como parâmetros, o nome do ficheiro do *corpus* contendo a lista de ficheiros intermédios gerados pelo *Tokenizer*, e um valor para a dimensão da janela de análise (*m*). Devolve um

único ficheiro binário, com extensão ".int", contendo a lista de *MWUs* identificadas. Este ficheiro tem o formato mostrado na Figura 5.9.

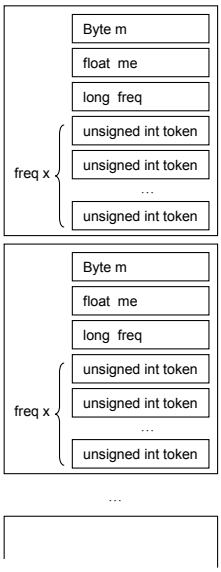


Figura 5.9 – *Representação intermédia* das *MWUs*

De acordo com as várias opções de implementação enunciadas no capítulo 4, foram criadas quatro versões distintas do *Extractor*. As várias versões correspondem às combinações de duas das principais opções de implementação experimentadas:

- *Matrix* implementada como uma matriz real a duas dimensões (Figura 4.17) com os *n-gramas* unitários representados pelo objecto *dummy* (Figura 4.20), ou *matrix* implementada como um vector de vectores (Figura 4.23) onde somente os *n-gramas* não unitários estão representados. A primeira opção de implementação designámos como *Ideal* e a segunda como *Final*. Na versão *Ideal* não são armazenados os valores da *ME*, sendo calculados quando necessários. Na versão *Final* existe cálculo prévio e armazenamento da *ME* para os *n-gramas* representados, ou seja, para os não unitários;
- Armazenar ou não armazenar de forma diferenciada os *n-gramas* associados à *máscara* 0. Designámos por *V1* a alternativa de implementação uniforme e por *V2* a opção pela diferenciação dos *n-gramas* associados à *máscara* zero;

Desta forma temos as seguintes quatro implementações: *Ideal V1*, *Ideal V2*, *Final V1*e *Final* V2. Com estas quatro alternativas pretendeu-se basicamente verificar o impacto de cada uma das opções na

eficiência espacial e temporal do sistema. Os resultados das experiências efectuadas sobre estas quatro versões são apresentados no capítulo 6.

5.4 TRANSLATOR

O *Translator* é um módulo de simples transformação da representação intermédia numérica das *MWUs*, saída do *Extractor,* para uma representação correspondente utilizando os caracteres da língua ou linguagem base do *corpus* original. Traduz as diferentes *MWUs* com base na informação existente no *dicionário*, ou seja, na associação entre *token* e a correspondente *unidade lexicográfica*.

Tem uma implementação extremamente eficiente que permite efectuar a tradução numa só passagem. Para tal o *dicionário* é lido para um vector indexado pelo valor de cada *token*. Deste modo a obtenção da tradução de cada *token* é uma operação de complexidade temporal constante $O(1)$, tendo a tradução no seu todo complexidade linear, ou seja, $O(W)$, onde W é o número total de *MWUs* geradas.

```
0.00006798610411 00003 os <gap> mil contos
0.00005438888184 00003 como <gap> de fundo
0.00003021604425 00002 um <gap> : A
0.00002266203410 00003 que <gap> sobre o
0.00002549478813 00003 um <gap> sobre a
0.00002589946780 00002 da <gap> em Portugal
0.00005179893560 00002 no <gap> desta semana
```

Figura 5.10 – Extracto dum ficheiro com *MWUs*

O ficheiro produzido (ver exemplo na Figura 5.10) é um ficheiro de texto a três colunas, a primeira contém o valor da *ME*, a segunda o valor da *frequência* e a terceira o texto da *MWU*. Os *gaps* são representados pela sequência de caracteres <*gap*>.

A versão do *Translator* desenvolvida tem a capacidade de traduzir a *representação intermédia* da lista de *MWUs* produzida pelo *Extractor* ou em alternativa traduzir ficheiros no formato da *representação intermédia* do *corpus*. Esta opção é interessante para analisar o resultado do processo de *tokenização* antes deste ser tratado pelo *Extractor*. Assim, o executável do *Translator* tem um primeiro parâmetro que indica o tipo de ficheiro de entrada, "-I" para um ficheiro no formato da *representação intermédia* do *corpus* ou "-O" para um ficheiro no formato produzido pelo *Extractor*. Tem mais dois parâmetros. O primeiro indicando o ficheiro contendo a *representação intermédia* da lista de *MWUs* e o último correspondente ao nome do ficheiro contendo o *dicionário* produzido pelo *Tokenizer*.

5.5 QUALIDADE DO CÓDIGO PRODUZIDO

O código final do protótipo, nas suas várias versões, não é o mais elegante em termos de programação sem ser de todo ofensivo para os padrões da boa programação. Sofre da juventude do programador no que se refere a experiência de programação em C++ e sofre das sucessivas alterações que foram sendo introduzidas ao longo das várias experiências realizadas. Dentro do tempo disponível a principal preocupação não foi produzir código definitivo mas tão somente encontrar as melhores estruturas de dados e algoritmos e dar-lhes uma implementação condigna, que permitisse tirar conclusões quanto à sua aplicabilidade.

Algum trabalho deverá ser efectuado no sentido de tornar mais robusta a implementação caso se pretenda tornar este protótipo num produto de utilização corrente. Em qualquer caso o código disponível corre sem quaisquer problemas detectáveis desde que sejam respeitados os formatos para as entradas de cada módulo.

5.6 LIMITES

A utilização de vectores coloca um problema prático relacionado com o valor máximo dos índices. Nas experiência efectuadas, numa máquina equipada com processador *Pentium,* utilizando variáveis do tipo *unsigned long* para os índices, pode-se indexar estruturas com um máximo de 2^{32} elementos, ou seja, quatro giga elementos. A dimensão máxima do *corpus* está unicamente limitada por este factor do seguinte modo:

1. Cada ficheiro do *corpus* não pode ter mais que 2^{32} *tokens* para poder ser lido para memória e indexado;
2. O *corpus* não pode ter mais que 2^{32} *tokens* devido ao vector ☚;

Estes dois limites podem ser ultrapassados recorrendo-se a outro tipo de programação para os índices.

As experiências efectuadas levaram à opção pela utilização de *unsigned int* para representar os *tokens* permitindo um máximo de 2^{32} *tokens* diferentes na arquitectura computacional utilizada. Formatos inferiores (e.g. *unsigned short int*) são facilmente ultrapassados, até por *corpus* somente com algumas dezenas de milhares de *tokens*. Este limite parece-nos difícil de ser ultrapassado em situações normais.

O valor de *m* deverá ser inferior a 256 dado que se estão a utilizar variáveis do tipo *Byte* (8 bits). Se necessário o sistema pode facilmente ser alterado para suportar valores superiores.

6 APRESENTAÇÃO E ANÁLISE DOS RESULTADOS OBTIDOS

Este capítulo é dedicado à apresentação e análise dos valores recolhidos ao longo dum conjunto de testes realizados sobre as quatro versões desenvolvidas do protótipo. Nestes testes a preocupação foi tirar medidas que permitissem avaliar a eficiência, espacial e temporal, da solução proposta, ou seja, procurou-se confirmar a qualidade da solução proposta como resposta ao enunciado do problema subjacente esta tese (i.e. encontrar uma solução eficiente para a implementação dum sistema de extracção de *MWUs* baseado no *GenLocalMaxs* e na *ME*). Procurou-se igualmente justificar as várias opções tomadas e que levaram à construção das quatro alternativas testadas para o *Extractor*.

6.1 PLATAFORMAS DE TRABALHO

O protótipo foi desenvolvido e os testes iniciais foram efectuados numa máquina equipada com um *Pentium III* a 900*MHz* e com 390*MB* de RAM, correndo *Microsoft Windows 2000*. O compilador utilizado é a versão *freeware* do compilador da *Borland* disponibilizada pelo fabricante via *download* (*Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland*). Neste ambiente, todos os executáveis foram produzidos utilizando a opção de geração de código que permite a máxima optimização da velocidade de processamento (i.e. "*-O2 = Generate fastest possible code*").

O sistema foi posteriormente recompilado em *Red Hat Linux release 7.1 (Seawolf)*, na máquina *Terra* pertencente ao laboratório do departamento de informática da FCT/UNL. Esta máquina tem dois processadores *Pentium III* a 900*MHz* e um total de 1GB de RAM. O compilador utilizado é a versão 2.96 do *gcc* para *Linux*. Optou-se pela geração de código com o maior grau de optimização (i.e. "*-O3*").

Em ambos os ambientes foram efectuadas medições do tempo de processamento e monitorizada a memória consumida para várias dimensões de *corpus*. Todos os testes foram efectuados com as máquinas em utilização exclusiva pelos executáveis do protótipo, ou seja, somente concorrendo com os processos, indispensáveis, do sistema operativo. Em ambos os ambientes, cerca de 100MB de memória estavam permanentemente dedicados aos processos do sistema operativo.

Foram unicamente efectuados testes para uma janela de análise de 7 posições (i.e $m = 7$), correspondendo a 43 *máscaras* ou *n-gramas* gerados por cada posição da janela.

Com base nos ficheiros parte01.txt, parte02.txt e parte03.txt do *corpus* do Público/MCT foram criados três conjuntos contendo ficheiros de três diferentes dimensões: 200KB, 500 KB, e 1,5MB. A conjugação de diferentes quantidades de ficheiros destes três conjuntos, permitiu criar um conjunto de 16 *corpus* de dimensões progressivamente crescentes entre 200KB e 34,8 MB, ou seja, entre 33,095 e 5,673,131 *tokens*. Os *corpus*, assim gerados, correspondem à mistura de pequenas noticias relacionadas com vários temas. Por este motivo o número de *tokens* diferentes, ou seja, a cardinalidade do *dicionário*, é igualmente crescente (ver Figura 6.1).

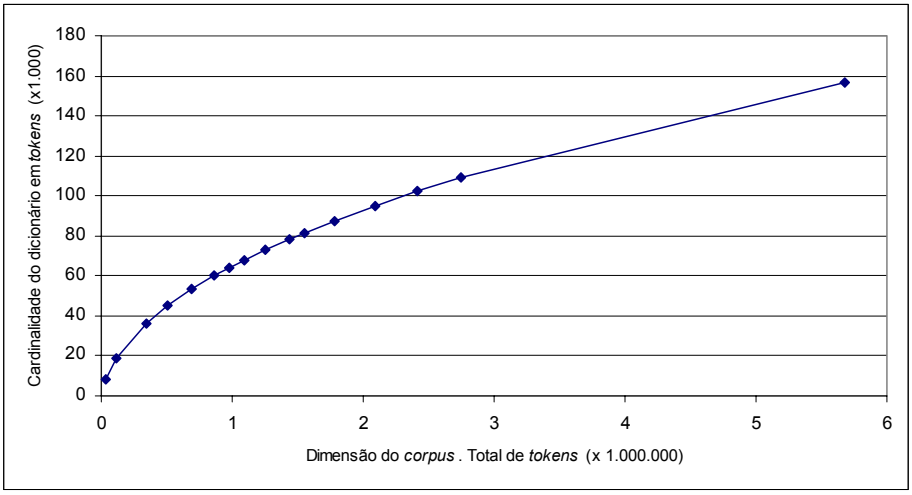


Figura 6.1 – Número de *tokens* diferentes versus número total de *tokens*

Os tempos foram medidos através de instruções de escrita da hora do sistema inseridas em pontos considerados relevantes do código, ou seja, os correspondentes às principais etapas de processamento: geração dos *tokens*, preparação da *matrix*, preenchimento da *matrix*, geração das *MWUs* e tradução final.

A memória consumida foi somente avaliada para o *Extractor* dado não ser relevante esta análise para os outros dois módulos. Foi estimada com base nas dimensões atingidas pelas estruturas principais constituintes do *Extractor* (i.e *corpus*, *matrix*, vector ☚ e *MNodes*), calculadas e exibidas por instruções inseridas no código para este efeito. Os valores estimados foram confirmados através das ferramentas de monitorização dos processos disponibilizadas pelos dois sistemas operativos utilizados (i.e. *Task Manager* em *Windows 2000* e *top* em *Linux*).

No ambiente *Windows 2000* foram avaliadas as prestações das quatro alternativas de implementação enunciadas na secção 5.3, ou seja, *Ideal V1*, *Ideal V2*, *Final V1* e *Final* V2. No final foi avaliado, o comportamento da última alternativa, *Final* V2, no segundo ambiente de testes (i.e *Linux* sobre uma máquina mais potente).

6.2 *CARACTERIZAÇÃO DOS CORPUS*

Apresentam-se em seguida, na Figura 6.2, as características dos 16 *corpus* testados, através dum conjunto de indicadores recolhidos durante os experimentos efectuados.

corpus	01	02	03	03A	04	05
Número de ficheiros	1	2	6	8	12	16
Dimensão em MB	0,2	0,7	2,1	3,1	4,2	5,3
Dimensão em *tokens*	33.095	114.373	342.734	506.259	685.274	864.790
Dimensão dicionário em *tokens*	8.359	18.454	36.009	44.879	53.141	60.199
Total *tokens* com *freq* > 1	27.661	103.641	323.315	482.832	657.968	834.373
Número total de *n-gramas*	1.423.085	4.918.039	14.737.562	21.769.137	29.466.782	37.185.970
Total de *n-gramas* não gerados	201	402	1.206	1.608	2.412	3.216
Total de *n-gramas* diferentes com *freq* > 1	14.469	66.969	257.481	412.200	596.410	792.569
Total de *n-gramas* com *freq* > 1	48.865	246.962	1.028.578	1.698.978	2.503.059	3.372.614
Total de *MWUs* identificadas	925	3.416	10.897	16.291	22.114	27.855

corpus	06	07	07A	08	09
Número de ficheiros	18	20	22	26	28
Dimensão em MB	6,0	6,7	7,7	8,8	9,5
Dimensão em *tokens*	978.647	1.092.723	1.256.006	1.435.930	1.550.435
Dimensão dicionário em *tokens*	64.367	68.051	73.010	78.413	81.603
Total *tokens* com *freq* > 1	946.341	1.058.720	1.219.898	1.397.309	1.510.445
Número total de *n-gramas*	42.081.821	46.987.089	54.008.258	61.744.990	66.668.705
Total de *n-gramas* não gerados	3.618	4.020	4.422	5.226	5.628
Total de *n-gramas* diferentes com *freq* > 1	914.166	1.041.794	1.226.988	1.432.348	1.571.512
Total de *n-gramas* com *freq* > 1	3.934.615	4.523.785	5.385.890	6.358.638	7.009.767
Total de *MWUs* identificadas	31.680	35.640	41.059	47.279	51.269

corpus	10	11	12	13	14
Número de ficheiros	32	37	41	45	57
Dimensão em MB	10,9	12,8	14,8	16,8	34,8
Dimensão em *tokens*	1.778.547	2.089.167	2.415.981	2.742.466	5.673.131
Dimensão dicionário em *tokens*	87.357	94.825	102.088	108.959	156.827
Total *tokens* com *freq* > 1	1.736.001	2.043.331	2.366.940	2.690.331	5.599.763
Número total de *n-gramas*	76.477.521	89.834.181	103.887.183	117.926.038	243.944.633
Total de *n-gramas* não gerados	6.432	7.437	8.241	9.045	11.457
Total de *n-gramas* diferentes com *freq* > 1	1.852.022	2.242.464	2.669.430	3.106.914	7.388.612
Total de *n-gramas* com *freq* > 1	8.339.362	10.210.569	12.254.108	14.370.587	35.579.420
Total de *MWUs* identificadas	59.048	69.767	80.852	92.295	192.923

Figura 6.2 – Características dos *corpus* testados

A explicação do significado de cada um dos indicadores é dada em seguida:

- *Número de ficheiros*: Número de ficheiros que compõem o *corpus*;
- *Dimensão do corpus em MB*: Soma da dimensão dos vários ficheiros do *corpus*;
- *Dimensão do corpus em tokens*: Total de *tokens* gerados pelo *Tokenizer*;
- *Dimensão do dicionário em tokens*: Total de *tokens* diferentes identificados pelo *Tokenizer* e inseridos no *dicionário*;
- *Total de tokens com freq > 1*: Total de *tokens* que ocorrem mais que uma vez no *corpus*;

- *Número total de n-gramas*: Total de *n-gramas*, correspondente, ao número de *máscaras* multiplicado pela dimensão do *corpus* em número de *tokens*. Para obter-se o total real de *n-gramas* gerados, deve ser subtraído o valor dos *n-gramas* não gerados para as posições finais de cada ficheiro, ou seja, o indicador a seguir enunciado;
- *Total de n-gramas não gerados*: Número de *n-gramas* não gerados para as últimas posições de cada ficheiro. Para *m* igual a 7, corresponde a multiplicar o número de ficheiros por 201;
- *Total de n-gramas diferentes com freq > 1*: Número de *n-gramas* (com número de *tokens* maior ou igual a dois) diferentes com *frequência* não unitária, ou seja, que ocorrem mais que uma vez no *corpus*;
- *Total de n-gramas com freq > 1*: Total de *n-gramas* com *frequência* não unitária;
- *Total de MWUs identificadas*: Número de *MWUs* identificadas.

6.3 CONSUMOS DE MEMÓRIA

Os valores recolhidos para as várias características dos *corpus* testados permitem-nos avançar para a estimativa dos consumos de memória associados a cada uma das alternativas de implementação. Esta estimativa foi confirmada, em cada teste, através da leitura do consumo de memória atribuído a cada processo, dado pelas ferramentas de sistema já mencionadas.

Para o cálculo do consumo de memória foram utilizadas as dimensões, em *bytes*, dos objectos criados com base nas classes mais elementares envolvidas na construção do *Extractor*. Estes valores estão resumidos no quadro da Figura 6.3.

Classe	Ideal	Final
Pos	8	8
Mask	24	24
MNode	8	12
NGram	12	16
Vnode	n/a	8

Figura 6.3 – Dimensão, em *bytes*, das classes elementares

De realçar que a dimensão de cada uma destas classes não é igual nas duas alternativas de implementação testadas devido às opções tomadas ao nível do cálculo prévio e armazenamento da *ME*. Nas versões dentro do grupo *Ideal* a *ME* não é armazenada. Nas versões do grupo *Final* a *ME* é calculada antes da execução do *GenLocalMaxs* e armazenada, para todos os *n-gramas* com *frequência* não unitária. Com base na dimensão e no número observado de objectos elementares criados, foi possível estimar o consumo de memória para os objectos criados a partir das classes mais

consumidoras de memória (i.e. *corpus*, *vector* ☚, *matrix* e *Mmodes*) e assim estimar o consumo total de memória do sistema para cada uma das alternativas implementadas.

Como já foi referido, o consumo real de memória foi aferido com base na leitura dos valores apresentados pelas ferramentas dos sistemas operativos para cada um dos processos em execução. Estes valores apresentam uma variação de cerca de 10% em relação aos valores estimados, correspondente à memória necessária para alojar o próprio programa, a pilha de execução[55] e os objectos criados a partir das restantes classes que compõem o sistema.

corpus	01	02	03	3A	04	05	06	07	7A	08	09	10	11	12	13	14
Ideal V1																
Corpus	0,1	0,4	1,3	1,9	2,6	3,3	3,7	4,2	4,8	5,5	5,9	6,8	8,0	9,2	10,5	21,6
Vector Delta	0,5	1,7	5,2	7,7	10,5	13,2	14,9	16,7	19,2	21,9	23,7	27,1	31,9	36,9	41,8	86,6
Matrix	5,4	18,8	56,2	83,0	112,4	141,9	160,5	179,2	206,0	235,5	254,3	291,7	342,7	396,3	449,9	930,6
MNodes	0,7	3,2	12,4	20,0	28,9	38,4	44,5	50,8	60,0	70,4	77,3	91,7	111,3	132,7	154,7	371,7
TOTAL	**6,8**	**24,2**	**75,2**	**112,7**	**154,3**	**196,7**	**223,7**	**250,9**	**290,0**	**333,3**	**361,2**	**417,3**	**493,9**	**575,1**	**656,9**	**1.410,5**
Ideal V2																
Corpus	0,1	0,4	1,3	1,9	2,6	3,3	3,7	4,2	4,8	5,5	5,9	6,8	8,0	9,2	10,5	21,6
Vector Delta	0,5	1,7	5,2	7,7	10,5	13,2	14,9	16,7	19,2	21,9	23,7	27,1	31,9	36,9	41,8	86,6
Matrix	5,4	18,8	56,2	83,0	112,4	141,9	160,5	179,2	206,0	235,5	254,3	291,7	342,7	396,3	449,9	930,6
MNodes	0,5	2,4	9,8	16,1	23,6	31,8	37,0	42,7	50,5	59,4	65,5	77,8	95,0	113,9	133,3	327,8
TOTAL	**6,5**	**23,3**	**72,6**	**108,8**	**149,1**	**190,1**	**216,2**	**242,8**	**280,4**	**322,4**	**349,4**	**403,4**	**477,5**	**556,2**	**635,5**	**1.366,6**
Final V1																
Corpus	0,1	0,4	1,3	1,9	2,6	3,3	3,7	4,2	4,8	5,5	5,9	6,8	8,0	9,2	10,5	21,6
Vector Delta	0,6	2,2	6,5	9,7	13,1	16,5	18,7	20,8	24,0	27,4	29,6	33,9	39,8	46,1	52,3	108,2
Matrix	0,6	2,7	10,3	16,6	24,1	32,1	37,2	42,6	50,4	59,2	65,0	76,9	93,5	111,6	130,2	314,2
MNodes	0,8	3,5	13,5	21,6	31,2	41,5	48,1	54,9	64,9	76,0	83,5	98,6	119,7	142,7	166,4	400,5
TOTAL	**2,1**	**8,8**	**31,6**	**49,8**	**71,0**	**93,4**	**107,7**	**122,5**	**144,0**	**168,1**	**184,0**	**216,2**	**261,0**	**309,6**	**359,3**	**844,6**
Final V2																
Corpus	0,1	0,4	1,3	1,9	2,6	3,3	3,7	4,2	4,8	5,5	5,9	6,8	8,0	9,2	10,5	21,6
Vector Delta	0,6	2,2	6,5	9,7	13,1	16,5	18,7	20,8	24,0	27,4	29,6	33,9	39,8	46,1	52,3	108,2
Matrix	0,6	2,7	10,3	16,6	24,1	32,1	37,2	42,6	50,4	59,2	65,0	76,9	93,5	111,6	130,2	314,2
MNodes	0,5	2,7	10,8	17,7	25,9	34,8	40,5	46,4	55,1	64,9	71,5	84,8	103,6	124,0	145,2	356,0
TOTAL	**1,9**	**7,9**	**29,0**	**45,9**	**65,7**	**86,7**	**100,1**	**114,0**	**134,3**	**156,9**	**172,0**	**202,4**	**244,9**	**290,9**	**338,1**	**800,0**

Figura 6.4 – Consumo de memória (MB) por classe

[55] *Stack* na literatura inglesa.

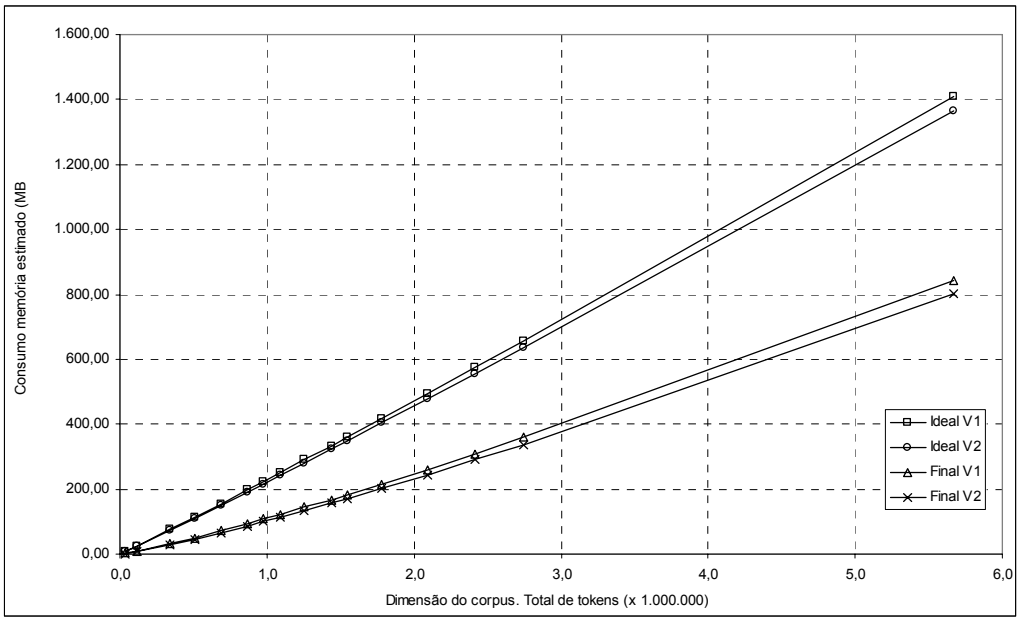


Figura 6.5 – Consumo de memória (MB) total

Na Figura 6.4 é mostrado o quadro de consumos para os diferentes *corpus*. As barras duplas verticais mostram o ponto em que o limite de memória do primeiro sistema de testes (i.e. 390MB) é atingido, ou seja, a partir do qual começam-se a fazer sentir os efeitos do *swapping* de páginas de memória, como fica evidente nos resultados relativos aos tempos de execução mostrados, mais adiante, na secção 6.4.

O consumo total de memória é mostrado, graficamente, na Figura 6.5, onde são evidentes as diferenças entre as várias alternativas de implementação testadas.

Da leitura dos números anteriores é especialmente importante verificar-se o peso de cada estrutura no valor total de memória consumido para cada umas das alternativas. É este balanceamento de pesos que se procurou mostrar no quadro da Figura 6.6, onde são mostradas as percentagens da memória total consumida atribuída a cada uma das classes principais.

corpus	01	02	03	3A	04	05	06	07	7A	08	09	10	11	12	13	14
Ideal V1																
Corpus	1,9%	1,8%	1,7%	1,7%	1,7%	1,7%	1,7%	1,7%	1,7%	1,6%	1,6%	1,6%	1,6%	1,6%	1,6%	1,5%
Vector Delta	7,5%	7,2%	7,0%	6,9%	6,8%	6,7%	6,7%	6,6%	6,6%	6,6%	6,5%	6,5%	6,5%	6,4%	6,4%	6,1%
Matrix	80,1%	77,6%	74,8%	73,7%	72,8%	72,1%	71,8%	71,4%	71,0%	70,7%	70,4%	69,9%	69,4%	68,9%	68,5%	66,0%
MNodes	10,6%	13,4%	16,5%	17,7%	18,7%	19,5%	19,9%	20,2%	20,7%	21,1%	21,4%	22,0%	22,5%	23,1%	23,6%	26,4%
Ideal V2																
Corpus	1,9%	1,9%	1,8%	1,8%	1,8%	1,7%	1,7%	1,7%	1,7%	1,7%	1,7%	1,7%	1,7%	1,7%	1,6%	1,6%
Vector Delta	7,7%	7,5%	7,2%	7,1%	7,0%	6,9%	6,9%	6,9%	6,8%	6,8%	6,8%	6,7%	6,7%	6,6%	6,6%	6,3%
Matrix	83,0%	80,4%	77,5%	76,3%	75,4%	74,6%	74,3%	73,8%	73,5%	73,1%	72,8%	72,3%	71,8%	71,2%	70,8%	68,1%
MNodes	7,4%	10,3%	13,5%	14,8%	15,9%	16,7%	17,1%	17,6%	18,0%	18,4%	18,7%	19,3%	19,9%	20,5%	21,0%	24,0%
Final V1																
Corpus	5,9%	4,9%	4,1%	3,9%	3,7%	3,5%	3,5%	3,4%	3,3%	3,3%	3,2%	3,1%	3,1%	3,0%	2,9%	2,6%
Vector Delta	29,7%	24,7%	20,7%	19,4%	18,4%	17,7%	17,3%	17,0%	16,6%	16,3%	16,1%	15,7%	15,3%	14,9%	14,6%	12,8%
Matrix	27,5%	30,3%	32,6%	33,4%	34,0%	34,4%	34,6%	34,8%	35,0%	35,2%	35,3%	35,6%	35,8%	36,0%	36,2%	37,2%
MNodes	36,8%	40,0%	42,6%	43,4%	44,0%	44,4%	44,6%	44,8%	45,0%	45,2%	45,4%	45,6%	45,9%	46,1%	46,3%	47,4%
Final V2																
Corpus	6,7%	5,5%	4,5%	4,2%	4,0%	3,8%	3,7%	3,7%	3,6%	3,5%	3,4%	3,4%	3,3%	3,2%	3,1%	2,7%
Vector Delta	33,6%	27,5%	22,6%	21,0%	19,9%	19,0%	18,6%	18,3%	17,8%	17,5%	17,2%	16,8%	16,3%	15,8%	15,5%	13,5%
Matrix	31,1%	33,7%	35,6%	36,3%	36,7%	37,0%	37,2%	37,4%	37,5%	37,7%	37,8%	38,0%	38,2%	38,3%	38,5%	39,3%
MNodes	28,6%	33,4%	37,3%	38,5%	39,4%	40,1%	40,4%	40,7%	41,1%	41,4%	41,6%	41,9%	42,3%	42,6%	42,9%	44,5%

Figura 6.6 – Percentagens de consumo de memória por classe

Da leitura dos quadros anteriores pensamos ser importante realçar os seguintes pontos:

- A linearidade da curva de consumo total de memória, em qualquer das versões, com o incremento da dimensão do *corpus*, confirmando a hipótese teórica de crescimento linear em *N*;
- O peso excessivo da *matrix* nas duas primeiras versões do sistema. Corresponde a 70 a 80% da memória consumida.
- A diferença, significativa, no total de memória necessária entre a implementação totalmente matricial da *matrix* (i.e. *Ideal*) e as versões baseadas num vector de apontadores para vectores (i.e. *Final*). Esta diferença é tanto maior quanto maior for o *corpus* e justifica claramente o investimento efectuado na procura duma solução alternativa, mais económica em termos de espaço de memória;
- O peso insignificante da opção de leitura da totalidade do *corpus* para memória;
- A confirmação que o número de *n-gramas* com frequência não unitária é uma percentagem pequena do total de *n-gramas* envolvidos no processamento, confirmando os dados avançados na secção 4.4;

6.4 TEMPOS DE PROCESSAMENTO

O sistema foi testado sobre as duas máquinas e ambientes atrás caracterizados e para as diferentes dimensões do *corpus*.

corpus	01	02	03	3A	04	05	06	07	7A	08	09	10	11	12	13
Ideal V1 (Windows)															
Tokenizador	0:00:00	0:00:01	0:00:04	0:00:04	0:00:07	0:00:08	0:00:07	0:00:09	0:00:15	0:00:17	0:00:14				
Extractor	0:00:11	0:00:45	0:02:40	0:04:11	0:05:55	0:07:44	0:08:53	0:10:05	0:11:52	0:18:06	1:27:10				
Preparação	0:00:01	0:00:04	0:00:10	0:00:14	0:00:19	0:00:25	0:00:27	0:00:31	0:00:36	0:00:40	0:00:52				
Criação *matrix*	0:00:08	0:00:35	0:02:09	0:03:23	0:04:49	0:06:16	0:07:13	0:08:11	0:09:36	0:11:12	0:13:02				
Extracção *MWUs*	0:00:02	0:00:06	0:00:21	0:00:34	0:00:47	0:01:03	0:01:13	0:01:23	0:01:40	0:06:14	1:13:16				
Translactor	0:00:00	0:00:01	0:00:01	0:00:02	0:00:02	0:00:02	0:00:02	0:00:05	0:00:05	0:00:10	0:00:05				
TOTAL	**0:00:11**	**0:00:47**	**0:02:45**	**0:04:17**	**0:06:04**	**0:07:54**	**0:09:02**	**0:10:19**	**0:12:12**	**0:18:33**	**1:27:29**				
Ideal V2 (Windows)															
Tokenizador	0:00:01	0:00:01	0:00:03	0:00:05	0:00:09	0:00:09	0:00:08	0:00:11	0:00:14	0:00:17	0:00:13				
Extractor	0:00:11	0:00:45	0:02:40	0:04:12	0:05:57	0:07:44	0:08:55	0:10:08	0:11:51	0:16:28	1:45:24				
Preparação	0:00:01	0:00:03	0:00:10	0:00:14	0:00:20	0:00:24	0:00:29	0:00:32	0:00:36	0:00:41	0:00:50				
Criação *matrix*	0:00:08	0:00:36	0:02:09	0:03:23	0:04:48	0:06:16	0:07:12	0:08:11	0:09:35	0:11:10	0:12:40				
Extracção *MWUs*	0:00:02	0:00:06	0:00:21	0:00:35	0:00:49	0:01:04	0:01:14	0:01:25	0:01:40	0:04:37	1:31:54				
Translactor	0:00:00	0:00:00	0:00:01	0:00:01	0:00:02	0:00:02	0:00:03	0:00:04	0:00:06	0:00:09	0:00:06				
TOTAL	**0:00:12**	**0:00:46**	**0:02:44**	**0:04:18**	**0:06:08**	**0:07:55**	**0:09:06**	**0:10:23**	**0:12:11**	**0:16:54**	**1:45:43**				
Final V1 (Windows)															
Tokenizador	0:00:01	0:00:02	0:00:05	0:00:04	0:00:07	0:00:07	0:00:10	0:00:09	0:00:10	0:00:11	0:00:10	0:00:20	0:00:13	0:00:22	0:00:28
Extractor	0:00:11	0:00:49	0:03:03	0:04:53	0:07:00	0:09:14	0:10:43	0:12:15	0:14:27	0:16:57	0:18:31	0:21:52	0:26:32	0:31:52	0:54:40
Preparação	0:00:01	0:00:02	0:00:04	0:00:06	0:00:08	0:00:10	0:00:12	0:00:14	0:00:15	0:00:18	0:00:19	0:00:22	0:00:25	0:00:30	0:00:42
Criação *matrix*	0:00:08	0:00:35	0:02:08	0:03:22	0:04:47	0:06:15	0:07:11	0:08:11	0:09:36	0:11:11	0:12:10	0:14:16	0:17:10	0:20:17	0:29:05
Calculo *MEs*	0:00:00	0:00:01	0:00:02	0:00:03	0:00:04	0:00:05	0:00:07	0:00:07	0:00:09	0:00:11	0:00:12	0:00:14	0:00:18	0:00:23	0:03:45
Extracção *MWUs*	0:00:02	0:00:11	0:00:49	0:01:22	0:02:01	0:02:44	0:03:13	0:03:43	0:04:27	0:05:17	0:05:50	0:07:00	0:08:39	0:10:42	0:21:08
Translactor	0:00:00	0:00:00	0:00:01	0:00:01	0:00:02	0:00:02	0:00:02	0:00:03	0:00:03	0:00:03	0:00:04	0:00:05	0:00:06	0:00:09	0:00:09
TOTAL	**0:00:12**	**0:00:51**	**0:03:09**	**0:04:58**	**0:07:09**	**0:09:23**	**0:10:55**	**0:12:27**	**0:14:40**	**0:17:11**	**0:18:45**	**0:22:17**	**0:26:51**	**0:32:23**	**0:55:17**
Final V2 (Windows)															
Tokenizador	0:00:01	0:00:01	0:00:03	0:00:05	0:00:06	0:00:06	0:00:10	0:00:10	0:00:12	0:00:12	0:00:14	0:00:15	0:00:19	0:00:21	0:00:25
Extractor	0:00:11	0:00:50	0:03:05	0:04:56	0:07:02	0:09:18	0:10:45	0:12:16	0:14:31	0:17:01	0:18:39	0:22:03	0:26:42	0:33:06	0:54:56
Preparação	0:00:00	0:00:02	0:00:04	0:00:06	0:00:08	0:00:11	0:00:12	0:00:13	0:00:15	0:00:17	0:00:19	0:00:22	0:00:25	0:00:30	0:00:43
Criação *matrix*	0:00:09	0:00:36	0:02:10	0:03:25	0:04:49	0:06:18	0:07:15	0:08:14	0:09:41	0:11:17	0:12:17	0:14:27	0:17:20	0:20:30	0:29:26
Calculo *MEs*	0:00:00	0:00:00	0:00:02	0:00:03	0:00:05	0:00:06	0:00:07	0:00:08	0:00:09	0:00:10	0:00:12	0:00:14	0:00:17	0:01:30	0:03:26
Extracção *MWUs*	0:00:02	0:00:12	0:00:49	0:01:22	0:02:00	0:02:43	0:03:11	0:03:41	0:04:26	0:05:17	0:05:51	0:07:00	0:08:40	0:10:36	0:21:21
Translactor	0:00:01	0:00:00	0:00:01	0:00:02	0:00:02	0:00:02	0:00:03	0:00:03	0:00:03	0:00:04	0:00:03	0:00:04	0:00:05	0:00:07	0:00:09
TOTAL	**0:00:13**	**0:00:51**	**0:03:09**	**0:05:03**	**0:07:10**	**0:09:26**	**0:10:58**	**0:12:29**	**0:14:46**	**0:17:17**	**0:18:56**	**0:22:22**	**0:27:06**	**0:33:34**	**0:55:30**

Figura 6.7 – Tempos de processamento para o primeiro ambiente de testes

Os resultados obtidos para o primeiro ambiente de testes (i.e. Pentium III, 390MB, Windows 2000) estão resumidos no quadro da Figura 6.7 e graficamente na Figura 6.8. Os últimos tempos medidos, para cada uma das versões testadas, correspondem ao limite de memória do sistema a partir do qual se começa a sentir a degradação de tempos provocada pelo *swapping* de páginas de memória para disco. Para lá deste limite, como é visível no gráfico, a curva dos tempos tem uma inflexão acentuada correspondente a tempos de processamento cada vez maiores. Esta é a razão das células em branco no lado direito do quadro para algumas das linhas.

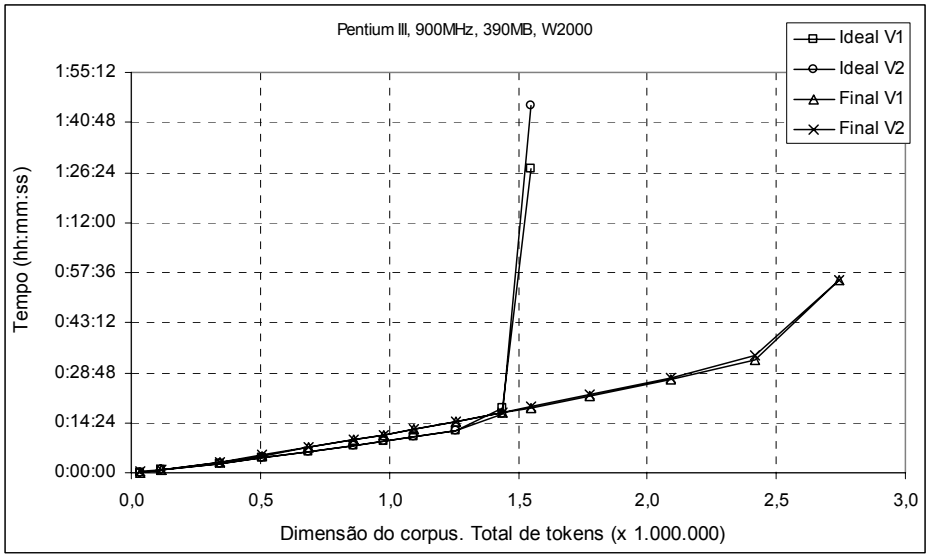


Figura 6.8 – Tempos totais de processamento para o primeiro ambiente de testes

No quadro da Figura 6.7 além dos valores totais são mostrados valores parciais para as etapas principais em que se subdivide o processo de extracção de *MWUs*, a saber: *tokenização*, preparação das *máscaras*, leitura do *corpus* e preparação do vector ☚, preenchimento da *matrix*, extracção das *MWUs* e tradução final. Para as versões *Final V1* e *Final V2* foi ainda medida a etapa correspondente ao cálculo e armazenamento das *MEs* para todos os *n-gramas* com *frequência* não unitária.

O sistema foi recompilado e testado no segundo ambiente de testes, optando-se neste caso por recolher unicamente os tempos para a versão mais económica, em termos de consumo de memória, das várias testadas (i.e. *Final V2*). Estes tempos são mostrados na Figura 6.9.

corpus	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**10**	**11**	**12**	**13**	**14**
Final V2 (Linux)														
Tokenizador	0:00:00	0:00:00	0:00:01	0:00:03	0:00:04	0:00:03	0:00:04	0:00:05	0:00:06	0:00:06	0:00:07	0:00:08	0:00:10	0:00:20
Extractor	0:00:09	0:00:42	0:02:32	0:06:14	0:08:16	0:09:18	0:10:52	0:15:26	0:16:54	0:20:13	0:24:17	0:28:27	0:31:06	1:24:07
Preparação	0:00:00	0:00:01	0:00:03	0:00:06	0:00:08	0:00:10	0:00:11	0:00:15	0:00:16	0:00:19	0:00:22	0:00:25	0:00:24	0:01:03
Criação *matrix*	0:00:07	0:00:30	0:01:48	0:04:06	0:05:23	0:05:57	0:06:55	0:09:40	0:10:35	0:12:30	0:14:56	0:17:41	0:18:17	0:45:54
Calculo *MEs*	0:00:00	0:00:00	0:00:01	0:00:04	0:00:05	0:00:07	0:00:08	0:00:11	0:00:13	0:00:16	0:00:18	0:00:23	0:00:22	0:02:02
Extracção *MWUs*	0:00:02	0:00:11	0:00:40	0:01:58	0:02:40	0:03:04	0:03:38	0:05:20	0:05:50	0:07:08	0:08:41	0:09:58	0:12:03	0:35:08
Translactor	0:00:00	0:00:00	0:00:00	0:00:00	0:00:01	0:00:01	0:00:01	0:00:01	0:00:01	0:00:01	0:00:02	0:00:01	0:00:02	0:00:04
TOTAL	**0:00:09**	**0:00:42**	**0:02:33**	**0:06:17**	**0:08:21**	**0:09:22**	**0:10:57**	**0:15:32**	**0:17:01**	**0:20:20**	**0:24:26**	**0:28:36**	**0:31:18**	**1:24:31**

Figura 6.9 – Tempos de processamento para o segundo ambiente de testes

O gráfico da Figura 6.10 permite avaliar da diferença de tempos de processamento nos dois ambientes de teste para a versão *Final V2*. Como seria de esperar a segunda máquina, um *dual Pentium*, é mais rápida. No entanto a diferença de tempos é menor do que seria previsível devido, supostamente, ao

facto do código não estar preparado para tirar partido da paralelização do processamento. É de realçar a capacidade da segunda máquina, dada a maior disponibilidade de memória, para processar *corpus* de muito maior dimensão que a primeira.

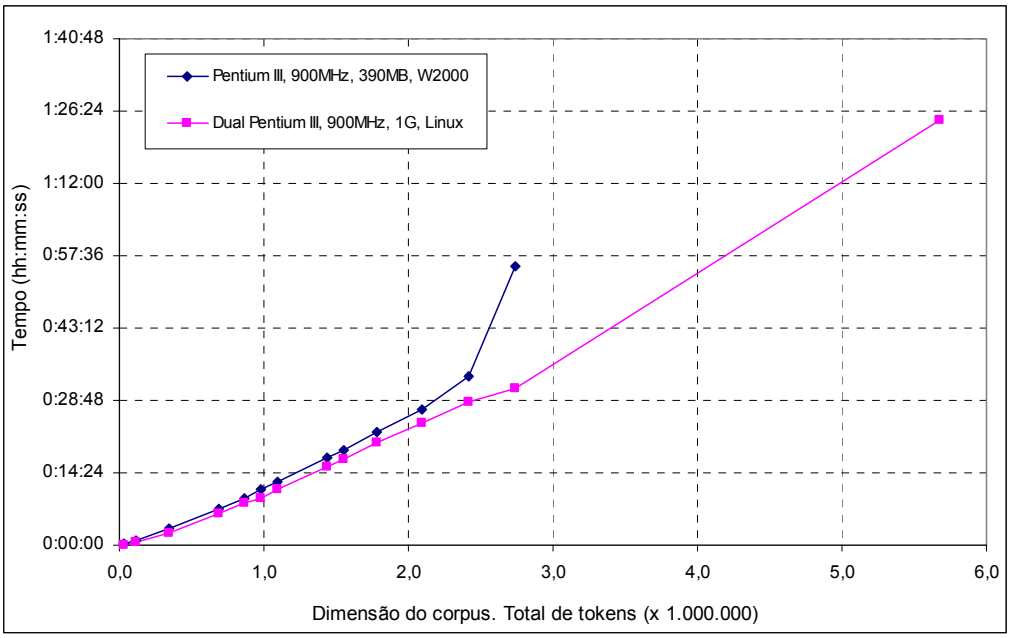


Figura 6.10 – Comparativo entre os dois ambientes de teste para *Final V2*

Igualmente importante, é avaliar onde é consumido o tempo, ou seja, qual a percentagem do tempo total atribuída a cada uma das fases de processamento.

corpus	01	02	03	3A	04	05	06	07	7A	08	09
Ideal V1 (Windows)											
Tokenizador	0,0%	2,1%	2,4%	1,6%	1,9%	1,7%	1,3%	1,5%	2,0%	1,5%	0,3%
Extractor	100,0%	95,7%	97,0%	97,7%	97,5%	97,9%	98,3%	97,7%	97,3%	97,6%	99,6%
Preparação	9,1%	8,5%	6,1%	5,4%	5,2%	5,3%	5,0%	5,0%	4,9%	3,6%	1,0%
Criação *matrix*	72,7%	74,5%	78,2%	79,0%	79,4%	79,3%	79,9%	79,3%	78,7%	**60,4%**	**14,9%**
Extracção *MWUs*	18,2%	12,8%	12,7%	13,2%	12,9%	13,3%	13,5%	13,4%	13,7%	**33,6%**	**83,7%**
Translactor	0,0%	2,1%	0,6%	0,8%	0,5%	0,4%	0,4%	0,8%	0,7%	0,9%	0,1%
Ideal V2 (Windows)											
Tokenizador	8,3%	2,2%	1,8%	1,9%	2,4%	1,9%	1,5%	1,8%	1,9%	1,7%	0,2%
Extractor	91,7%	97,8%	97,6%	97,7%	97,0%	97,7%	98,0%	97,6%	97,3%	97,4%	99,7%
Preparação	8,3%	6,5%	6,1%	5,4%	5,4%	5,1%	5,3%	5,1%	4,9%	4,0%	0,8%
Criação *matrix*	66,7%	78,3%	78,7%	78,7%	78,3%	79,2%	79,1%	78,8%	78,7%	**66,1%**	**12,0%**
Extracção *MWUs*	16,7%	13,0%	12,8%	13,6%	13,3%	13,5%	13,6%	13,6%	13,7%	**27,3%**	**86,9%**
Translactor	0,0%	0,0%	0,6%	0,4%	0,5%	0,4%	0,5%	0,6%	0,8%	0,9%	0,1%

Figura 6.11 – Percentagem de tempo atribuída a cada etapa para versões *Ideal V1 e V2*

Na Figura 6.11 é mostrada a percentagem de tempo atribuída a cada fase para as versões *Ideal V1* e *Ideal V2* onde é perfeitamente visível a pouca importância, no tempo global, das etapas respeitantes à

tokenização, preparação das *máscaras* e do *vector* ☚ e à tradução final. A maioria do tempo é despendido no preenchimento da *matrix*. De realçar o agravamento brutal do tempo necessário para a função de extracção das *MWUs* para *corpus* superiores à capacidade máxima de memória disponível, resultante da activação do *swapping* de páginas de memória por parte do sistema operativo. O peso desta última etapa é crescente com o aumento da dimensão do *corpus*.

Na Figura 6.12 é efectuada uma análise idêntica para as versões *Final V1* e *Final V2*. Para estas duas versões é de realçar o menor peso da etapa respeitante à criação da *matrix* com o correspondente agravar da importância da etapa de extracção das *MWUs*. De salientar, igualmente, o crescente peso da etapa correspondente ao cálculo da *ME*.

corpus	01	02	03	3A	04	05	06	07	7A	08	09	10	11	12	13
Final V1 (Windows)															
Tokenizador	8,3%	3,9%	2,6%	1,3%	1,6%	1,2%	1,5%	1,2%	1,1%	1,1%	0,9%	1,5%	0,8%	1,1%	0,8%
Extractor	91,7%	96,1%	96,8%	98,3%	97,9%	98,4%	98,2%	98,4%	98,5%	98,6%	98,8%	98,1%	98,8%	98,4%	98,9%
Preparação	8,3%	3,9%	2,1%	2,0%	1,9%	1,8%	1,8%	1,9%	1,7%	1,7%	1,7%	1,6%	1,6%	1,5%	1,3%
Criação *matrix*	66,7%	68,6%	67,7%	67,8%	66,9%	66,6%	65,8%	65,7%	65,5%	65,1%	64,9%	64,0%	63,9%	62,6%	52,6%
Calculo *MEs*	0,0%	2,0%	1,1%	1,0%	0,9%	0,9%	1,1%	0,9%	1,0%	1,1%	1,1%	1,0%	1,1%	1,2%	6,8%
Extracção *MWUs*	16,7%	21,6%	25,9%	27,5%	28,2%	29,1%	29,5%	29,9%	30,3%	30,7%	31,1%	31,4%	32,2%	33,0%	38,2%
Translactor	0,0%	0,0%	0,5%	0,3%	0,5%	0,4%	0,3%	0,4%	0,3%	0,3%	0,4%	0,4%	0,4%	0,5%	0,3%
Final V2 (Windows)															
Tokenizador	7,7%	2,0%	1,6%	1,7%	1,4%	1,1%	1,5%	1,3%	1,4%	1,2%	1,2%	1,1%	1,2%	1,0%	0,8%
Extractor	84,6%	98,0%	97,9%	97,7%	98,1%	98,6%	98,0%	98,3%	98,3%	98,5%	98,5%	98,6%	98,5%	98,6%	99,0%
Preparação	0,0%	3,9%	2,1%	2,0%	1,9%	1,9%	1,8%	1,7%	1,7%	1,6%	1,7%	1,6%	1,5%	1,5%	1,3%
Criação *matrix*	69,2%	70,6%	68,8%	67,7%	67,2%	66,8%	66,1%	66,0%	65,6%	65,3%	64,9%	64,6%	64,0%	61,1%	53,0%
Calculo *MEs*	0,0%	0,0%	1,1%	1,0%	1,2%	1,1%	1,1%	1,1%	1,0%	1,0%	1,1%	1,0%	1,0%	4,5%	6,2%
Extracção *MWUs*	15,4%	23,5%	25,9%	27,1%	27,9%	28,8%	29,0%	29,5%	30,0%	30,6%	30,9%	31,3%	32,0%	31,6%	38,5%
Translactor	7,7%	0,0%	0,5%	0,7%	0,5%	0,4%	0,5%	0,4%	0,3%	0,4%	0,3%	0,3%	0,3%	0,3%	0,3%

Figura 6.12 – Percentagem de tempo atribuída a cada etapa para versões *Final V1 e V2*

Foi ainda comparada a eficiência da versão *Final V2* em relação ao SENTA no processamento dum mesmo *corpus* no segundo ambiente de testes. As experiências efectuadas mostraram que o SENTA é muito pouco eficiente quando comparado com a nova implementação proposta. Necessita cerca de 10 minutos para processar um *corpus* com 32718 *tokens*, enquanto que para a nova implementação bastam 10 segundos.

6.5 CORRECÇÃO DOS RESULTADOS

A análise de correcção dos resultados, ou seja, a constatação que a geração do conjunto de *MWUs* é correcta, foi efectuada recorrendo a dois métodos complementares:

1. A reprodução manual, para *n-gramas* escolhidos aleatoriamente, dos algoritmos e comparação com os resultados obtidos via protótipo;
2. A comparação dos resultados do processamento de *corpus*, em simultâneo, na implementação proposta e no SENTA;

A conjugação destes dois métodos permitiu concluir que o processamento efectuado é o correcto, ou seja, que a implementação proposta respeita os algoritmos de cálculo da *ME* e o *GenLocalMaxs*.

As experiências efectuadas permitiram concluir que o SENTA, para obter reduções nos tempos de processamento, não aplica de forma rigorosa o *GenLocalMaxs*. Ignora todos os *n-gramas* com *frequência* unitária, excepto para a eleição dos *bi-gramas* e ignora as posições finais dos ficheiros. Apresenta ainda alguns problemas no tratamento de *corpus* pequenos devido a deficiências nas ferramentas de sistema utilizadas (e.g. função *split*).

Pelas razões apresentadas no parágrafo anterior, os resultados obtidos através da implementação proposta diferem dos obtidos via SENTA, tal como ilustrado no exemplo apresentado na Figura 6.13.

Total *tokens* = 32718	SENTA	Novo sistema
2-gramas	181	187
3-gramas	618	397
4-gramas	196	212
5-gramas	28	46
6-gramas	32	7
Total	**1055**	**819**

Figura 6.13 – Comparação de resultados com o SENTA

A análise das resultados obtidos para o exemplo mostrado na Figura 6.13, permitiu encontrar as razões por detrás das diferenças detectadas e concluir da correcção da nova implementação.

6.6 ANÁLISE GLOBAL DOS RESULTADOS

Os resultados apresentados em termos da eficiência espacial e temporal das várias alternativas implementadas mostram claramente que as opções tomadas foram as correctas em termos da evolução efectuada para a estrutura de dados e para os algoritmos ao longo dos sucessivos refinamentos da solução. A versão mais elaborada, *Final V2*, apresenta níveis de eficiência espacial muitos superiores às primeiras versões, cerca de 50% menos espaço, sem que tal represente um agravamento significativo da sua eficiência temporal, 15 a 20% mais demorado. Esta última versão permite, sem alterar as condições base da plataforma *hardware*, processar, em memória e sem que haja *swapping*, *corpus* de dimensão muito superior. Na primeira plataforma de testes, o limite passa de 1,4 milhões de tokens para 2,7 milhões de acordo com os pontos de inflexão no gráfico da Figura 6.8.

Os dados recolhidos permitem concluir que algoritmos mais complexos não implicam necessariamente tempos de resposta muito piores desde que sejam compensados com outras medidas de redução do tamanho dos vários ciclos e consequentemente por uma redução do número de operações efectuadas. Este facto é visível quando se comparam as versões *Ideal* e *Final*, nomeadamente no impacto positivo na eficiência temporal, resultante do cálculo prévio e armazenamento do valor da *ME* para todos os *n-*

gramas com frequência não unitária, nas versões *Final*, e o impacto, igualmente positivo, da redução do ciclo de busca das *MWUs* resultante da implementação da *matrix*, nas versões *Final*, excluindo, logo à partida, todos os *n-gramas* unitários, sem potencial para serem *MWUs* e assim reduzir o espaço de busca em relação às versões *Ideal.*

As versões *V2* apresentam uma ligeira redução de espaço consumido em relação às versões *V1* como seria previsível. Esta redução tem como reflexo um ligeiro impacto negativo na eficiência temporal das soluções, levando a concluir que qualquer das opções é válida sendo preferível a segunda quando o espaço for um factor totalmente crítico, ou quando o *N* for muito elevado.

O agravamento, com o crescimento da dimensão do *corpus*, do peso da etapa correspondente à identificação das *MWUs* é consequência do peso cada vez maior do cálculo da *ME* dos *n-gramas* que não tenham este valor previamente armazenado, pelo peso das operações de escrita em ficheiro do número, cada vez maior, de *MWUs* e pelo aumento do número de *n-gramas* visitados. Este último factor prende-se igualmente com o agravar do tempo necessário para desmarcar, marcar e testar os *MNodes*, por forma a evitarem-se repetições.

Verificou-se, igualmente, nas experiências efectuadas, que as operações finais de libertação de memória têm um peso cada vez mais importante com o crescimento do *N*, não podendo ser desprezadas para *corpus* de dimensão elevada. Este é um factor não contabilizado à partida mas que tem influência nos valores finais.

De realçar igualmente, para os valores de *N* testados, a quase linearidade, tanto espacial como temporal, de qualquer das alternativas testadas, comprovando, as hipóteses teóricas levantadas em termos de complexidades na secção 4.7.

7 CONCLUSÕES E PERSPECTIVAS DE TRABALHO FUTURO

Uma análise cuidada dos detalhes do método de extracção de *MWUs* proposto em (Dias *et al*.1999a) e das suas propriedades, permitiu o desenvolvimento duma solução muito eficiente, tanto do ponto de vista espacial como temporal. A solução está baseada na proposta de adopção duma nova representação para os *n-gramas*, baseada num par de referências na forma {*pos*, *mask*}, onde o primeiro termo representa uma posição do *corpus* e o segundo o identificador duma *máscara*. Esta nova representação permite subir para um grau de abstracção superior, até então não alcançado, no que respeita ao tratamento de *n-gramas*. A aplicação desta nova representação e dum conjunto de propriedades da *frequência*, da *ME* e do algoritmo *GenLocalMaxs*, quando utilizado em conjunto com a *ME*, permitiu a construção duma solução muito eficiente, respondendo à totalidade do enunciado do problema proposto. A estrutura de dados e algoritmos propostos dão resposta aos três principais problemas subjacentes à extracção de *MWUs* dum *corpus* através do método proposto em (Dias *et al*.1999a), a saber: cálculo eficiente da *frequência*, cálculo eficiente da *ME* e geração eficiente do conjunto de *MWUs* através do algoritmo *GenLocalMaxs*.

De realçar que a eficiência da solução é alcançada através duma estrutura, o vector de *máscaras*, que necessita de espaço e de tempo de processamento inicial para a sua criação, totalmente desprezíveis. O caminho para esta solução foi encetado através da tentativa de estender a *n-gramas* não contíguos o trabalho desenvolvido por Yamamoto *et al*. (2000) em torno duma estrutura, baseada num *suffix-array*, para obtenção eficiente da *frequência* de qualquer *string* dum texto. A solução que propomos, apesar de ter este ponto de contacto inicial, aponta para caminhos algo diferentes.

Igualmente importante para a eficiência da solução foi a adopção duma estrutura matricial para indexar o conjunto de *n-gramas* associado ao *corpus*. O estudo e as experiências efectuadas na procura da melhor implementação desta estrutura são igualmente um importante contributo deste trabalho, nomeadamente ao nível da análise efectuada relativamente às várias alternativas de implementação dum algoritmo de cálculo da *frequência* de cada *n-grama*.

A análise dos resultados das experiências efectuadas (ver capítulo 6) mostra claramente que a solução proposta conduz a uma solução com um comportamento praticamente linear para os valores de N e de $|\boxtimes|$ utilizados ao longo das experiências. Como vimos, esta linearidade é previsivelmente comutada

para uma resposta com complexidade tipicamente $O(N \log N)$ para valores crescentes da dimensão do *corpus*, para um mesmo valor de *m* e desde que a dimensão do *dicionário* estabilize com o crescimento de *N*.

Os resultados das experiências efectuadas mostram igualmente que a quantidade de memória disponível é um factor limitativo da dimensão máxima do *corpus* que pode ser tratado em cada máquina, o que leva a pensar que soluções alternativas baseadas num misto de memória central e memória secundária devem ser equacionadas tirando partido das soluções apontadas por este estudo. As conclusões tiradas e as soluções propostas ao longo deste estudo, permitem que, quer em termos duma implementação exclusivamente em memória, tal como testado neste trabalho, quer em potenciais sistemas futuros que tirem partido destas conclusões, baseados em memória central e em memória secundária (i.e. ficheiros ou bases de dados), possam ser desenvolvidas soluções eficientes para sistemas de extracção de *MWUs*.

7.1 TRABALHOS FUTUROS

Ao longo do documento foram sendo deixadas algumas pistas respeitantes a trabalhos futuros que prolonguem as conclusões tiradas com o presente estudo, que o complementem ou que o melhorem. A própria característica inovadora da solução proposta para a representação dos *n-gramas* é, por si só, uma importante alavanca para que trabalhos futuros sejam empreendidos, estendo esta solução a outras áreas de aplicação.

Neste contexto, é apresentada em seguida, uma lista de sugestões para trabalhos futuros que de algum modo tirem partido das soluções propostas nesta dissertação. A lista está divida em três grupos de acordo com o tipo de trabalho proposto.

7.1.1 AUMENTAR A FLEXIBILIDADE DA SOLUÇÃO

- Alargamento do *Tokenizer* a *corpus* baseados em outros alfabetos, sintaxes e semânticas que não o utilizado nas experiências efectuadas. Tornar o *Tokenizer* mais flexível para poder suportar outros tipos de *corpus* e de regras para identificação das *unidades lexicográficas*;
- Permitir a reutilização dum mesmo *dicionário* em múltiplas sessões para *corpus* com a mesma estrutura base e regras para a extracção de *unidades lexicográficas*;
- Possibilidade de adaptação e flexibilização do sistema para utilização de outras medidas que não a *ME*, tirando partido das soluções propostas para armazenamento e processamento dos *n-gramas*;
- Possibilitar a inserção ou remoção directa dum ficheiro no *corpus* tirando partido da informação armazenada relativamente ao posicionamento de cada *n-grama* no *corpus*;

7.1.2 AUMENTAR EFICIÊNCIA E ALCANCE DA SOLUÇÃO

- Como ficou provado pelos resultados apresentados, o recurso ao *swapping* de páginas de memória, controlado pelo sistema operativo, é muito pouco eficiente. Este controlo deve passar para a aplicação pela inclusão de técnicas semelhantes, por exemplo, às utilizadas pelas *B-trees* (Sedgewick 1998). Este controlo deve passar pela utilização das ideias propostas nesta dissertação estendidas a ficheiros ou bases de dados. Ver, igualmente, o relatório (Crausser *et al*. 1999) ou (Baeza-Yates *et al*. 1999, capítulo 8), onde é explorada a utilização de *suffix-arrays* em memória externa;

- Tendo em conta a previsivelmente elevada dimensão dos *corpus* é imprescindível a exploração das capacidades de processamento paralelo e/ou distribuição do processamento. Os algoritmos propostos, dadas as suas características, podem perfeitamente ser adaptados por forma a distribuírem o processamento por vários processos correndo em paralelo. Por exemplo, o algoritmo proposto para o cálculo da *frequência* (ver Figura 4.13) divide o conjunto de *n-gramas* em pequenos conjuntos totalmente independentes, por *máscara* e por *token*, podendo lançar vários processos simultâneos para a ordenação dos vários conjuntos e para a inserção dos resultados na *matrix*.

- Tirar partido do facto de, em cada etapa do *WriteMWUs* (ver Figura 4.24), somente ser necessário estarem presentes em memória os *n-gramas* com *k, k*-1 *e k*+1 *tokens*, tomando a *ME* como calculada previamente e armazenada. Para tirar partido desta característica será necessário tratar as *máscaras* por ordem crescente do número de *tokens* e ter um meio alternativo, ficheiros ou base de dados, para armazenamento temporário de *frequências* e *ME*.

  A Figura 7.1 mostra a percentagem de *máscaras* necessárias em cada etapa para o caso de $m = 7$. Por exemplo, quando são processadas *n-gramas* com 2 *tokens* basta termos em memória os *n-gramas* associados a *máscaras* com 1 e 3 tokens, ou seja, precisamos de 3 + 1 + 9 = 13 *máscaras* diferentes presentes. Todos os *n-gramas* associados a outras máscaras podem manter-se em memória secundária.

NumOfTokens	#	soma	%
1	1		
2	3	13	30%
3	9	25	58%
4	13	33	77%
5	11	29	67%
6	5	17	40%
7	1		
Total	43		

Figura 7.1 – Percentagem de *n-gramas* necessários em cada ciclo do *WriteMWUs*

São possíveis reduções entre 30% e 70% no número de *máscaras* necessárias. No entanto, estes valores não têm uma correspondência directa em espaço, dado que não existe uma distribuição uniforme de *n-gramas*, com *frequência* não unitária, por todas as *máscaras*.

- Procura de algoritmos mais eficientes para a contagem de ocorrências dos *n-gramas*, como sejam:
  - Versões alternativas dos algoritmos baseadas em listas;
  - Versões iterativas dos algoritmos, nomeadamente o *QuickSort*;
  - Algoritmo *Forward Radix Sort* (Anderson *et al*. 1994), (Nilson 1996) , (Anderson *et al*.1998) em substituição do *Multikey QuickSort* ;
  - Algoritmo *Key-Indexed Counting* (Sedgewick 1998) como base à obtenção da *frequência*;
  - Utilização de técnicas de *switching* para algoritmos mais eficientes quando o conjunto de elementos a ordenar é pequeno. Tipicamente é utilizado *Insertion Sort* (Anderson *et al*.1998);
- Afinações do código utilizando técnicas mais eficientes, nomeadamente, ao nível da gestão da memória. Uma menor utilização do *new* e do *delete*, possível em alguns casos, poderá conduzir a alguma melhoria na eficiência global do sistema dado que estamos a falar de milhares de operações repetidas;
- Substituição de ciclos *for*, envolvendo basicamente cópia de elementos dentro dum vector, por operações tipo *memory copy*;

7.1.3 PROCURAR NOVOS CAMINHOS

- Encontrar novas relações entre *n-gramas* que permitam optimizar os algoritmos de cálculo da *frequência*, *ME* e *GenLocalMaxs*. Existirão classes de *n-gramas* à semelhança das classes de *sub-strings* propostas em (Yamamoto *et al*. 2000)? Será possível calcular recursivamente a *frequência* ou a *ME* aproveitando os cálculos anteriores nos passos seguintes?

8 REFERÊNCIAS BIBLIOGRÁFICAS

ANDERSON, A. & LARSSON, N.J. & SWANSON, K. (1996), *Suffix Trees on Words*, em Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching 1996, Technical report. LU-CS-TR:95-158, LUNFD6/(NFCS-3107)/1-14/(1995), Dept. of Computer Science, Lund University, Sweden

ANDERSON, A. & NILSON, S. (1994), *A new efficient Radix Sort*, em Proceedings of the 35th Annual IEEE symposium on Foundations of Computer Science, páginas 714-721

ANDERSON, A. & NILSON, S. (1998), *Implementing Radix Sort*, The ACM Journal of Experimental Algorithmics, Volume 3

BAEZA-YATES, R. & RIBEIRO-NETO, B. (1999), *Modern Pesquisa de Informação em Documentos*, ACM Press

BENTLEY, J.L. & SEDGEWICK, R. (1997), *Fast Algorithms for Sorting and Searching Strings*, Em Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms, New Orleans, Janeiro

BENTLEY, J.L. & SEDGEWICK, R. (1998), *Ternary Search Trees*, Dr. Dobb's Journal, Abril

BURKHARDT, S. & CRAUSER, A. & FERRAGINA, P. & LENHOF, H. & RIVALS, E. & VINGRON, M. (1999), *q-gram Based Database Searching Using a Suffix Arrays (QUASAR)*, RECOMB 1999: 77-83

CHARRAS, C. & LECROQ, T. (1997), *Handbook of Exact String-Matching Algorithms*, Laboratoire d'Informatique de Rouen, Université de Rouen, Faculté des Sciences et des Techniques, France

CRAUSSER, A. & FERRAGINA, P. (1999), *Research report : A theoretical and experimental study on the construction of suffix arrays in external memory*, MPI-I-1999-1-001, Março

DALE, N. (1999), *C++ Plus Data Structures*, Jones and Bartlett Publishers, páginas 585-618, 638-642, 434-441

DIAS, G. & GUILLORÉ, S. & LOPES, J.G.P. (1999a), *Language Independent Automatic Acquisition of Rigid Multiword Units from Unrestricted Text Corpora*, Em "Conférence TALN 1999", Cargése, 12-17 Julho

DIAS, G. & GUILLORÉ, S. & LOPES, J.G.P. (1999b), *Multilingual Aspects of Multiword Lexical Units*, Em Proceedings of Workshop on Language Technologies, Ljubljana, Slovenia

DIAS, G. & GUILLORÉ, S. & LOPES, J.G.P. (2000a), *Extracting Textual Associations from Part-Of-Speech Tagged Corpora*, Em Proceedings of the European Association for Machine Translation Workshop, Ljubljana, Slovenia.

DIAS, G. & GUILLORÉ, S. & LOPES, J.G.P. (2000b) *Mining Textual Associations in Text Corpora*, Em Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2000) Workshop on Text Mining, Boston, MA, USA, 20-23 Agosto

DIAS, G. & GUILLORÉ, S. & LOPES, J.G.P. (2000c), *Normalization of Association Measures for Multiword Lexical Unit Extraction*, Em "International Conference on Artificial and Computational Intelligence for Decision, Control and Automation in Engineering and Industrial Applications", Monastir, Tunisia

DIAS, G. & GUILLORÉ, S. & BASSANO, J.C. & LOPES, J.G.P. (2000d), *Combining Linguistics with Statistics for Multiword Term Extraction: A Fruitful Association?*, Em "Recherches d'informations Assistée par Ordinateur-Content-Based Multimedia Information Access", Paris, France

DIAS, G. (2002), *Extraction Automatique d'associations lexicales à partir de corpora*, Tese de Doutoramento, UNL/FCT & Université d'Orléans

FRAKES, B.W. & BAEZA-YATES, R. (1992), *Information Retrieval. Data Structures & Algorithms*, Prentice Hall

FUJII, A. & ISHIKAWA, T. (2000), *Utilizing the World Wide Web as an Encyclopedia: Extracting Term Descriptions from Semi-Structured Texts*, University of Library and Information Science, Japan, arXIv:cs.CL/0011001, Novembro

JAJA, J. (2000), *A perspective on Quicksort*, 10 top algorithms, University of Maryland, 1521-9615/00, IEEE, Janeiro/Fevereiro

GUERREIRO, P. (2000), *Programação com Classes em C++*, FCA – Editora de Informática Lda

GUSFIELD, D. (1999), *Algorithms on Strings, Trees and Sequences. Computer Science and Computational Biology*, Cambridge University Press, páginas 87-207

IKEHARA, S. & SHIRAI, S. & UCHINO, H. (1996), *A statistical method for extracting uninterrupted and interrupted collocations from very large corpora*, em Proceedings of COLING-96 (16th International Conference on Computer Linguistics), Vol.1, páginas 574-579

IRVING, R.W. & LOVE, L. (2000), *The suffix binary search tree and suffix AVL tree*, University of Glasgow Computing Science Department Research Report TR-2000-54

IRVING, R.W. & LOVE, L. (2001), *The suffix binary search trees and suffix arrays*, University of Glasgow Computing Science Department Research Report TR-2001-82

LARSSON, N.J. & SADAKANE, K. (1999), *Faster Suffix Sorting,* Department of Computer Science, Lund University, Sweden, LU-CS-TR:99-214

MANBER, U. & MYERS, G. (1991), *Suffix Arrays: A new method for on-line string searches*, SIAM J. Comput. 22(5), 935-948

MCILROY P. M. & BOSTIC, K. & MCILROY M. D. (1993), *Enginnering Radix Sort*, Computing Systems, 6(1):5-27

MCILROY P. M. & MCILROY M. D. (1997), *Suffix Sort - ssort.cpp*, Lucent Technologies, http:://cm.bell-labs.com/cm/cs/who/doug/ssort.c

NAGAO, M. & MORI, S. (1994), *New Method of N-gram Statistics for Large Number of n and Automatic Extraction of Words and Phrases from Large Text Data of Japanese*, Em COLING'94, 15th International Conference on Computational Linguistic, Agosto 5-9, Vol I, 611-615

NILSON, S. (1996), *Radix Sorting & Searching*, LUTEDX/(TECS-1007)1-109/(1996), Dept. of Computer Science, Lund University, Sweden

PEDERSEN, T. (2002), *N-Gram Statistics Package (NSP). Version 5*, University of Minnesota, Duluth, www.d.umn.edu/~tpederse/code.html

RIBEIRO, A.. & DIAS, G. & LOPES. G. & MEXIA, J. (2001), *Cognates Alignment*. In: Bente Maegaard (ed.). Proceedings of the Machine Translation Summit VIII (MT Summit VIII), Santiago de Compostela, Spain, September 18-22, 2001. European Association of Machine Translation. pp. 287-292.

RODRIGUES, P. & PEREIRA, P. & SOUSA, M. (1998), *Programação em C++. Conceitos básicos e Algoritmos*, FCA – Editora de Informática Lda

RODRIGUES, P. & PEREIRA, P. & SOUSA, M. (2000), *Programação em C++. Algoritmos e Estruturas de Dados*, FCA – Editora de Informática Lda

RUMBAUGHT J.& JACOBSON. I. & BOOCH G. (1999), *The Unified Modeling Language Reference Manual, Addison-Wesley*

SADAKANE, K. (1997), *Comparison among Suffix Array Construction Algorithms*, Department of Information Science, University of Tokyo, IPSJ SIG Notes 97-AL-59

SADAKANE, K. & Imai, H. (1998), *Constructing Suffix Arrays for Large Texts, Department of Information Science*, University of Tokyo, em Proc. of DEWS'98

SEDGEWICK, R. (1998), *Algorithms in C++, Parts 1-4, Third edition*, Addison Wesley, páginas 315-346, 417-450, 623-668

SCHMULLER, J. (1999), *Teach yourself UML in 24 hours*, SAMS

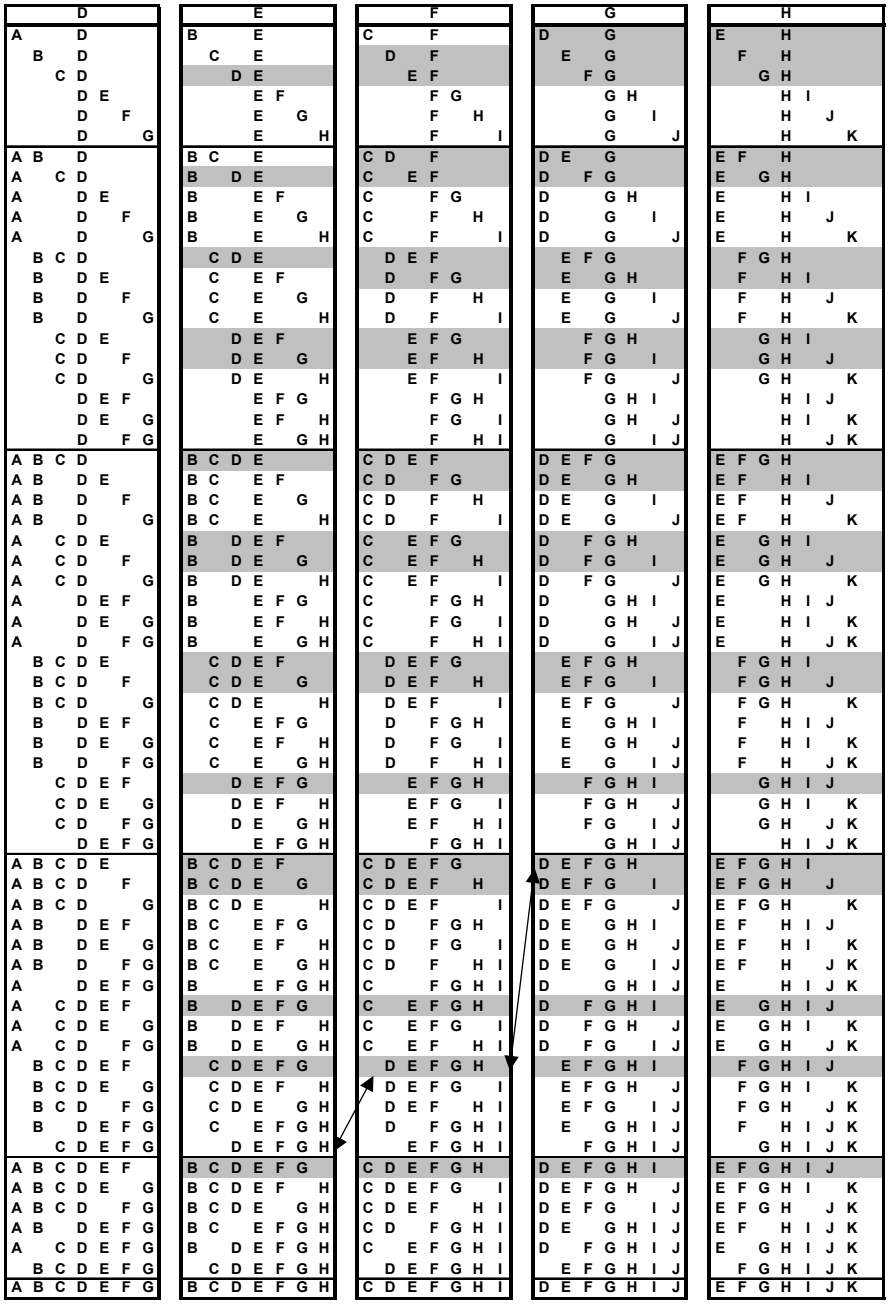
SILVA, F. J. & DIAS, G. & GUILLORÉ S. & LOPES, J.G.P. (1999), *Using LocalMaxs Algorithm for the Extraction of Contiguous and Non-contiguous Multiword Lexical Units*, Em "9th Portuguese Conference in Artificial Intelligence", Pedro Barahona and Júlio Alferes (eds), Lecture Notes in Artificial Intelligence nº1695, Springer-Verlag, Universidade de Évora, Évora, Portugal, Setembro, 113-132

SILVA, J.F. & MEXIA, J. & COELHO, C.A. & LOPES, J.G.P. (2001). *Document Clustering and Cluster Topic Extraction in Multilingual Corpora*. In: Nick Cercone, T. Y. Lin, Xindong Wu (Eds.). Proceedings of the IEEE 2001 International Conference on Data Mining (ICDM'01), San Jose, California, 29 November -2 December, IEEE Computer Society. pp. 513-520.

YAMAMOTO, M. & CHURCH, K.C. (2000), *Using Suffix Arrays to Compute Term Frequency and Document Frequency for All Substrings in a Corpus*, Association for Computational Linguistics, Vol.27, No.1, pp. 1-30.

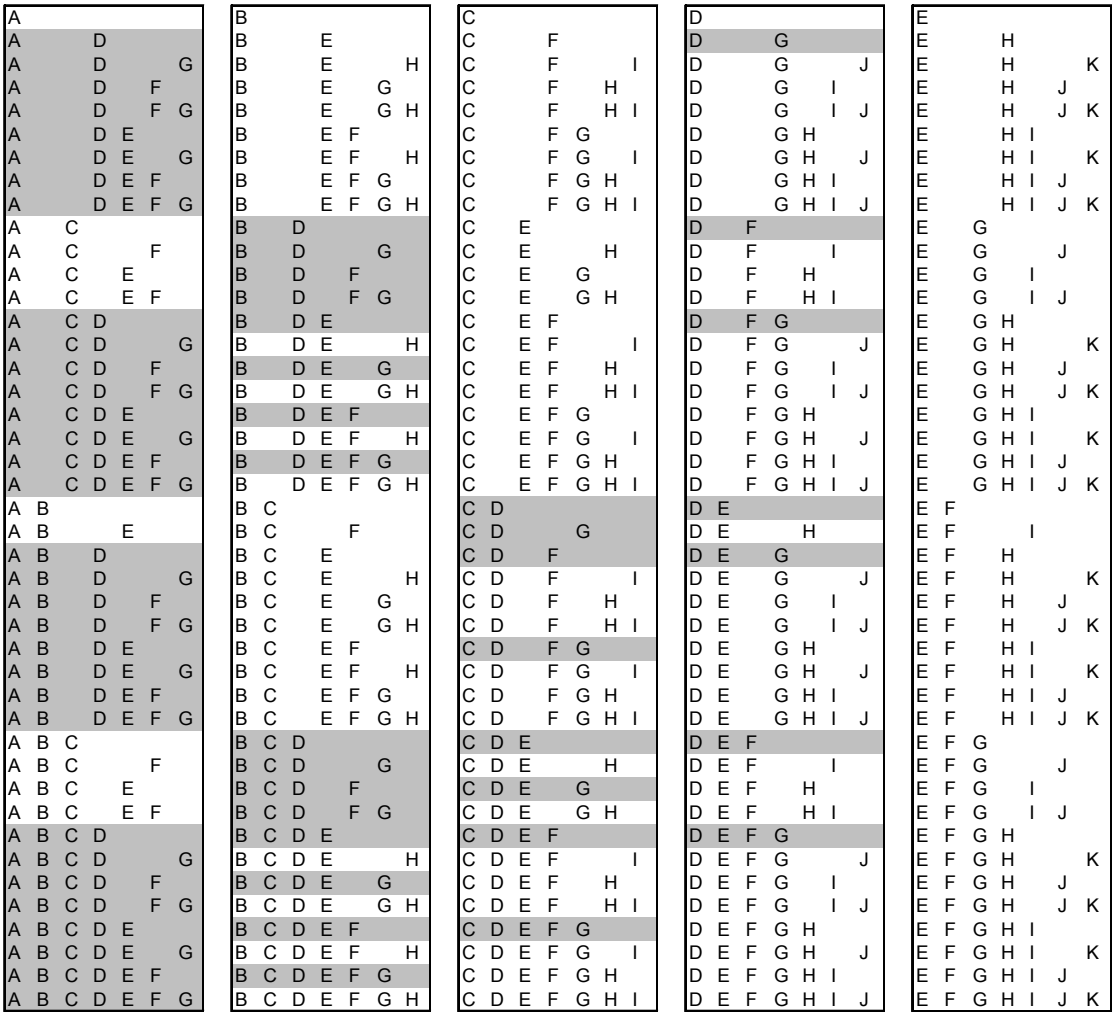
ANEXO A: SEQUÊNCIA DE N-GRAMAS PARA M = 7

corpus = [A B C D E F G H I J K], *m* = 7. As repetições estão assinaladas a cinzento.



ANEXO B: GERAÇÃO DE N-GRAMAS SEM REPETIÇÕES

corpus = [A B C D E F G H I J K], *m* = 7.



ANEXO C: MÁSCARAS PARA M = 7

Lista de *máscaras* válidas

0	1	0	0	0	0	0	0
1	1	0	0	1	0	0	0
2	1	0	0	1	0	0	1
3	1	0	0	1	0	1	0
4	1	0	0	1	0	1	1
5	1	0	0	1	1	0	0
6	1	0	0	1	1	0	1
7	1	0	0	1	1	1	0
8	1	0	0	1	1	1	1
9	1	0	1	0	0	0	0
10	1	0	1	0	0	1	0
11	1	0	1	0	1	0	0
12	1	0	1	0	1	1	0
13	1	0	1	1	0	0	0
14	1	0	1	1	0	0	1
15	1	0	1	1	0	1	0
16	1	0	1	1	0	1	1
17	1	0	1	1	1	0	0
18	1	0	1	1	1	0	1
19	1	0	1	1	1	1	0
20	1	0	1	1	1	1	1
21	1	1	0	0	0	0	0
22	1	1	0	0	1	0	0
23	1	1	0	1	0	0	0
24	1	1	0	1	0	0	1
25	1	1	0	1	0	1	0
26	1	1	0	1	0	1	1
27	1	1	0	1	1	0	0
28	1	1	0	1	1	0	1
29	1	1	0	1	1	1	0
30	1	1	0	1	1	1	1
31	1	1	1	0	0	0	0
32	1	1	1	0	0	1	0
33	1	1	1	0	1	0	0
34	1	1	1	0	1	1	0
35	1	1	1	1	0	0	0
36	1	1	1	1	0	0	1
37	1	1	1	1	0	1	0
38	1	1	1	1	0	1	1
39	1	1	1	1	1	0	0
40	1	1	1	1	1	0	1
41	1	1	1	1	1	1	0
42	1	1	1	1	1	1	1

Lista de *máscaras* inválidas

43	1	0	0	0	0	0	1
44	1	0	0	0	0	1	0
45	1	0	0	0	0	1	1
46	1	0	0	0	1	0	0
47	1	0	0	0	1	0	1
48	1	0	0	0	1	1	0
49	1	0	0	0	1	1	1
50	1	0	1	0	0	0	1
51	1	0	1	0	0	1	1
52	1	0	1	0	1	0	1
53	1	0	1	0	1	1	1
54	1	1	0	0	0	0	1
55	1	1	0	0	0	1	0
56	1	1	0	0	0	1	1
57	1	1	0	0	1	0	1
58	1	1	0	0	1	1	0
59	1	1	0	0	1	1	1
60	1	1	1	0	0	0	1
61	1	1	1	0	0	1	1
62	1	1	1	0	1	0	1
63	1	1	1	0	1	1	1

ANEXO D: ATRIBUTOS DAS MÁSCARAS PARA M = 7

id	dim	nTok	sub-masks {mask id / delta}	super-masks [mask id/delta]
000	1	1		
001	4	2	{000 / 3} {000 / 0}	[023 / 0] [013 / 0] [005 / 0] [003 / 0] [002 / 0] [022 / -1] [010 / -2] [002 / -3]
002	7	3	{001 / 3} {001 / 0}	[024 / 0] [014 / 0] [006 / 0] [004 / 0]
003	6	3	{009 / 3} {001 / 0}	[025 / 0] [015 / 0] [007 / 0] [004 / 0]
004	7	4	{013 / 3} {002 / 0} {003 / 0}	[026 / 0] [016 / 0] [008 / 0]
005	5	3	{021 / 3} {001 / 0}	[027 / 0] [017 / 0] [007 / 0] [006 / 0]
006	7	4	{023 / 3} {002 / 0} {005 / 0}	[028 / 0] [018 / 0] [008 / 0]
007	6	4	{031 / 3} {003 / 0} {005 / 0}	[029 / 0] [019 / 0] [008 / 0]
008	7	5	{035 / 3} {004 / 0} {006 / 0} {007 / 0}	[030 / 0] [020 / 0]
009	3	2	{000 / 2} {000 / 0}	[031 / 0] [013 / 0] [011 / 0] [010 / 0] [023 / -1] [011 / -2] [003 / -3]
010	6	3	{001 / 2} {009 / 0}	[032 / 0] [015 / 0] [012 / 0] [024 / -1]
011	5	3	{009 / 2} {009 / 0}	[033 / 0] [017 / 0] [012 / 0] [025 / -1]
012	6	4	{013 / 2} {010 / 0} {011 / 0}	[034 / 0] [019 / 0] [026 / -1]
013	4	3	{021 / 2} {001 / 0} {009 / 0}	[035 / 0] [017 / 0] [015 / 0] [014 / 0] [027 / -1] [012 / -2] [004 / -3]
014	7	4	{022 / 2} {002 / 0} {013 / 0}	[036 / 0] [018 / 0] [016 / 0]
015	6	4	{023 / 2} {003 / 0} {010 / 0} {013 / 0}	[037 / 0] [019 / 0] [016 / 0] [028 / -1]
016	7	5	{027 / 2} {004 / 0} {014 / 0} {015 / 0}	[038 / 0] [020 / 0]
017	5	4	{031 / 2} {005 / 0} {011 / 0} {013 / 0}	[039 / 0] [019 / 0] [018 / 0] [029 / -1]
018	7	5	{033 / 2} {006 / 0} {014 / 0} {017 / 0}	[040 / 0] [020 / 0]
019	6	5	{035 / 2} {007 / 0} {012 / 0} {015 / 0} {017 / 0}	[041 / 0] [020 / 0] [030 / -1]
020	7	6	{039 / 2} {008 / 0} {016 / 0} {018 / 0} {019 / 0}	[042 / 0]
021	2	2	{000 / 1} {000 / 0}	[031 / 0] [023 / 0] [022 / 0] [031 / -1] [013 / -2] [005 / -3]
022	5	3	{001 / 1} {021 / 0}	[033 / 0] [027 / 0] [032 / -1] [014 / -2]
023	4	3	{009 / 1} {001 / 0} {021 / 0}	[035 / 0] [027 / 0] [025 / 0] [024 / 0] [033 / -1] [015 / -2] [006 / -3]
024	7	4	{010 / 1} {002 / 0} {023 / 0}	[036 / 0] [028 / 0] [026 / 0]
025	6	4	{011 / 1} {003 / 0} {023 / 0}	[037 / 0] [029 / 0] [026 / 0]
026	7	5	{012 / 1} {004 / 0} {024 / 0} {025 / 0}	[038 / 0] [030 / 0]
027	5	4	{013 / 1} {005 / 0} {022 / 0} {023 / 0}	[039 / 0] [029 / 0] [028 / 0] [034 / -1] [016 / -2]
028	7	5	{015 / 1} {006 / 0} {024 / 0} {027 / 0}	[040 / 0] [030 / 0]
029	6	5	{017 / 1} {007 / 0} {025 / 0} {027 / 0}	[041 / 0] [030 / 0]
030	7	6	{019 / 1} {008 / 0} {026 / 0} {028 / 0} {029 / 0}	[042 / 0]
031	3	3	{021 / 1} {009 / 0} {021 / 0}	[035 / 0] [033 / 0] [032 / 0] [035 / -1] [017 / -2] [007 / -3]
032	6	4	{022 / 1} {010 / 0} {031 / 0}	[037 / 0] [034 / 0] [036 / -1]
033	5	4	{023 / 1} {011 / 0} {022 / 0} {031 / 0}	[039 / 0] [034 / 0] [037 / -1] [018 / -2]
034	6	5	{027 / 1} {012 / 0} {032 / 0} {033 / 0}	[041 / 0] [038 / -1]
035	4	4	{031 / 1} {013 / 0} {023 / 0} {031 / 0}	[039 / 0] [037 / 0] [036 / 0] [039 / -1] [019 / -2] [008 / -3]
036	7	5	{032 / 1} {014 / 0} {024 / 0} {035 / 0}	[040 / 0] [038 / 0]
037	6	5	{033 / 1} {015 / 0} {025 / 0} {032 / 0} {035 / 0}	[041 / 0] [038 / 0] [040 / -1]
038	7	6	{034 / 1} {016 / 0} {026 / 0} {036 / 0} {037 / 0}	[042 / 0]
039	5	5	{035 / 1} {017 / 0} {027 / 0} {033 / 0} {035 / 0}	[041 / 0] [040 / 0] [041 / -1] [020 / -2]
040	7	6	{037 / 1} {018 / 0} {028 / 0} {036 / 0} {039 / 0}	[042 / 0]
041	6	6	{039 / 1} {019 / 0} {029 / 0} {034 / 0} {037 / 0} {039 / 0}	[042 / 0] [042 / -1]
042	7	7	{041 / 1} {020 / 0} {030 / 0} {038 / 0} {040 / 0} {041 / 0}	

id = Identificador da *máscara*

dim = Dimensão da *máscara*

nTok = Número de posições correspondentes a *tokens* na *máscara*

sub-masks = conjunto de *sub-masks*

super-masks = conjunto de *super-masks*