Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

Estruturas de Dados para Representação de um Léxico Bilingue

Jorge André Nogueira da Costa (28045)

Lisboa
(2010)

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Estruturas de Dados para Representação de um Léxico Bilingue

Jorge André Nogueira da Costa (28045)

Orientador: Prof. Doutor Luís Manuel Silveira Russo
Co-orientador: Prof. Doutor José Gabriel Pereira Lopes

Trabalho apresentado no âmbito do Mestrado em Engenharia Informática, como requisito parcial para obtenção do grau de Mestre em Engenharia Informática.

Lisboa
(2010)

A todas as estrelas que brilham no meu céu.

Agradecimentos

Em primeiro lugar quero agradecer ao meu orientador, o Professor Luís Russo, por todos os conselhos, pelas horas que dispôs a tirar-me todas as dúvidas que iam surgindo, pela força e apoio quando os resultados pareciam não aparecer e pelas ajudas cruciais que me deu, sempre que me confrontava com problemas mais complexos. Em segundo lugar quero deixar um agradecimento ao Professor Gabriel Pereira Lopes, por me ajudar a compreender melhor o contexto e o "mundo" à volta deste trabalho e pelos conselhos importantes que me deu, tanto no rumo a seguir na fase de preparação, como no desenvolvimento deste documento. Em terceiro lugar agradeço ao Luís Gomes, pela paciência a responder às muitas duvidas que lhe coloquei, por dar sempre ideias importantes para o sistema e por toda a informação crucial que dele recebi, e que muito me ajudou no desenvolver desta dissertação. Foi para mim um privilégio trabalhar com todos eles, não só pelo conhecimento que ganhei, mas também pela simpatia e disponibilidade com que sempre me receberam.

Deixo um agradecimento especial à FCT-UNL, faculdade onde fiz toda a minha formação universitária e da qual sempre tive muita honra e orgulho em fazer parte. Este agradecimento estende-se a todos os Professores que tive nesta instituição, pois se não fosse os conhecimentos recebidos por eles, teria sido impossível desenvolver este trabalho. Presto também a minha homenagem a todos os Professores que me acompanharam desde o início, na Escola Primária, até aos professores do Secundário que, pela sua exigência, me prepararam para a entrada no mundo universitário.

Agradeço aos meus Pais, pois sem o apoio deles ao longo de todos estes anos, não seria ninguém, nem estaria onde estou hoje. Este trabalho nunca poderia ter sido desenvolvido se não fosse motivado pela constante força que me transmitiram, principalmente nestes últimos anos, quando a carga começava a ser mais pesada. Agradeço-lhes também as muitas palavras de incentivo e os conselhos preciosos. Devo-lhes tudo, pelo que foram, pelo que são e pelo que continuarão a ser: os melhores.

Quero agradecer aos meus avós, pela importância que tiveram na minha educação e que tanto contribuíram para a pessoa que sou hoje. Infelizmente, o meu avô partiu antes de poder ver-me a terminar a Licenciatura e a chegar ao fim de mais esta etapa, mas onde quer que esteja, espero que esteja muito orgulhoso. Muito do trabalho aqui desenvolvido é dedicado a ele, pelo pilar que foi para mim durante tantos anos.

Igualmente importantes, os meus padrinhos: a minha tia Maria da Conceição e o meu primo Pedro Mendes, por estarem sempre presentes e por me apoiarem incondicionalmente, em todos os momentos.

Agradeço a todos os meus colegas da FCT, que me acompanharam ao longo destes 5 anos. Sem as muitas ajudas que me foram dando ao longo do tempo, provavelmente não teria terminado esta etapa tão cedo. Foi um prazer enorme ter sido vosso colega, ter criado amizade convosco e ter passado tempos fantásticos e que ficarão na memória, mesmo quando o trabalho parecia ser infinito e a nossa paciência parecia não aguentar muito mais.

Agradeço a todos os outros meus amigos que me apoiaram ao longo da minha vida de estudante, principalmente desde a mudança para o Secundário, até aos que fiz mais recentemente e que já são uma parte tão importante de minha vida.

Por fim, um agradecimento especial à minha segunda família. Agradeço do fundo do coração a todos aqueles que fazem parte do grupo de amigos, do qual tenho o prazer de fazer parte desde que nasci. Vivi de tudo convosco e o carinho que sinto por todos é demasiado grande para explicar. Sinto que caminharam ao meu lado durante todo o tempo e me deram muito apoio. Agradeço aos meus "irmãos" Cátia Murtinheira, Daniela Caetano e André Florindo, por simplesmente existirem e fazerem parte da minha vida. Sempre estiveram ao meu lado e sei que sempre estarão até ao fim.

A todas estas pessoas, um muito obrigado por tudo. Sem vocês, esta dissertação nunca teria sido conseguida, nem este meu grande objectivo alcançado.

Resumo

Através do processo de tradução, vários textos importantes tornaram-se universais e disponíveis em várias línguas. A globalização torna o processo de tradução cada vez mais crítico, devido à maior quantidade de textos disponíveis online, sendo por isso importante desenvolver novos projectos nesta área.

O objectivo desta dissertação foi implementar um sistema para gestão e representação de um léxico bilingue. O léxico bilingue é uma estrutura essencial em ferramentas para tradução e armazena expressões de duas línguas diferentes. A implementação do sistema é baseada em árvores de sufixos generalizadas, uma para cada linguagem representada. As árvores de sufixos são construídas usando o algoritmo de Ukkonen.

Na gestão das duas árvores de sufixos definem-se ligações de correspondência entre duas expressões de línguas diferentes, que sejam adicionados ao sistema, marcando-os como tradução um do outro. Porém, a característica única do sistema é a cobertura, que pode ser monolingue ou bilingue. A cobertura monolingue verifica quais os segmentos de uma expressão que se encontram na respectiva árvore. A cobertura bilingue faz a mesma verificação para um par de expressões, analisando depois quais os diferentes pares de segmentos têm ligação de correspondência entre eles.

Com este tipo de informação, o sistema torna-se muito útil a aplicações que envolvam extracção de pares de tradução e alinhamento de textos paralelos, permitindo descobrir traduções que sejam desconhecidas. No final, é feita uma comparação da eficiência das operações de cobertura com uma implementação baseada nas árvores de sufixos, contra uma implementação baseada em *arrays* de sufixos.

Palavras-chave: Cobertura, Correspondência, Alinhamento, Tradução, Léxico.

Abstract

Through the translation process, several important texts became available in several languages. The globalization makes the translation even more critical, due to the amount of texts available online and in books, that keep growing over time. Thus, it is important to develop new projects in this area.

The goal of this dissertation was to implement a system for management and representation of a bilingual lexicon. The bilingual lexicon is an essential structure, in tools for translation, and stores the expressions of two different languages. The system is based on two generalized suffix trees, one for each represented language. The suffix trees are built using the Ukkonen's algorithm.

In the management of both suffix trees, we define correspondence links between the expressions of the two different languages. The correspondence marks an expression as the translation of the other and vice-versa. The unique characteristic of the system is the coverage operations, which can be unilingual or bilingual. The unilingual coverage checks which segments of an expressions exist in the respective tree. The bilingual coverage does the same verification for a pair of expressions, analyzing next which different pairs of segments have correspondence links between them.

This system can be very useful to translation applications, which are involved with extraction of translation pairs and parallel text alignment, allowing the inferring of new and unknown translations, based on the information from the coverage operations. In the end, we compare the coverage implementation based on suffix trees, with another implementation of the same operations with a different data structure.

Keywords: Coverage, Correspondence, Alignment, Translation, Lexicon.

Conteúdo

Lista de Figuras

Lista de Tabelas

Listagens

1

Introdução

1.1 Motivação

A tradução é um processo importante, que permite o acesso a culturas e a tipos de informação distintas, de uma forma bastante mais fácil para a nossa compreensão. Em várias áreas desde a literatura à informática, é necessário que qualquer tipo de informação seja perceptível a qualquer utilizador, seja qual for a sua nacionalidade, pelo que a tradução tem aqui um papel decisivo. Com a quantidade de informação em formato de texto que existe nos dias de hoje, é importante desenvolver novas ferramentas que tornem o processo de tradução mais eficiente.

O processo de tradução consiste em obter um texto numa determinada língua objectivo B, a partir de uma língua mãe A. Para isso, é essencial que se conheçam traduções na língua B, de termos e expressões da língua A. Nesta tese estudamos uma estrutura que armazena os termos de ambas as linguagens. Essa estrutura é o léxico bilingue.

O léxico armazena várias expressões, ou termos, de uma dada língua, podendo estas ser constituídas por várias palavras ou por apenas uma. Visto que o processo envolve duas línguas diferentes, é necessário representar um léxico bilingue. Desta forma, é possível emparelhar termos de línguas distintas, de modo a que se possam definir pares de tradução entre eles. Torna-se assim muito importante que o léxico seja organizado e gerido de forma eficiente, de modo a manter esta informação actual.

A Tabela 1.1 mostra um exemplo de um pequeno léxico bilingue, com termos em inglês representados na coluna esquerda e termos em português na coluna direita. Os

termos que se encontram na mesma linha, assumem-se como pares de tradução conhecidos pelo léxico.

Tabela 1.1: Exemplo de um pequeno léxico bilingue Inglês-Português

bilingual	bilingue
bilingual suffix tree	árvore de sufixos bilingue
suffix	sufixo
suffix tree	árvore de sufixos
tree	árvore

Como o léxico é uma estrutura essencial para a tradução, a implementação definida nesta dissertação, pode ser muito útil para outro tipo de protótipos nesta área, principalmente pelas suas características únicas, apresentadas mais à frente neste relatório, e pela eficiência que permite na gestão do léxico.

1.2 Apresentação do Sistema

O principal objectivo desta dissertação foi apresentar um sistema para representação e gestão de um léxico bilingue, usando estruturas de dados que permitissem definir operações eficientes, e que possam trazer vantagens a outros projectos de tradução e não só. Portanto, o foco do trabalho realizado centrou-se nas estruturas de dados em si, estando apenas relacionado indirectamente com o processo de tradução.

Para representar o léxico bilingue usamos árvores de sufixos, pois são estruturas de dados que devido às suas características, apresentadas em detalhe no Capítulo 2, permitem conseguir tempo linear, tanto na sua construção, como na realização de pesquisas. Dado que o léxico contém vários termos distintos, são utilizadas especificamente árvores de sufixos generalizadas, devido às árvores simples apenas permitirem que seja armazenada uma única expressão, ou seja, uma única cadeia de caracteres.

As árvores de sufixos têm aplicações muito variadas na área da bioinformática [BSTU09], porém em termos de linguística o seu uso não é muito comum. No entanto, existem alguns resultados estabelecidos com outra estrutura da mesma categoria das árvores de sufixos, os *arrays* de sufixos [ALG09, GAL09, CBBS05b, CBBS05a].

Usando árvores de sufixos, definimos o sistema LEXMAN, de *Lexicon Manager* ou Gestor de Léxico em português. O sistema faz uma gestão de toda a informação do léxico, assim como fornece operações únicas sobre os dados armazenados, introduzidas de seguida, que tornam este sistema singular.

1.2.1 Solução Desenvolvida

O léxico é representado por duas árvores de sufixos generalizadas, uma para cada uma das linguagens. Cada par validado que é recebido, é armazenado na árvore respectiva, ou seja, um termo numa língua é adicionado à árvore representativa dessa língua, seguindo-se a mesma lógica para a outra árvore e para o outro termo. Por isso, o LEXMAN oferece algumas operações básicas de gestão de dados, como a adição ou remoção de um par de tradução.

As duas árvores de sufixos generalizadas que fazem parte do sistema, são construídas com recurso ao algoritmo de Ukkonen [Ukk95], caracterizado pelos tempos lineares na construção de árvores de sufixos.

Com os termos repartidos em duas árvores, é possível definir os pares de tradução com ligações de correspondência, entre as duas estruturas. A correspondência entre dois termos de línguas diferentes, é uma ligação que marca esses mesmos termos como tradução um do outro. Os termos podem ser frases completas, no sentido gramatical (”A estrutura usada é uma árvore de sufixos generalizada”), ou apenas algumas partes (”árvore de sufixos generalizada”), assim como nomes de entidades (“Comissão Europeia”) ou palavras únicas (”Europa”). Na Figura 1.1, baseada no léxico da Tabela 1.1, é possível verificar que “suffix tree” tem uma ligação de correspondência com ”árvores de sufixos” (a tracejado).

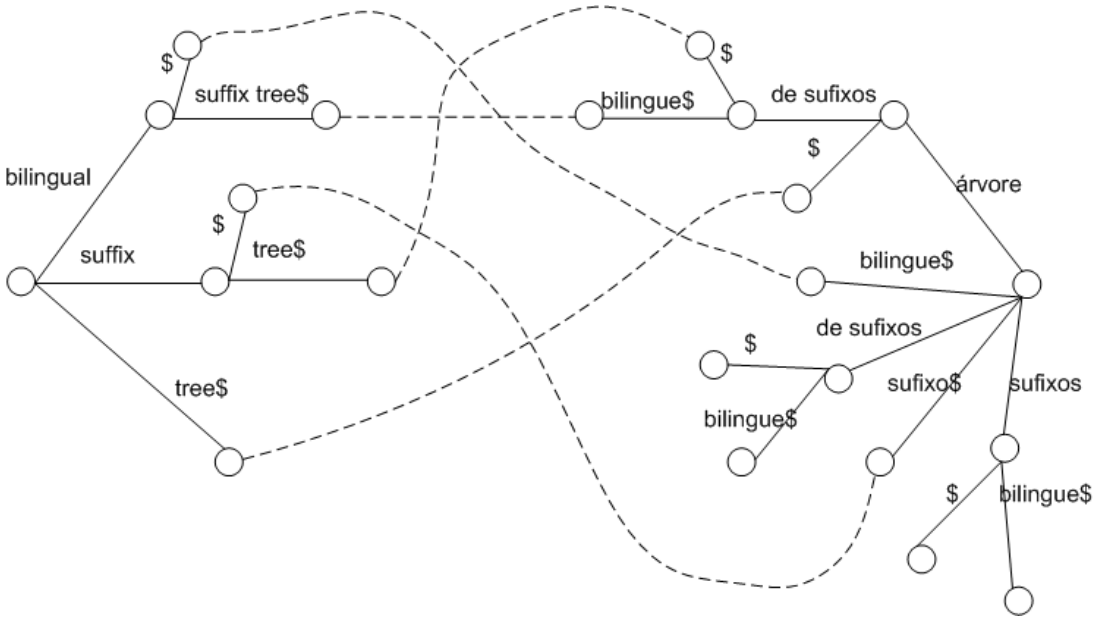

Figura 1.1: Ligações de correspondência entre os nós das duas árvores

Esta é uma característica do sistema muito importante por duas razões. Em primeiro lugar, com uma ligação de correspondência é possível, na extracção de traduções, saber à partida se dois termos formam um par de tradução, ou seja, se são uma possível tradução um do outro. Em segundo lugar, é útil para as operações seguintes.

Devido às características do sistema, este fornece algumas operações que podem ser consideradas extra em relação à gestão do léxico. Na realidade, acabam por estar indirectamente relacionadas com a gestão, pois são elas que permitem que o léxico receba mais e melhores dados. Estas operações permitem fazer perguntas sobre os dados armazenados e são denominadas por cobertura monolingue e bilingue.

A cobertura monolingue verifica quais são os vários segmentos de um termo que se encontram na árvore da sua língua, devolvendo aqueles que não se encontram no sistema. Diz-se que um termo tem cobertura total, se todo o termo existe na árvore respectiva, o que pode acontecer se o termo completo existir ou se todos os seus segmentos existirem. No entanto, é possível ter-se apenas cobertura parcial, se só forem conhecidas alguns segmentos. Por exemplo, com o termo multi-palavra "European Public Assessment Report", os vários segmentos possíveis são: o termo completo, "European Public Assessment", "Public Assessment Report", "European Public", "Assessment Report"e as quatro palavras individuais.

A cobertura bilingue é mais complexa, sendo que aqui, a correspondência tem um papel bastante importante. Os dois termos passam por uma verificação semelhante à cobertura monolingue, cada um na sua árvore respectiva. Para os segmentos encontrados, verifica-se se existe alguma ligação de correspondência entre eles. Existe cobertura total quando os termos completos são encontrados e todos os seus segmentos têm ligações de correspondência entre si.

Na Figura 1.1, o termo ”suffix” esta coberto na árvore em inglês e ”de sufixos” na árvore em português, mas o par ”suffix” <=> ”de sufixos”, não pode ser considerado como tendo cobertura bilingue, pois não existe uma ligação de correspondência entre os termos. Isto acontece, pois ”de sufixos” é um sufixo de ”árvore de sufixos” e ainda não se conhece a sua tradução directa. Um caso de cobertura parcial seria, por exemplo, ”generalized suffix trees” <=> ”árvores de sufixos generalizadas”, onde ”generalized” <=> ”generalizadas” não tem cobertura no léxico.

Qualquer uma destas operações, segundo o nosso conhecimento, não é usada em qualquer tipo de estrutura semelhante, tornando por isso a nossa implementação do léxico inovadora. A cobertura permite descobrir informação muito importante, com complexidades relativamente baixas. No Capítulo 5 são mostrados alguns resultados, que mostram a menor eficiência da operação de cobertura com outras implementações.

Juntando todas as funcionalidades, o LEXMAN oferece sete operações. Quatro delas são para gerir a estrutura, enquanto as outras três funcionam como perguntas ao sistema, estando relacionadas com os conceitos de correspondência e cobertura. Nos argumentos, sempre que sejam termos, o argumento à esquerda refere-se sempre à mesma língua, que pode ser considerada por língua 1, enquanto o argumento da direita se refere à outra língua representada. O sistema foi testado com as línguas Inglês e Português, como a língua 1 e 2 respectivamente.

- **add_pair(termo, termo) –** método para adicionar um par ao sistema, sendo o primeiro termo adicionado à primeira árvore e o segundo termo adicionado à outra árvore representada.
- **delete_pair(termo, termo) –** método para "eliminar"um par do sistema. Os termos continuarão nas respectivas árvores, sendo apenas eliminada a ligação de correspondência entre eles.
- **add_pairs(nome do ficheiro) –** método que lê vários pares de termos de um ficheiro, adicionando-os ao sistema.
- **all_pairs(nome do ficheiro) –** método que escreve em ficheiro, todos os pares existentes no sistema, podendo funcionar como uma espécie de cópia de segurança para os dados na árvore.
- **is_pair(termo, termo) –** método que verifica se dois termos têm uma ligação de correspondência entre si, ou seja, se formam um par de tradução.
- **mono_coverage(lingua do termo, termo) –** método que determina a cobertura monolingue de um termo, na árvore de uma dada linguagem.
- **bili_coverage(termo, termo) –** método que determina a cobertura bilingue de dois termos no léxico.

O LEXMAN será integrado num processo iterativo de três passos, onde é feito alinhamento de textos, extracção de pares de tradução e validação de entradas extraídas, que será explicado em maior detalhe na secção seguinte.

1.2.2 Contexto

Gomes desenvolveu um protótipo para alinhamento de textos paralelos [Gom09], que utiliza unicamente léxicos bilingues externos. Diz-se que dois textos são paralelos se um deles for tradução do outro, ou se foram ambos traduzidos a partir do mesmo

texto fonte. O alinhamento procura identificar os vários segmentos de um texto, que correspondam com segmentos do outro texto. Entende-se por segmento de um texto, partes do mesmo que sejam constituídas por palavras, frases ou nomes. Ambos os textos devem ser paralelos, para que seja possível encontrar correspondências.

No contexto da pesquisa realizada por Gomes et al [GAL09], o alinhamento é apenas uma parte de um processo iterativo bastante mais complexo, como se pode verificar na Figura 1.2, onde o léxico acumula novos pares de tradução no final de cada iteração. Para iniciar o processo, usa-se um pequeno léxico bilingue que efectua o primeiro alinhamento. Este léxico pode ser obtido através de dicionários bilingues disponíveis online, como o IDP [Cha] ou pode ser extraído do corpus paralelo com outro método de alinhamento, tal como o *unl-aligner* [IL05] ou o GIZA++ [ON03].

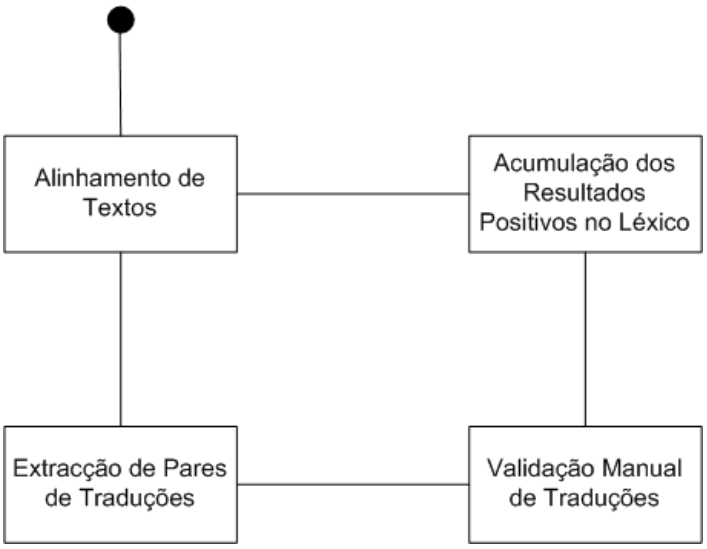

Figura 1.2: Processo iterativo de alinhamento e extracção de traduções.

De seguida, usando o método demonstrado por Aires et al. [ALG09], extraem-se novos pares de traduções desconhecidos de palavras ou frases. Estas entradas são submetidas a um processo de avaliação linguística, sendo que aquelas que são anotadas como positivas, são posteriormente submetidas ao léxico. Consequentemente, este aumenta a sua dimensão após cada iteração terminar, o que acaba por tornar o alinhamento na próxima iteração ainda mais eficiente, por possuir mais informação à sua disposição. Esta melhoria verifica-se para as iterações seguintes, até a precisão do algoritmo parar de melhorar significativamente, pois atinge-se a partir de uma certa altura um valor assimptótico.

Por fim, existe o passo de validação linguística das entradas extraídas. A aproximação seguida neste processo, afasta-se do estado de arte [IAH07], ao não ter uma

natureza completamente automática. Em vez disso, é utilizada uma sinergia Pessoa-Máquina que permite melhorar os resultados através de informação fornecida ao sistema, o que tem um impacto directo na qualidade do léxico e indirecto no alinhamento. O sistema de validação espera pela aprovação (ou desaprovação) por parte do utilizador, para cada par recebido. O passo de validação é bastante mais demorado que todos os outros juntos, mas os resultados obtidos são de melhor qualidade.

1.3 Contribuições

Analisando as características do léxico bilingue, pode perceber-se que, provavelmente o uso de bases de dados seria directo e até uma opção de implementação simples e viável. Porém, para se obter alguns dos resultados que o LEXMAN consegue devolver, as perguntas feitas a um sistema implementado com a base de dados, seguindo uma lógica de três tabelas, teriam de ser complexas e extensas. Mesmo que se tentasse simplificá-las, isso levaria a que o sistema tivesse um trabalho extra demasiado pesado, devido à grande quantidade de dados que teria de processar.

Numa base de dados, para se definir a operação de cobertura monolingue por exemplo, teria de ser feita uma pesquisa pelo termo todo e depois o mesmo processo para todos os seus segmentos. Usando o exemplo da expressão "European Public Assessment Report" já mencionado, teria de se repetir o processo de pesquisa para todos os segmentos até encontrar algo. Para termos de uma palavra ou para casos em que os termos completos existam no léxico, a base de dados seria eficiente, mas esperam-se muito mais perguntas ao LEXMAN sobre os casos mais complexos.

Assim, com o uso de árvores de sufixos, além de resolver-se a questão do armazenamento, consegue definir-se operações mais complexas, como a de cobertura, de maneira muito mais eficiente, devido à facilidade e eficácia inerente a percorrer a árvore em busca de informação.

Também com a cobertura, é possível descobrir para termos multi-palavra, se estes existem por completo ou caso não existam, que segmentos são desconhecidos. Analisando o processo iterativo já demonstrado, é evidente que a extracção de traduções beneficia directamente com estas operações, pois estas tornam a descoberta de novos pares de tradução mais facilitada, utilizando aqueles já são conhecidos e a informação proveniente do alinhamento. Com os segmentos dos textos alinhados e sabendo que partes desses segmentos são desconhecidas, então pode-se fazer a correspondência do que falta num texto para o outro, de acordo com o que fizer mais sentido.

Outra característica interessante do LEXMAN é o benefício que traz, não só na gestão do léxico, mas também no alinhamento, ao tornar a operação de *lookup* de Gomes

[Gom09] mais eficiente. O problema da implementação actualmente descrita por Gomes, é o de não ter uma boa performance para textos pequenos, pois é feito um pré-processamento dos textos e não do léxico. Inicialmente esta metodologia funcionava bem, pois a maior parte dos textos para alinhar eram maiores que o léxico utilizado, porém com o processo iterativo já descrito neste capítulo, essa situação já não se verifica, devido à constante acumulação de dados na estrutura. Com a utilização das árvores de sufixos para localizar os termos do léxico nos textos, a eficiência da operação aumentaria significativamente. Esta operação é descrita no Capítulo 3.

Por fim, este sistema pela sua interface simples e pelas suas funcionalidades, pode ser facilmente adaptável a outros sistemas e utilizado por aplicações a alto nível, que possam usufruir igualmente das operações implementadas. Além disso, a adição de novas funcionalidades está também em aberto, nomeadamente um maior desenvolvimento das operações de cobertura.

1.4 Estrutura do Documento

Nesta secção, é feito um guia da estrutura do documento, onde são apresentados os vários capítulos que constituem o mesmo, junto com uma curta introdução aos conteúdos de cada um deles.

- **Capítulo 2: Árvores de Sufixos e Algoritmo de Ukkonen.** Capítulo onde é explicado com mais detalhe, o essencial sobre as estruturas de dados usadas na implementação, começando pelas árvores de sufixos para uma sequência de caracteres apenas, ate às árvores de sufixos generalizadas. Também será introduzido o algoritmo de Ukkonen, usado para construir as árvores de sufixos.

- **Capítulo 3: Trabalho Relacionado.** Neste capítulo é analisada a operação de *lookup* em mais pormenor, introduzindo os conceitos necessários para a sua melhor percepção. Também serão brevemente apresentados os projectos de alinhamento e extracção de traduções. Por fim, será feita uma breve abordagem a outro projecto na área de linguística, baseado em árvores de sufixos.

- **Capítulo 4: Implementação do Sistema.** Nesta secção do relatório, é explicada em maior detalhe toda a implementação feita da estrutura em si, como das operações oferecidas pelo LEXMAN. No que toca às operações, é mostrado o *output* que é gerado por cada uma delas, com alguns exemplos práticos.

- **Capítulo 5: Apresentação de Resultados.** Neste capítulo são apresentados resultados de consumos de tempo e memória, assim como uma análise da complexidade temporal. Também são mostrados alguns resultados comparativos com

outra implementação das operações de cobertura, nomeadamente usando *arrays* de sufixos.

- **Capítulo 6: Considerações Finais.** Capítulo onde é feita uma análise a todo o projecto, de modo a aferir se todos os objectivos foram propostos. Também são dadas algumas sugestões de trabalho futuro interessantes, que só poderão trazer benefícios ao LEXMAN e ao contexto onde seria utilizado.

- **Apêndice A: Comandos das Operações do LEXMAN.** Neste apêndice, mostramos os vários comandos que se devem usar, para executar as várias operações que o LEXMAN oferece.

2

Árvores de Sufixos e Algoritmo de Ukkonen

2.1 Árvores de Sufixos

Neste capítulo, a letra n é usada para indicar o tamanho de uma cadeia de caracteres. São usadas outras letras e outros símbolos, mas em casos mais particulares, sendo os seus significados indicados devidamente, quando são utilizados.

2.1.1 Introdução aos Índices de Texto Completo

É muito comum a utilização da operação de pesquisa por uma dada cadeia de caracteres, noutra de tamanho consideravelmente superior, como é o caso de um texto. Efectuar este tipo de operações pode ter um custo demasiado elevado.

Quando é usada uma lógica sequencial para resolver este tipo de problemas, é necessário percorrer todo o texto em busca das ocorrências da cadeia de caracteres em questão, obtendo-se complexidades na ordem de $O(m+n)$ ou $O(m \times n)$, em que m seria o tamanho do texto. Para evitar isso, podemos usar umas estruturas de dados denominadas de índices, que por não realizarem pesquisas sequenciais, conseguem obter complexidades temporais mais eficientes.

Existem algumas condicionantes no que toca ao uso de índices. Navarro e Mäkinen [NM07] enunciaram algumas dessas situações que são necessárias analisar antes da

sua utilização, de modo a aferir-se se os índices são de facto uma boa opção.

- Tem de existir espaço de armazenamento suficiente para fazer a gestão do índice e permitir acessos eficientes.
- A busca sequencial deve ser dispendiosa o suficiente, em termos de recursos temporais consumidos.
- As alterações aos dados devem ser suficientemente frequentes, de modo a não tornar a manutenção do índice demasiado dispendiosa, quando comparada com o pouco uso que se dá à estrutura.

Nesta dissertação, utilizamos índices de texto completo. Estas estruturas de dados são caracterizadas por armazenarem todo um texto (enquanto sequência de caracteres), tendo ainda suporte para o uso de emparelhamento de cadeia de caracteres.

Os índices de texto completo mais importantes são as árvores de sufixos e os *arrays* de sufixos. Neste capítulo, as árvores de sufixos serão explicadas mais ao pormenor, pois são as estruturas utilizadas na implementação do projecto. No Capítulo 4 é dada uma curta explicação do porquê da utilização das árvores, ao invés da possibilidade de usar os *arrays*. Para ajudar à explicação das estruturas, introduzimos antes o conceito de *trie* de sufixos e apresentamos uma noção mais precisa do que é um sufixo e um prefixo, conceitos chave para melhor compreensão das árvores.

2.1.2 Prefixo e Sufixo

Uma cadeia de caracteres é uma sequência $C_1C_2...C_n$, onde n é o seu número total de caracteres. Um exemplo de uma cadeia é ”abac”, com n= 4. Nesta dissertação, as notações cadeia e sequência de caracteres podem ser utilizadas para referirem-se a este tipo de estruturas.

Um prefixo de uma cadeia de caracteres é uma outra sequência $C_1...C_i$, caracterizada por consistir nos primeiros i caracteres da cadeia inicial, com $i \leq n$. Exemplos de prefixos para ”abac”: ”aba”, ”a”, ”ab”, ”abac”.

Um sufixo é também uma cadeia de caracteres, só que ao contrário do prefixo é do tipo $C_i...C_n$, consistindo nos últimos n-i+1 caracteres da sequência inicial, com $i \leq n$. Exemplos de sufixos para ”abac”: ”bac”, ”c”, ”ac”, ”abac”.

2.1.3 Introdução às *Tries*

Uma *trie* [Fre60, Knu73] armazena um conjunto de cadeias de caracteres e suporta pesquisas, em tempo proporcional ao tamanho da sequência procurada. A estrutura é

construída em tempo $O(s)$ [NM07], assumindo que o tamanho do alfabeto é constante e que s corresponde à soma do tamanho de todas as cadeias representadas.

Cada nó da *trie* representa um prefixo da sequência de caracteres no conjunto. A raiz representa o prefixo vazio e para cada nó expandido, vão sendo encontrados novos prefixos até ao fim da *trie*. Numa pesquisa, utiliza-se o primeiro carácter da sequência a procurar, expandindo-se para o nó seguinte caso haja correspondência com a sua etiqueta. De seguida, analisa-se o segundo carácter da sequência a pesquisar e tenta-se expandir, fazendo o mesmo para o resto da sequência, até se atingir uma folha, o que significa que a pesquisa foi bem sucedida.

No entanto, é possível chegar a um ponto em que os nós seguintes não correspondem ao carácter a ser analisado, o que significa que a cadeia de caracteres não se encontra armazenada. Este processo é feito em $O(|S|)$, com S a ser a cadeia de caracteres a pesquisar e $|S|$ o seu tamanho.

Uma *trie* de sufixos [NM07], é uma *trie* construída sobre todos os sufixos de uma cadeia de caracteres. Uma *trie* de sufixos tem $O(n^2)$ nós. Pode ver-se um exemplo desta estrutura na Figura 2.1. No entanto, os caminhos únicos a partir de um nó podem ser substituídos por um apontador para o resto do sufixo, diminuindo assim o número de nós para $O(n)$ [FS96]. Seguindo esta ideia, chega-se à definição de árvore de sufixos.

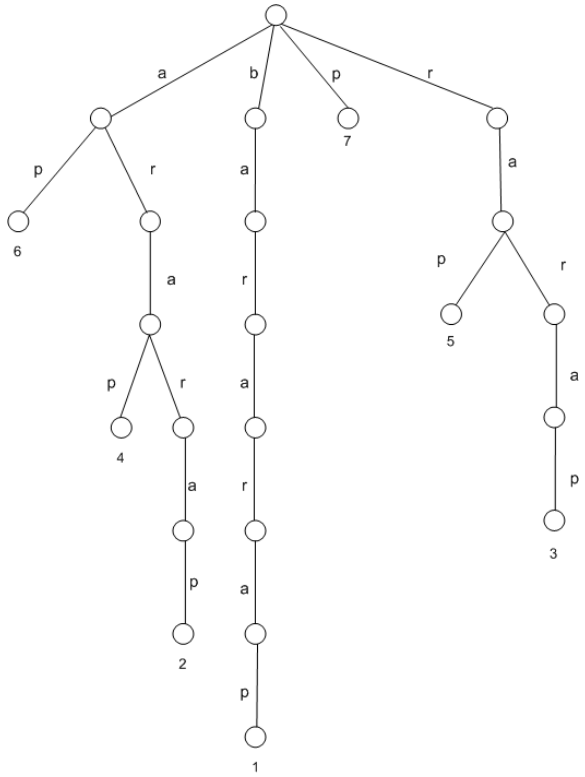

Figura 2.1: *Trie* de sufixos para a sequência "bararap"

2.1.4 Árvores de Sufixos

Uma árvore de sufixos [NM07], trata-se de uma *trie* de sufixos onde cada um dos caminhos unários, referidos no final do capítulo anterior, é convertido numa única aresta. Cada uma destas arestas, terá como etiqueta a cadeia de caracteres obtida a partir da concatenação de todos os caracteres desse caminho. Desta forma, diminui-se o número de nós que acabavam por ser desnecessários, originando um número de nós de ordem $O(n)$ [Far97]. Segundo esta lógica, caminhos mais longos tornam-se bastante mais curtos, como se pode verificar na Figura 2.2.

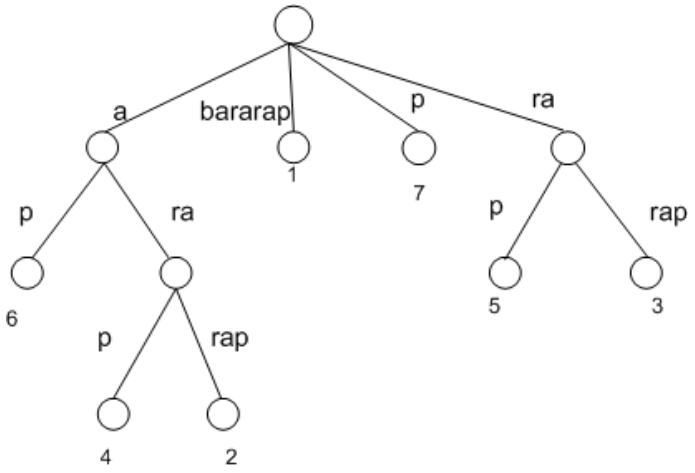

Figura 2.2: Árvore de sufixos para a sequência "bararap"

Uma árvore de sufixos de uma cadeia de caracteres S de tamanho n, é uma árvore com exactamente n folhas numeradas de 1 a n [Gus97]. Cada nó interno, possivelmente com excepção da raiz, tem pelo menos dois descendentes, cujas arestas têm como etiqueta uma sub-cadeia não vazia de S. Nenhuma etiqueta de uma aresta proveniente de um nó, pode ter um primeiro carácter igual ao primeiro carácter de outra aresta, proveniente do mesmo nó (ver arestas da Figura 2.2 que saem da raiz).

As folhas da árvore apontam para a posição da cadeia de caracteres original, como por exemplo um texto, onde o respectivo sufixo encontrado se inicia. Os números inteiros representados na Figura 2.2, são um exemplo de um índice que representa a posição no texto para onde a folha aponta. A característica chave das árvores de sufixos está relacionada com estes apontadores, onde para cada folha i, a concatenação de todas as etiquetas no caminho da raiz até i, corresponde ao sufixo de S que começa na posição i, ou seja, $S[i..n]$. A folha com o número 3, aponta para a 3ª posição da cadeia de caracteres representada, podendo-se obter o sufixo "rarap".

Com a definição anterior, não é garantido que exista uma árvore de sufixos para

qualquer cadeia de caracteres S. Se S contiver um prefixo de um dos seus sufixos, então é impossível existir uma representação com árvore de sufixos, já que o caminho para o primeiro sufixo não terminaria numa folha. Como exemplo, dada uma cadeia de caracteres "rapra", então "ra" seria um prefixo do sufixo "rapra", pelo que o caminho com etiqueta "rapra", nunca terminaria numa folha, pois a seguir a "ra", podemos ter o fim da sequência ou "pra". Para evitar este problema, a cadeia teria de ser, por exemplo "raprac", ou seja, o último carácter tem de ser único em toda a sequência, para que nenhum sufixo possa ser prefixo de outro representado.

Desta forma, haveria muitas sequências de caracteres que não podiam ser representadas. Portanto, convenciona-se um carácter final, que não seja usado em qualquer palavra, e que sirva de terminador universal. Esse carácter usualmente é o $, sendo então armazenada na árvore a sequência S$. Daqui para a frente, esta notação não é utilizada, dado que $ passará a estar sempre implícito em S.

As árvores de sufixos são muito usuais para os problemas de emparelhamento de cadeias de caracteres. Isso deve-se à capacidade que a estrutura tem de resolver este tipo de problemas em tempo linear. No entanto, a sua principal virtude, está relacionada com o facto de conseguir resolver outros problemas bem mais complicados, também em tempo linear. Um exemplo é a sub-cadeia de caracteres comum, o que é muito útil para a operação de cobertura.

Tanto o espaço ocupado, como o tempo de construção de uma árvore de sufixos é de ordem $O(n)$. Mas para pesquisar uma cadeia de caracteres, o tempo será de ordem $O(m)$, com m a ser o tamanho da sequência a procurar. Deste modo, é uma operação cuja complexidade temporal é praticamente independente da dimensão da árvore. Este tipo de valores são bastante úteis, pois normalmente as sequências de caracteres que são recebidas para pesquisas, após a árvore estar construída, são pequenas (m é bastante inferior a n) e em grande número.

2.2 Algoritmo de Ukkonen

Existem vários algoritmos para construir uma árvore de sufixos. O primeiro que surgiu com tempo linear, foi o algoritmo de Weiner [Wei73]. Mais tarde, surgiu o algoritmo de McCreight [McC76], também linear, mas mais eficiente em termos de espaço, quando comparado com o antecessor. Porém, o algoritmo escolhido foi o de Ukkonen [Ukk95], pois além de permitir construir uma árvore de sufixos em tempo linear, tem todas as regalias de espaço que o algoritmo de McCreight oferece e tem uma descrição, prova e análise temporal mais simples. Essa simplicidade advém do algoritmo ser

desenvolvido a partir de um método muito simples, mas ineficiente, que exige algumas técnicas de implementação, para tornar os tempos gradualmente mais favoráveis, especialmente para o pior caso.

2.2.1 Árvores de Sufixos Implícitas

O algoritmo de Ukkonen constrói uma sequência de árvores de sufixos implícitas. A última árvore da sequência, será convertida numa árvore de sufixos explícita, para uma cadeia de caracteres S, que é o objectivo final do algoritmo.

Uma árvore de sufixos implícita [ABM08, Gus97] de uma cadeia de caracteres S é obtida a partir da árvore de sufixos de S, ao remover em primeiro lugar todos os terminadores $ das etiquetas das arestas. De seguida removem-se todas as arestas que ficaram sem etiqueta. Por fim, retiram-se todos os nós que tenham menos de dois filhos. Na Figura 2.3 pode ver-se a árvore de sufixos da cadeia "rapra", juntamente com a árvore de sufixos implícita obtida a partir da mesma.

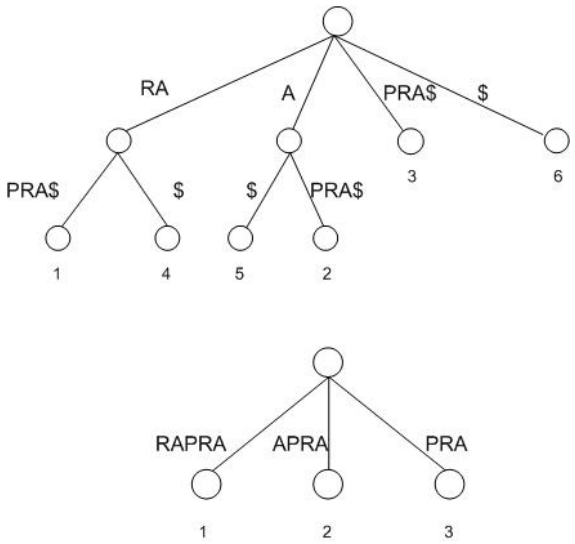

Figura 2.3: Árvore de sufixos implícita para a sequência "rapra"

É possível observar que a árvore de sufixos implícita tem menos folhas. No entanto, isso só acontece quando a cadeia representada, contém um sufixo que também é sub-cadeia própria. Ou seja, tendo a cadeia "rapra", "ra" é sufixo mas também é sub-cadeia de "rapra", com o primeiro sublinhado a representar a sub-cadeia e o segundo o sufixo.

Mesmo não tendo uma folha para cada sufixo, a árvore de sufixos implícita contém todos os sufixos de S. A única questão é que, se não acabar numa folha, o final do sufixo não terá nenhum marcador a indicar o seu fim. Por esse motivo, as árvores de sufixos

implícitas possuem menos informação do que uma explícita, não sendo por isso boas alternativas quando se procura a poupança de recursos de memória.

O algoritmo de Ukkonen constrói uma sequência de árvores de sufixos implícitas T_n, para cada prefixo $S[1..i]$ de S. Em cada iteração o valor de i é incrementado em 1 valor. A árvore de sufixos propriamente dita é construída a partir de T_n e o tempo de execução do algoritmo é de ordem $O(n)$. Se o último carácter da cadeia for único, então T_n será igual à árvore de sufixos explícita final. A primeira descrição feita de seguida, dá origem a um algoritmo com tempo cúbico, ou seja, $O(n^3)$, que será optimizado até atingir tempo $O(n)$.

2.2.2 Algoritmo Básico

O algoritmo é dividido em n fases. Na fase $i+1$, T_{i+1} é construída a partir de T_i. Cada fase $i+1$ é dividida em $i+1$ extensões, uma para cada $i+1$ sufixos de $S[1..i+1]$. Na extensão j da fase $i+1$, o algoritmo encontra, em primeiro lugar, o fim do caminho iniciado na raiz e etiquetado com a sub-cadeia $S[j..i]$. Em seguida, estende-se a sub-cadeia adicionando o carácter $S(i+1)$ no final, a não ser que este já lá se encontre. Consequentemente, em toda a fase $i+1$, a sequência $S[1..i+1]$ é colocada em primeiro lugar na árvore, seguido de $S[2..i+1]$, $S[3..i+1]$, etc, correspondendo cada um destes passos às várias extensões. A extensão $i+1$ da fase $i+1$, estende o sufixo vazio de $S[1..i]$, ou seja, coloca o carácter $S(i+1)$ na árvore. Com estas noções, pode inferir-se que T_1 contém apenas uma única aresta etiquetada com o carácter $S(1)$. O pseudo-código do algoritmo é apresentado na Listagem 2.1, sendo conhecido como o Algoritmo de Extensão Única (AEU) [Gus97].

Listing 2.1: Algoritmo de Extensão Única

```
1  construir_arvore(T1);
2  for i:= 1 to n-1 do
3     begin
4         start(fase i+1);
5         for j:= 1 to i+1 do
6             begin
7                 start(extensão j);
8             end;
9         encontrar_caminho_da_raiz(S[j..i]);
10        if (não existe S(i+1)) then
11            adiciona_caracter(S(i+1));
12    end;
```

Após o segundo *for*, a ideia é encontrar o fim do caminho com etiqueta $S[j..i]$ na árvore corrente, partindo da raiz. Se necessário, então estende-se o caminho ao adicionar

o carácter $S(i+1)$, assegurando assim que a sequência $S[j..i+1]$ está na árvore.

Para transformar esta ideia num algoritmo mais concreto, é necessário introduzir a noção de extensão de um sufixo. Seja β um sufixo de $S[1..i]$, na extensão j, quando o algoritmo encontra o fim de β na árvore corrente, estende β para garantir que $\beta S(i+1)$ se encontra na árvore. Este processo segue as seguintes regras [Gus97].

Definição 2.1 (Primeira Regra das Extensões ou PRE)**.** Na árvore corrente, o caminho com início na raiz e com etiqueta β, termina numa folha. Para actualizar a árvore, o carácter $S(i+1)$ é adicionado no final da etiqueta da aresta respectiva a essa folha.

Definição 2.2 (Segunda Regra das Extensões ou SRE)**.** Nenhum caminho a partir do fim da cadeia de caracteres β, começa com $S(i+1)$, mas pelo menos um caminho com outra etiqueta, continua a partir do fim de β. Neste caso, uma nova aresta para uma folha tem de ser criada, com etiqueta $S(i+1)$, e tem de estar ligada a um novo nó criado a meio da aresta única que existia previamente.

Definição 2.3 (Terceira Regra das Extensões ou TRE)**.** Existe um caminho do final da cadeia de caracteres β que começa com $S(i+1)$. Desta feita, a sequência $\beta S(i+1)$ já se encontra na árvore, pelo que nada é feito.

Na Figura 2.4, podemos ver um exemplo de algumas destas regras. No terceiro sufixo (na folha com número 3), aplica-se a Definição 2.1, adicionando-se o carácter 'c' à etiqueta da aresta. Posteriormente, a Definição 2.2 pode verificar-se no sufixo dois e cinco da árvore após a extensão, onde é criado um nó interno. Esse nó expande duas novas arestas para duas folhas, com o novo carácter 'c' (também ele um novo sufixo) e o resto do segundo sufixo, representado anteriormente numa só aresta com "apra".

Usando as regras das extensões referidas anteriormente, uma vez que o fim de um sufixo β de $S[1..i]$ foi encontrado na árvore, o tempo necessário para executar as regras de extensão é constante [Gus97]. O ponto chave ao implementar este algoritmo é saber como encontrar o final dos $i+1$ sufixos de $S[1..i]$.

Numa primeira abordagem, podíamos encontrar o final de qualquer sufixo β com um tempo de ordem $O(|\beta|)$, percorrendo os caminhos a partir da raiz. Com esta aproximação, a extensão j da fase $i+1$ teria um tempo de ordem $O(i+1-j)$, T_{i+1} poderia ser criada a partir de T_i com um tempo $O(i^2)$ e deste modo, T_n seria criada em $O(n^3)$, o que não é um tempo razoável.

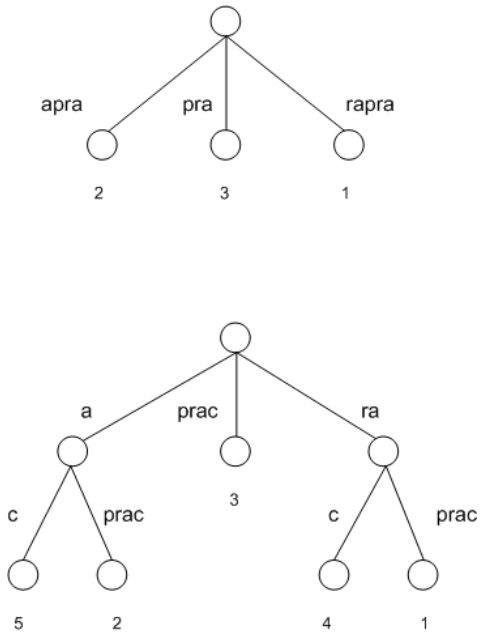

Figura 2.4: Árvore de sufixos implícita de "rapra", estendida após a adição de 'c'

Este tempo de ordem cúbica vai ser reduzido para um algoritmo de tempo linear, através de algumas observações e técnicas de implementação.

2.2.3 Primeira Técnica - *Suffix Links*

Definição 2.4 (*Path-Label*)**.** A etiqueta de um caminho da raiz até um nó, é a concatenação das sub-cadeias que constituem as etiquetas de cada aresta, presentes nesse caminho. A *path-label* de um nó é a etiqueta de todo o caminho da raiz ao próprio nó.

Definição 2.5 (*String Depth*)**.** A *string depth* de um nó, é o número de caracteres em que consiste a sua *path-label*.

Definição 2.6 (*Suffix Link*)**.** Seja xα uma sequência de caracteres arbitrária, onde x denota um único carácter e α uma sub-cadeia de S (possivelmente vazia). Para um nó interno v, de uma árvore com *path-label* xα, existe sempre outro nó $s(v)$ com *path-label* α. Desta forma, o apontador de v para $s(v)$ é chamado de *suffix link* [Gus97].

Analisando a Figura 2.3, nomeadamente a árvore de sufixos de cima, se v for o nó com *path-label* "ra" e $s(v)$ o nó cuja *path-label* é o carácter 'a', então existe um *suffix link*

de v para $s(v)$. No caso em que α é vazio, qualquer nó interno com *path-label* $x\alpha$ aponta para a raiz. O único nó que pode não ter um *suffix link* é a raiz.

Cada nó criado no algoritmo de Ukkonen, terá um *suffix link* a partir de si próprio, após a próxima extensão. Em qualquer árvore de sufixos implícita, se o nó interno v tiver *path-label* $x\alpha$, então existe um nó $s(v)$ com *path-label* α.

2.2.3.1 Seguir *Suffix Links*

Na fase $i+1$ do algoritmo, na extensão j é localizado o sufixo $S[j..i]$ da sequência $S[1..i]$, com j a variar entre 1 e $i+1$. De uma forma algo simplista, esta localização seria obtida comparando $S[j..i]$, com as etiquetas de todas as arestas do caminho a partir da raiz. Os *suffix links* podem ser usados para encurtar caminho em todas as n extensões. As duas primeiras extensões de cada fase serão explicadas de seguida, de modo a servirem como base para a explicação das seguintes.

O final da sequência $S[1..i]$ tem de se situar numa folha de T_i, já que é a maior sequência representada na árvore. Assim, torna-se mais fácil encontrar o fim desse sufixo e a sua extensão é tratada segundo a PRE. Portanto, a primeira extensão de qualquer fase é caracterizada por tempo constante, já que o algoritmo tem um apontador para o final da cadeia de caracteres completa.

Seja $S[1..i] = x\alpha$, onde x é um único carácter e α é uma sub-cadeia. De seguida, o algoritmo terá que encontrar o final da sequência $S[2..i] = \alpha$ na árvore derivada de T_i. Se o nó v for a raiz, então para encontrar o fim de α o algoritmo apenas percorre a árvore, seguindo o caminho etiquetado com α. Mas se v for um nó interno, tem um *suffix link* para o nó $s(v)$. Como $s(v)$ tem um *path-label* que é prefixo de α, o fim de α tem de terminar numa sub-árvore de $s(v)$. Por isso, ao procurar pelo final de α, o algoritmo não necessita de percorrer todo o caminho desde a raiz, podendo começar simplesmente por $s(v)$.

Considere-se γ a etiqueta da aresta $(v,1)$. Para encontrar o final de α, percorre-se o caminho da folha um para o nó v, segue-se o *suffix link* de v para $s(v)$ e percorre-se o caminho (ou caminhos), a partir de $s(v)$, com etiqueta γ. O final desse caminho coincide com o final de α, actualizando-se a árvore através da respectiva regra das extensões.

Para estender a sequência de $S[j..i]$ para $S[j..i+1]$, com $j > 2$, então deve repetir-se a mesma ideia. Começando no final da sequência $S[j-1..i]$, percorre-se o caminho para a raiz ou para um nó interno v com *suffix link*. Supondo que se chega a um nó interno, atravessa-se o *suffix link* para $s(v)$, percorrendo de seguida o caminho γ (como para a segunda extensão) até ao final de $S[j..i]$. No final, realiza-se a extensão para $S[j..i+1]$ de acordo com a respectiva regra.

Existe apenas uma pequena diferença entre as extensões para $j > 2$ e as duas primeiras. O final da sequência $S[j-1..i]$, pode ser um nó que tem um *suffix link*. Nesse caso, este será atravessado. Mesmo quando a SRE se aplica à extensão $j-1$, se o pai do nó não é a raiz então já tem um *suffix link*, como é garantido na Definição 2.6. Portanto, na extensão j o algoritmo percorre no máximo uma aresta. A Figura 2.5 demonstra precisamente esta ideia, com o caminho percorrido entre folha, nó v, *suffix link*, nó $s(v)$, acabando na descida pela aresta que sai de $s(v)$, ao seguir o caminho com *path-label* γ.

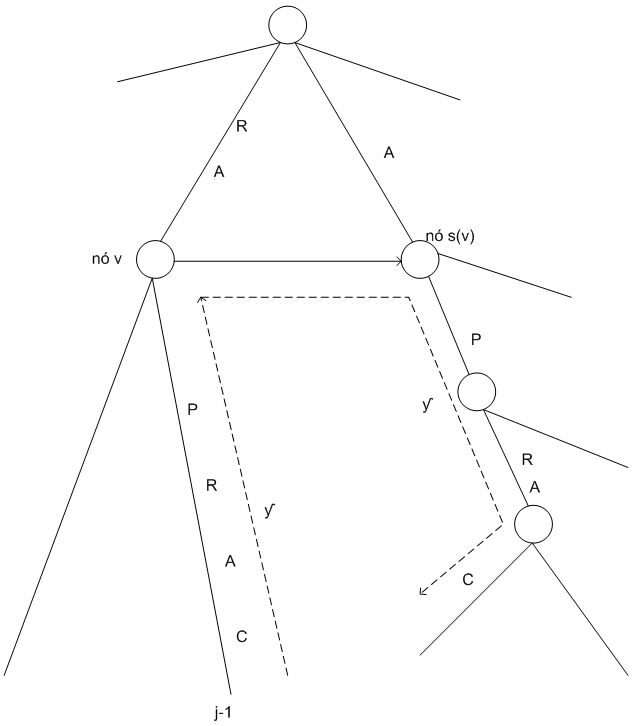

Figura 2.5: Extensão $j > 1$ na fase $i+1$

Juntando todas estas ideias e usando *suffix links*, a extensão $j \geq 2$ da fase $i+1$ pode ser implementada usando o Algoritmo de Extensão Única (AEU):

- Encontrar o primeiro nó v no (ou antes do) final da sequência $S[j-1..i]$ que, ou tem um *suffix link* ou é a raiz, o que obriga a percorrer, no máximo uma aresta.
- Seja γ a cadeia de caracteres entre v e o fim de $S[j-1..i]$. Se v não for a raiz, atravessa-se o *suffix link* de v para $s(v)$ para depois seguir o caminho etiquetado por γ. Se v for a raiz, então segue-se o caminho correspondente a $[j..i]$, como no algoritmo mais simplista.
- Com as regras das extensões, garantir que a sequência $S[j..i]S(i+1)$ está na árvore.

- Se um novo nó interno w foi criado na extensão $j-1$, então a cadeia de caracteres α tem de terminar num nó $s(w)$, ou seja, o nó destino do apontador que parte de w. Cria-se então o *suffix link* $(w, s(w))$ de w para $s(w)$.

Assumindo que o algoritmo mantém um apontador para a sequência completa $S[1..i]$, a primeira extensão da fase $i+1$ não necessita de percorrer quaisquer caminhos. Adicionalmente, a primeira extensão da fase $i+1$ aplica sempre a PRE.

Apesar dos *suffix links* serem uma ajuda bastante valiosa para aumentar a eficiência do algoritmo, não conseguem melhorar o tempo no pior caso. Desta forma, será introduzida de seguida a segunda técnica, que com os *suffix links* reduz o pior caso para tempo de ordem quadrática, $O(n^2)$.

2.2.4 Segunda Técnica - Saltar e Contar

No segundo passo da extensão $j+1$, o algoritmo percorre, a partir de $s(v)$, um caminho com etiqueta γ. Uma implementação directa desta operação, demora um tempo proporcional a $|\gamma|$. Mas com a técnica de saltar e contar, o tempo é proporcional ao número de nós existentes nesse caminho.

Seja g o tamanho de γ, g' o número de caracteres de uma aresta e seja o primeiro carácter de γ, igual ao primeiro carácter de uma das arestas que saem de $s(v)$. Caso $g' < g$, então o algoritmo não necessita de considerar mais caracteres dessa aresta, pois é impossível γ estar completamente representada nessa aresta. Deste modo, atribui-se a g o valor da subtracção $g-g'$ e a uma variável h o valor $g'+1$. Nas arestas resultantes, procura-se aquela cujo primeiro carácter da etiqueta coincida com o h-ésimo carácter de γ, tal como foi feito anteriormente para as arestas de $s(v)$.

Em geral, quando o algoritmo identifica a próxima aresta no caminho, compara o valor corrente de g com o número de caracteres g' dessa mesma aresta. Quando o g é pelo menos tão grande como g', o algoritmo passa para o nó no final dessa aresta, define g como $g-g'$ e h como $h+g'$, e encontra a aresta cujo primeiro carácter é o h-ésimo carácter de γ, repetindo esta mesma iteração de seguida. Quando se chega a uma aresta onde g é menor ou igual a g', então o algoritmo passa para o carácter g na aresta e pára, assegurando que o caminho γ de $s(v)$ termina na aresta exactamente g caracteres abaixo da sua etiqueta. Desta forma, não é necessário percorrer todos os caracteres, efectuando-se saltos entre os nós.

Na Figura 2.6, adaptada de Gusfield [Gus97], na fase $i+1$, a sub-cadeia γ tem tamanho dez. Existe uma cópia de γ a sair do nó $s(v)$, cujo final se encontra no terceiro carácter da última aresta. Neste caso faz-se a execução de quatro saltos (*suffix link* e os três nós seguintes), até se chegar ao último carácter, que neste caso é 'y'.

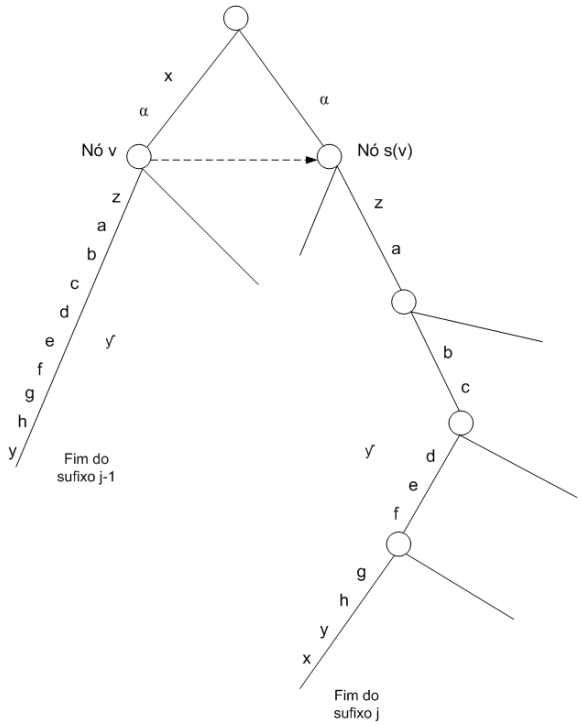

Figura 2.6: Técnica de saltar e contar

Esta técnica faz com que a implementação da operação, de descer de um nó para o próximo, fique com tempo constante, $O(1)$. Portanto, o tempo total para percorrer um caminho é proporcional ao número de nós, em vez do número de caracteres.

Usando esta técnica de saltar e contar, qualquer fase do algoritmo de Ukkonen tem tempo de ordem $O(n)$. Como o algoritmo de Ukkonen tem n fases, pode ser implementado com *suffix links*, de modo a obter um tempo de ordem quadrática, $O(n^2)$. Note-se que este tempo foi obtido tendo em conta as suas n fases e o tempo $O(n)$ de cada uma delas, multiplicando-se esses termos. No entanto, veremos nas próximas secções que estes conceitos são essenciais, pois juntamente com mais algumas técnicas, é possível apresentar ainda melhores tempos.

2.2.5 Etiquetas das Arestas

As etiquetas das arestas de uma árvore de sufixos podem conter mais do que $O(n)$ caracteres no total. Como o tempo de funcionamento do algoritmo é pelo menos tão grande como o tamanho do seu *output*, com um número de caracteres desta ordem, a árvore pode requerer $O(n^2)$ de espaço. Sendo assim, esta é uma barreira que torna o tempo $O(n)$ impossível. Se considerarmos uma sequência de caracteres como o abecedário, todos os sufixos começam com um carácter distinto, o que faz com que saiam

26 arestas da raiz, cada uma etiquetada por um sufixo diferente. Com tal número, seriam representados um total de $26 \times (27/2)$ caracteres. Para cadeias de caracteres maiores que o tamanho do alfabeto, alguns caracteres serão repetidos, mas mesmo assim consegue-se construir sequências de caracteres de tamanho arbitrário n, de modo a que as etiquetas resultantes tenham mais que $O(n)$ caracteres ao todo.

Um algoritmo com complexidade $O(n)$, para construir uma árvore de sufixos, requer um esquema alternativo para representar as etiquetas das arestas. Em vez de cada aresta ter associada uma sub-cadeia de caracteres explícita, terá um par de índices a especificar a posição inicial e final dessa mesma sub-cadeia em S, como se pode ver na Figura 2.7, adaptada de Gusfield [Gus97], para a sequência "pqrstupqrghi".

Como o algoritmo tem uma cópia de S, pode localizar qualquer carácter em S dada a sua posição, em tempo constante. Assim, podemos descrever qualquer algoritmo de árvores de sufixos como se as etiquetas das arestas fossem explícitas, mas podem ser implementados com apenas dois símbolos por etiqueta.

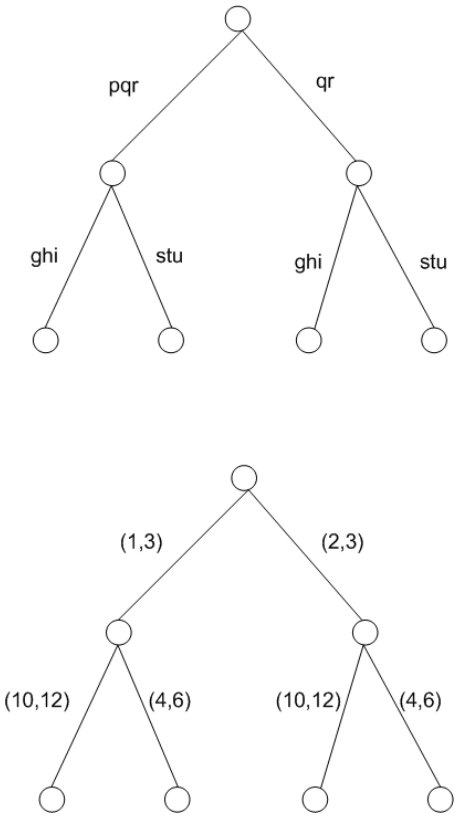

Figura 2.7: Fragmento da árvore de sufixos, sem e com etiquetas comprimidas

Quando o algoritmo de Ukkonen faz a pesquisa por uma sequência numa aresta, basta usar o par de índices da etiqueta para retirar os caracteres necessários de S.

As regras das extensões também são facilmente implementadas com esta ideia. Quando a segunda regra se aplica a uma fase $i+1$, etiqueta-se a nova aresta com o par $(i+1, i+1)$. No caso de se aplicar a PRE, muda-se o par nessa aresta de (a,b) para $(a,b+1)$, ou seja, $(a,i+1)$.

Ao usar estes índices, as arestas passam a conter apenas dois números nas suas etiquetas. Como o número de arestas é no máximo $2n-1$, as árvores de sufixos necessitam apenas de $O(n)$ símbolos, o que significa também espaço $O(n)$. Desta forma, é ultrapassada uma barreira para a construção da árvore de sufixos em tempo linear.

2.2.6 Terceira Técnica - Terceira Regra das Extensões

Em qualquer fase do algoritmo de Ukkonen, se a TRE se aplicar na extensão j, também se aplicará nas futuras extensões até ao final da fase em questão. Ou seja, se numa extensão j se aplicar a TRE, significa que o carácter já se encontra no local em que era suposto ser inserido. Assim, nas extensões seguintes $j+1, j+2, .., i+1$, esse carácter também se encontrará já na árvore, pois nessas extensões a ideia seria só completar o sufixo, que neste caso já se encontra completo.

Quando esta terceira regra se aplica, não é necessário trabalho extra por parte do algoritmo, dado que o sufixo de interesse já se encontra na árvore. Além disso, é necessário adicionar à árvore um novo *suffix link*, apenas quando se aplica a segunda regra. Estes dados levam a outra técnica para melhorar este algoritmo.

Essa técnica consiste em terminar a fase $i+1$, na primeira vez que a TRE é usada. Se isto acontecer na extensão j, então não é necessário encontrar explicitamente o fim de qualquer sequência $S[k..i]$, para $k > j$, pois já se encontram na árvore. As extensões na fase $i+1$ que se encontrem terminadas após a execução da TRE, dizem-se terminadas implicitamente. As extensões opostas, onde o final de $S[j..i]$ é explicitamente encontrado, são denominadas por extensões explícitas.

Apesar de ser uma boa heurística, não é garantido que ajude no pior caso, pelo que é necessária outra técnica para oferecer as garantias necessárias.

2.2.7 Quarta Técnica - Relativa às Folhas

Se em algum ponto do algoritmo de Ukkonen, uma folha é criada e etiquetada por j (sufixo a começar na posição j de S), então a folha continuará sempre a ser uma folha nas árvores que são criadas de seguida, durante a execução do algoritmo. Isso deve-se ao algoritmo não ter nenhum mecanismo para estender uma aresta, para além da sua correspondente folha. Ou seja, uma vez que existe uma folha etiquetada por j, a PRE será sempre aplicada à extensão j, em qualquer uma das fases seguintes, não sendo

por isso criado qualquer nó adicional.

A primeira folha é criada na fase número um, portanto em qualquer fase i existe uma sequência inicial de extensões consecutivas, onde se aplicam a PRE e a SRE.

Seja j_i a última extensão da sequência. Como cada aplicação da segunda regra cria uma nova folha, segue do parágrafo anterior que $j_i \leq j_{i+1}$. Ou seja, a sequência inicial das extensões onde a PRE e SRE se aplicam, não podem diminuir nas fases que se seguem. Isto sugere uma nova técnica, tal que na fase $i+1$ se evitam todas as extensões explícitas de 1 a j_i. Se forem tratadas como extensões implícitas, é apenas necessário tempo constante.

Na fase $i+1$, quando a aresta de uma folha é criada pela primeira vez, seria etiquetada com a sub-cadeia $S[a..i+1]$. Em vez de etiquetar a aresta com os índices $(a, i+1)$, devemos antes colocar os índices (a,e), onde e é um símbolo a denotar o final da fase, ou seja, este símbolo é um índice global afectado com o valor $i+1$, uma vez em cada fase. Na fase $i+1$, como o algoritmo "sabe" que a PRE se aplicará às extensões 1 a j_i, não é necessário nenhum trabalho adicional para implementar essas j_i extensões. Em vez disso, apenas demora tempo constante a incrementar a variável e, realizando então o trabalho adicional para algumas extensões, a começar pela extensão j_{i+1}.

2.2.8 Conclusões Sobre o Algoritmo

Com estas duas últimas técnicas, as extensões explícitas na fase $i+1$, só são necessárias a partir da extensão j_{i+1}, até à primeira extensão onde a TRE se consiga aplicar, ou até a extensão $i+1$ terminar. Todas as outras são feitas de forma implícita. Portanto, a fase $i+1$ será implementada através do algoritmo AFU (Algoritmo de Fase Única) [Gus97], mais propriamente com os três passos seguintes:

- Incrementar o índice e para $i+1$. Pela quarta técnica, isto implementa correctamente todas as extensões implícitas de i a j_i.
- Computar explicitamente as extensões sucessivas (usando o AEU), começando na j_{i+1} até chegar à primeira extensão k, onde a TRE se aplica, ou até todas as extensões terminarem nesta fase. Pela terceira técnica, esta ideia implementa correctamente todas as extensões implícitas adicionais, de $k+1$ até $i+1$.
- Atribuir a j_{i+1}, o valor $k-1$ para a próxima fase.

O terceiro passo do algoritmo define o valor de j_{i+1}, porque a sequência inicial de extensões, onde a PRE e a SRE se aplicam, tem de terminar no ponto em que a TRE se aplica pela primeira vez.

A chave deste algoritmo está na fase $i+2$, ser capaz de computar extensões explícitas com k, onde k era a última extensão explícita computada na fase $i+1$. Portanto, duas fases consecutivas partilham, no máximo, um índice (k) onde uma extensão explícita é executada. Além disso, a fase $i+1$ termina a saber onde a sequência $S[k..i+1]$ acaba, pelo que a extensão k repetida na fase $i+2$ pode executar a regra de extensão para k, sem qualquer passagem por *suffix links* ou sem passar nós à frente. Isso significa que a primeira extensão explícita em qualquer fase, apenas demora tempo constante. Desta forma, é possível demonstrar o seguinte resultado:

- Usando *suffix links* e as restantes técnicas enunciadas, o algoritmo de Ukkonen constrói árvores de sufixos implícitas de T_i a T_n em tempo linear, $O(n)$ [Gus97].

2.2.9 Construção da Árvore de Sufixos Explícita

A árvore de sufixos implícita T_n, pode ser convertida numa árvore de sufixos explícita. Primeiro, adiciona-se um terminador ao final de S, permitindo que o algoritmo de Ukkonen continue para esse carácter. Desta forma, nenhum sufixo é agora prefixo de outro sufixo, pelo que a execução do algoritmo resulta numa árvore de sufixos implícita, em que cada sufixo termina numa folha.

A outra alteração necessária, trata-se de substituir cada índice e em cada folha por n, pois simbolizam o final dos sufixos da cadeia de caracteres. Isto é conseguido ao percorrer em tempo linear a árvore, $O(n)$, visitando todos as arestas das folhas. No final, o resultado é uma árvore de sufixos completa, construída (com todos os *suffix links*) em $O(n)$ [Gus97].

2.3 Árvores de Sufixos Generalizadas

Até ao momento neste capítulo, as árvores de sufixos foram sempre definidas em relação a uma cadeia de caracteres única. No entanto, um léxico armazena um conjunto de sequências de caracteres, pelo que a estrutura abordada até agora não é uma opção válida para o representar. Assim, surgem as árvores de sufixos generalizadas, capazes de representar sufixos de um conjunto de sequências de caracteres $S_1, S_2, ..., S_z$.

Uma maneira simples de construir uma árvore de sufixos generalizada, é através da concatenação a todas as cadeias de caracteres, de um terminador distinto entre todas. De seguida, é feita a concatenação de todas essas cadeias, para construir a árvore com toda essa sequência final, resultante da concatenação.

A árvore de sufixos obtida, tem uma folha para cada sufixo da cadeia final e é construída em tempo proporcional à soma de todos os comprimentos das cadeias constituintes. A numeração das folhas pode ser convertida para dois números, um a identificar uma sequência de caracteres S_i e outro a posição inicial de S_i.

O método anterior também pode ser simulado sem recorrer à concatenação de todas as cadeias de caracteres do conjunto. Se tivermos duas sequências de caracteres S_1 e S_2 distintas, pode-se fazer o seguinte. Em primeiro lugar, constrói-se a árvore de sufixos para S_1, recorrendo ao algoritmo de Ukkonen. De seguida, começando pela raiz da árvore, emparelha-se S_2 com qualquer caminho até ocorrer uma não correspondência. Suponhamos que os primeiros i caracteres de S_2 emparelham. A árvore neste ponto codifica todos os sufixos de S_1 e, implicitamente, todos os sufixos da sequência $S_2[1..i]$. Essencialmente, as primeiras i fases do algoritmo de Ukkonen para S_2 foram executadas sobre a árvore para S_1. Portanto, o algoritmo de Ukkonen deve continuar para S_2, a partir da fase $i+1$. Quando S_2 estiver completamente processada, a árvore armazenará todos os sufixos, tanto de S_1 como de S_2.

Repetindo estes passos para todas as cadeias de caracteres do conjunto, cria-se a árvore de sufixos generalizada, em tempo proporcional à soma dos tamanhos de todas as cadeias a representar.

A abordagem tomada para a implementação deste sistema, centrou-se na técnica da concatenação de todas as cadeias de caracteres entre si.

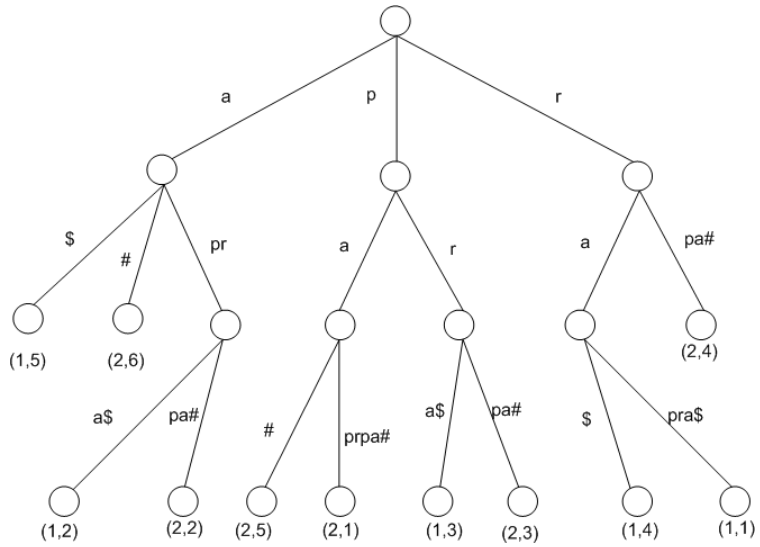

Figura 2.8: Árvore de sufixos generalizada para as sequências "rapra" e "paprpa"

3

Trabalho Relacionado

Nesta secção, fazemos uma breve introdução a alguns protótipos e a algumas ferramentas importantes, relacionadas com o âmbito desta dissertação. Começamos por introduzir o método de *lookup*, abordando a sua funcionalidade e a forma como utiliza um léxico. De seguida, apresentamos de uma forma geral, os mecanismos de alinhamento e de extracção de traduções, que podem utilizar o sistema LEXMAN. Por fim, introduzimos um protótipo para árvores de sufixos bilingues, que utiliza as árvores de sufixos generalizadas e que contém algumas funcionalidades semelhantes às apresentadas neste capítulo.

3.1 Método *Lookup*

A implementação desta operação é baseada em *arrays* de sufixos, pelo que de seguida introduzimos esta estrutura de dados. Estas estruturas também fazem parte da categoria de índices de texto completo, à semelhança das árvores de sufixos.

3.1.1 *Array* de Sufixos

Um *array* de sufixos [MM90, GBYS92] é uma permutação de todos os sufixos de uma sequência de caracteres S, de forma a que estejam lexicograficamente ordenados. A ordem lexicográfica, caracteriza-se por ordenar as sequências de caracteres com base no abecedário, utilizando o carácter mais à esquerda e desempatando com os seguintes.

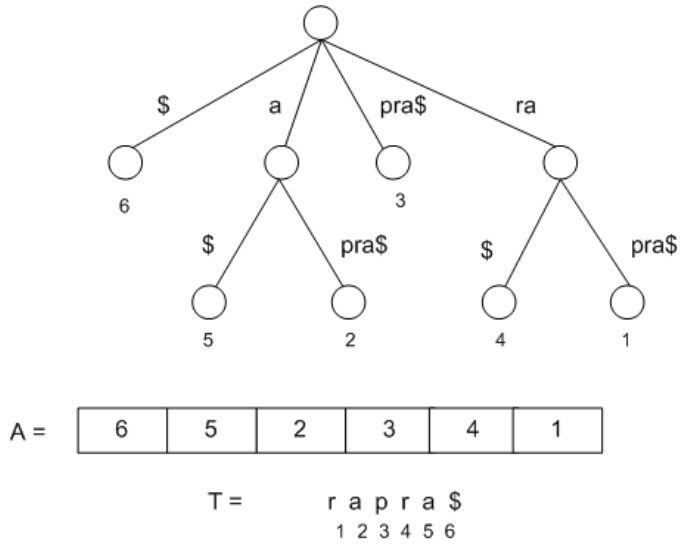

Figura 3.1: *Array* de sufixos e respectiva árvore, para a sequência "rapra"

Assumindo que os filhos dos nós da árvore de sufixos se encontram ordenados lexicograficamente da esquerda para a direita, o *array* de sufixos pode ser obtido ao coleccionar as folhas da árvore por essa ordem. No entanto, é bastante mais útil construí-los directamente, do que seguir este método.

Qualquer algoritmo de ordenação pode ser usado para colocar os sufixos na ordem desejada, mas pode ter custos demasiado elevados se existirem sub-cadeias repetidas na própria cadeia (ou cadeias) de caracteres armazenadas. Na prática, os melhores algoritmos para este efeito não obtém um tempo linear [LS07, IT99, MF04, SS07], precisamente devido a esta condicionante das sub-cadeias.

Como se pode ver na Figura 3.1, cada sub-árvore da árvore de sufixos corresponde a um intervalo do *array*, que contém as suas folhas. De notar que o terminador universal $, corresponde à primeira posição do *array*, pois é o sufixo que, seguindo uma ordem lexicográfica, é sempre menor que todos os outros.

O *array* de sufixos juntamente com a cadeia de caracteres armazenada, contém informação suficiente para pesquisar padrões de forma eficiente.

O resultado da pesquisa numa árvore de sufixos é uma sub-árvore. De forma semelhante, o resultado da pesquisa num *array* de sufixos, é um intervalo, pois os sufixos prefixados pelo mesmo prefixo P, estão lexicograficamente contíguos num *array* A.

Portanto, é possível pesquisar pelo intervalo de A, que contém os sufixos prefixados por P, através de duas pesquisas binárias em A. A primeira determina a posição inicial *sp*, para os sufixos lexicograficamente superiores ou iguais a P. A segunda determina a posição final *ep*, para os sufixos que começam por P. O resultado final é o intervalo

delimitado por $[sp, ep]$ [NM07].

3.1.2 Léxico e Algoritmo de *Lookup*

O léxico é gerido por uma base de dados relacional constituída por três tabelas. A primeira contém os termos de uma das línguas representadas, a segunda os termos na outra língua e a terceira armazena as correspondências entre estes mesmos termos.

Periodicamente, o léxico é exportado da base de dados para um conjunto de ficheiros que, posteriormente, são utilizados pelo alinhador. O número de ficheiros é igual ao número de tabelas na base de dados, ou seja, três. O seu formato é muito simples, estando os termos um por cada linha, seguindo uma ordem lexicográfica.

Um segmento de uma sequência de caracteres é representado como um par (f, l), correspondente à posição do primeiro e último carácter respectivamente. Para cada par de termos no léxico, pretende obter-se um par de vectores de ocorrências, que será usado para conseguir as correspondências entre dois textos.

O vector O_x tem todas as ocorrências do termo t_x no texto X, ordenadas pelas posições onde são encontradas no texto. O mesmo se passa com O_y, respectivamente ao termo t_y e ao texto Y. O problema é simétrico e é usado o mesmo procedimento para obter cada um destes vectores. O nome desse procedimento é *lookup*, mostrado na Figura 3.2, retirada de [Gom09].

```
function LOOKUP(t, a)
    L ← {}                                          ▷ the list of occurrence lists
    t ← first term in t
    s ← first suffix in a
    repeat
        if t is prefix of s then
            ℓ ← length of t
            o_t ← empty list                        ▷ the list of occurrences of term t
            s_t ← s
            repeat
                pos ← offset of s_t in the text
                append (pos, pos + ℓ) to o_t ▷ add occurrence of t at the position pos to the list
                s_t ← next suffix in a
            until lcp(s_k, s_{k-1}) < ℓ or we have run through all suffixes in a
            add o_t to L
            t ← the next term in t
        else if t is lexicographically lower than s then
            t ← the next term in t
        else                                        ▷ t is lexicographically higher than s
            s ← the next suffix in a
        end if
    until we have run through all terms of t or all suffixes of a
    return L
end function
```

Figura 3.2: Método *Lookup*

Como o léxico está representado em três tabelas, temos uma tabela tb_x e outra tb_y, com os termos das duas línguas, onde $tb_x = (t_{x1}, t_{x2}, ...)$ e $tb_y = (t_{y1}, t_{y2}, ...)$. Adicionalmente existe a tabela tb_p, com as ligações de correspondência entre termos, com estes representados apenas por identificadores.

A função de *lookup* recebe dois argumentos. O primeiro é uma lista ordenada de termos (t) e o segundo é o texto (a). A lista é acedida, elemento a elemento, de forma sequencial. O texto está definido num *array* de sufixos, criado especificamente para o armazenar. Desta forma, tem-se inicialmente duas listas ordenadas de cadeias de caracteres. O procedimento retorna um vector de ocorrências, para cada termo da lista que ocorra pelo menos uma vez no texto. Por exemplo, pode-se executar *lookup* (tb_x, X), para encontrar termos da tabela tb_x, no texto X.

A ideia geral do algoritmo é comparar os primeiros elementos das duas listas. Se o termo for prefixo do primeiro sufixo do *array* com o texto, então foi encontrada uma ocorrência desse termo no texto. Caso não seja prefixo desse sufixo, existem duas possibilidades: ou o termo é lexicograficamente inferior ao sufixo, onde se passa para o termo seguinte, ou então o sufixo é lexicograficamente inferior ao termo, passando-se para o sufixo seguinte, já que este não tem qualquer possibilidade de se encontrar no *array*. Como cada comparação entre termo e sufixo, demora um tempo proporcional ao prefixo comum entre os dois, este método acaba por ser ineficiente.

Para melhorar a performance, quando é criado o ficheiro com a lista de termos para ser usado no método *lookup*, é calculado o maior prefixo comum (mpc) entre cada termo e o seu anterior. O mpc(A,B) é o tamanho do maior prefixo comum entre as cadeias de caracteres A e B. Os mpcs entre sufixos adjacentes num *array* P, são denotados por outro *array* H, tal que $H[k] = mpc(T_{p[k-1]}, T_{p[k]})$ [KLA+01], com k a tomar valores de 1 ao tamanho do *array* P, e com T a ser um texto. Tendo estas duas sequências de caracteres "abcdefg" e "abe", o maior prefixo comum das duas sequências é "ab".

De notar, que quando o *array* de sufixos é criado, o vector de mpc dos sufixos também é calculado. Assim, o *lookup* sabe quantos caracteres no inicio do termo são comuns ao anterior e, com o uso do mpc dos termos e dos sufixos, reduz-se o número de comparações necessárias entre caracteres.

No código da Figura 3.2, considera-se que existe um *array* de sufixos **a** para um texto T, com a informação do mpc e que a lista de termos **t** se encontra ordenada. Por cada termo t em **t**, a ideia é localizar todos os sufixos em a que tenham t como prefixo.

Este algoritmo tem tempo linear, $O(tt + ttb)$, com tt a ser o tamanho do texto e ttb o tamanho da tabela, e não impõe tamanho limite aos termos.

3.2 Alinhamento de Textos Paralelos

Dois textos são considerados paralelos, se um deles for a tradução do outro. O alinhamento de dois textos paralelos consiste em dividir os textos num determinado número de segmentos, tal que o i-ésimo segmento de um texto corresponde ao i-ésimo segmento do outro. Considera-se um segmento, como uma parte do texto, ou seja, uma palavra, uma frase ou uma expressão composta por várias palavras e que não seja necessariamente uma frase completa.

Uma restrição inerente ao alinhamento, é a impossibilidade de existir correspondência entre segmentos cruzados, ao que se chama Restrição de Monotonicidade. Na Figura 3.3, retirada de Gomes [Gom09], é mostrado um exemplo de um alinhamento. Note-se que para o texto em português, as palavras "da", são usadas como "de a"

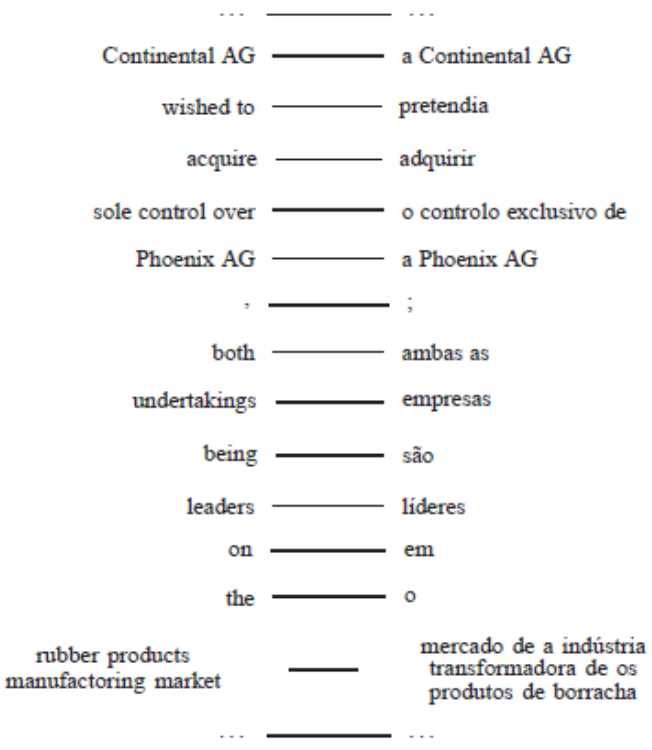

Figura 3.3: Alinhamento entre excertos de textos em Inglês e Português

Uma correspondência entre dois textos paralelos, é uma mapeamento entre segmentos dos dois textos. Neste caso, é possível existirem correspondências cruzadas, tal como correspondências descontíguas, que são aquelas que ocorrem entre um segmento ou vários segmentos descontíguos num dado texto, e entre outros segmentos

descontíguos noutro texto diferente. Na fFgura 3.4, retirada de Gomes[Gom09], pode ver-se o mesmo exemplo que na Figura 3.3, mas com as correspondências descontíguas. Caso haja regiões dos textos não cobertas por correspondências, então temos correspondência parcial. Caso todas estejam cobertas, temos uma situação de correspondência total.

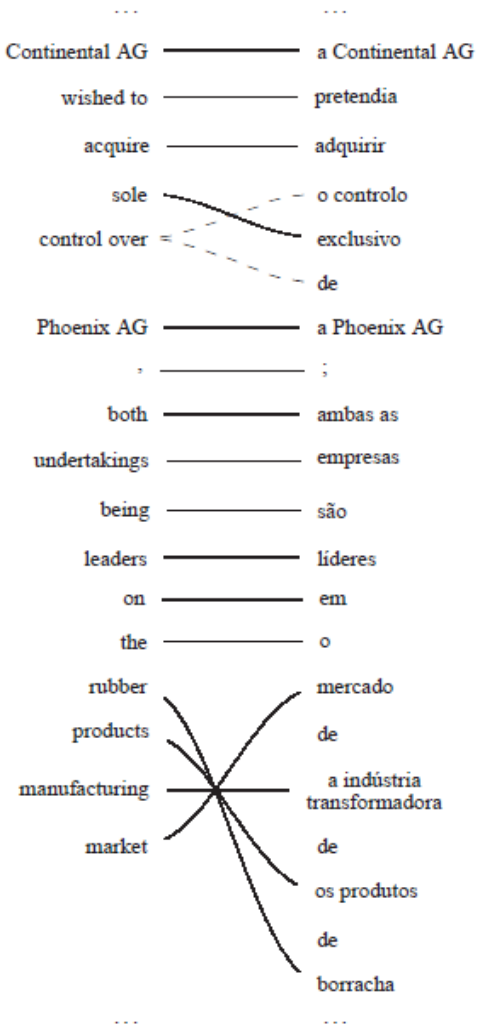

Figura 3.4: Correspondências entre as mesmas passagens da Figura 3.3

Na Figura 3.3 tem-se um exemplo de correspondência total, enquanto que na Figura 3.4 tem-se correspondência parcial, devido à falta de ligações entre alguns termos, nomeadamente o "de". Na Figura 3.4 ainda se encontram representadas ligações descontíguas, a tracejado.

Ao contrário da principal tendência do estado de arte em métodos de alinhamento, a implementação do protótipo de alinhamento de textos paralelos, é independente da

extracção de traduções. O alinhador usa apenas traduções de léxicos fornecidos antes da execução, ao contrário do PWA [AMHT00], GIZA++ [ON03] e SIMR [Mel99], que inferem um léxico em tempo real. Esta separação traz três importantes vantagens:

- Obtêm-se léxicos bilingues mais ricos, devido ao uso de métodos mais sofisticados para a sua extracção.
- Obtém-se controlo sobre o léxico, o que significa que podem-se remover entradas menos correctas, com maior facilidade.
- O alinhamento torna-se computacionalmente mais leve.

O número de correspondências fornecidas pelo método, depende quase exclusivamente do tamanho do léxico, o que significa que quanto maior for o léxico, mais correspondências correctas são detectadas.

Outra característica importante da implementação de Gomes [Gom09], está relacionada com a sua capacidade de abordar as várias línguas possíveis, de uma forma o mais geral possível. Os projectos anteriormente publicados no que toca a esta temática, dependiam bastante das frases [MW02], das palavras [ON03], da gramática da linguagem ou de cognatos [RDLM01]. Um cognato é uma palavra que derivou de outra comum, como é o exemplo de ”president” em Inglês e ”presidente” em Português, que derivam da palavra do Latim ”president”.

Estas abordagens podem obter excelentes resultados, mas dependem muito da língua usada. Pensando em vários exemplos, a gramática varia muito de língua para língua; os cognatos são muito frequentes entre português e espanhol, mas entre inglês e japonês, são muito raros; as palavras em chinês e japonês não são separadas por espaços. Estas divergências fazem com que os resultados possíveis possam divergir muito, estando muito dependentes das línguas utilizadas pelo processo.

O algoritmo de alinhamento definido estabelece correspondências entre segmentos ao nível da sub-frase e é capaz de usar conhecimento previamente adquirido, de modo a melhorar a sua precisão. A sua implementação é uma adaptação do algoritmo definido por Ricardo et al. [RDLM01].

3.3 Extracção de Traduções

O alinhador mencionado na secção anterior, alinha colecções de textos paralelos usando um léxico com termos e expressões, que tenham sido previamente extraídas, e usando pares de tradução aceites como segmentos correspondentes. Esta é mais uma prova de que o processo iterativo e cíclico mencionado anteriormente, é caracterizado por uma

ligação bastante forte entre cada uma das suas fases, pelo que a extracção de traduções é também um elemento chave.

No entanto, trocas ou omissões entre textos, podem impedir que o alinhamento se torne mais eficiente. Mesmo assim, o alinhamento é um excelente ponto de partida para a extracção de equivalentes de tradução, pois consegue identificar possíveis pares com facilidade, apesar de não ser aconselhável seguir o método à risca.

Analisando a Figura 3.4, pode verificar-se a relação "wished to" <=> "pretendia", entre outras. No entanto, a relação "being" <=> "são" pode não ser a mais correcta, pois a tradução de "being" mais precisa seria "sendo". Por isso, se a ligação "being" <=> "sendo" ocorrer mais vezes no texto, a ligação "being" <=> "são" poderá ser rejeitada. Desta forma, não é suficiente ligar simplesmente os termos entre si, com os dados que possam vir do alinhamento. Os problemas causados por desconhecimento de termos, erros, características específicas da linguagem ou ate traduções dúbias, têm de ser analisados de modo a reduzir os efeitos negativos no final.

A extracção de traduções utiliza um conjunto de medidas, combinadas por um esquema de votação, de modo a escalar os pares de tradução de acordo com a sua coesão interna, para submete-los a avaliação [ALG09]. A precisão obtida é bastante superior, quando comparada com outros trabalhos relacionados [Hje], equiparando-se aos melhores resultados obtidos na extracção ao nível da palavra [Mel00, RPLM00].

Com os textos paralelos alinhados, é muito mais simples associar termos. A ideia é juntar cada termo de um texto, com os seus segmentos adjacentes, obtendo de seguida as correspondências desses mesmos termos adjacentes. O interesse reside na captação de nova informação, portanto as correspondências de termos já conhecidas são posteriormente descartadas. O número de vezes que um dado termo é associado a outro, é a sua frequência de emparelhamento.

Um equivalente de tradução terá três propriedades básicas. Estas propriedades são usadas para calcular as medidas de avaliação mencionadas acima, que posteriormente servirão para analisar e possivelmente melhorar os resultados:

- frequência de emparelhamento;
- frequência do termo 1 no seu respectivo texto;
- frequência do termo 2 no seu respectivo texto.

Nos casos em que existem limitações no alinhamento, como termos fora de ordem, existem duas possibilidades. Caso não se conheça nada das linguagens, pode tentar-se todas as hipóteses de tradução possíveis, esperando que depois as estatísticas determinadas consigam ajudar a decidir qual a melhor solução, ou seja, a equivalência de

tradução mais provável. Caso se conheça algumas características das linguagens, pode tentar-se explorar essas mesmas características. Por exemplo, é natural que um dado segmento esteja delimitado por espaços em branco, o que é um padrão na maior parte das línguas ocidentais. Deste modo, podem definir-se segmentos, seguindo a lógica dos espaços em branco, tentando extrair informação com esses segmentos.

Analisando o exemplo seguinte, pode-se usar esta estratégia para definir a terceira equivalência como a mais provável, precisamente porque na primeira, segunda e quarta, ficaria um segmento relacionado com outro vazio. A segunda equivalência não seria a ideal, pois ”grave” seria equivalente a ”serious”, mas ”criminalidade” ficava sem termo para formar uma ligação de correspondência. A mesma ordem de ideias aplica-se a todas as outras, à excepção da terceira equivalência, onde se consegue ligar ”serious” com ”grave” e ”criminalidade” com ”crime”, sendo por isso a melhor opção.

”crime” <=> ”criminalidade grave”
”serious” <=> ”criminalidade grave”
”serious crime” <=> ”criminalidade grave”
”serious crime” <=> ”grave”

Como é necessário aceder aos termos de forma eficiente, assim como é necessário obter as variadas frequências, a implementação do extractor de traduções é baseada em *arrays* de sufixos. Estas estruturas oferecem mecanismos eficientes e importantes no acesso aos termos, tal como fornecem uma capacidade de cálculo, das frequências de cada termo, bastante eficiente.

3.4 Árvores de Sufixos Bilingues

Munteanu et. al.[MM02] apresentaram as árvores de sufixos bilingues (BST), uma estrutura de dados para explorar corpus (conjuntos de vários textos) paralelo.

Uma BST resulta do emparelhamento entre uma árvore de sufixos generalizada, representativa de uma língua mãe, com outra árvore de sufixos generalizada, representativa da língua objectivo. Duas sequências de caracteres dizem-se emparelhadas se as palavras que constituem uma das sequências, tenham ligações de correspondência com outras palavras da outra sequência e sejam traduções uma da outra.

Para realizar a operação de emparelhamento, todos os caminhos percorridos na pesquisa numa das árvores, serão percorridos de forma semelhante na outra árvore, até se dar um caso em que o emparelhamento já não se verifica. No processo a árvore representativa da língua objectivo, aumenta os seus dados com informação sobre os

alinhamentos, ou seja, os caminhos emparelhados, tornando-se assim numa árvore de sufixos bilingue.

A Figura 3.5, adaptada de Munteanu et al. [MM02], mostra uma BST, assim como o corpus que esta representa e o léxico bilingue com as ligações de correspondência entre termos. As arestas a tracejado indicam o final do alinhamento entre os textos do corpus, mostrando também o resto dos dados que não foi possível emparelhar.

Como existe uma correspondência de um para um, entre as sub-cadeias de caracteres no texto e os caminhos nas árvores de sufixos, a operação descrita irá descobrir todos os alinhamentos monótonos parciais definidos pelo léxico, a partir de todos os pares das sub-cadeias nos dois textos dados como *input*.

A operação de emparelhamento descrita, é semelhante à definida por Biegansky et al. [BRCR94], com a diferença no operador de emparelhamento e na estrutura da árvore resultante. No caso de Biegansky et al., as árvores são definidas sobre o mesmo alfabeto, havendo apenas a hipótese de verificar se duas sequências de caracteres combinam na totalidade. O resultado final é outra árvore de sufixos sobre o mesmo alfabeto, que codifica as subsequências comuns. No caso de Munteanu et al., as árvores são definidas sobre vários alfabetos e o emparelhamento é definido pelo léxico bilingue. Também se dá o caso, em que uma sequência numa árvore, pode emparelhar com várias sequências da outra árvore, sendo por isso uma operação muito mais complexa.

Com as características das BST, estas estruturas tornam-se úteis para explorar corpus comparável, com algoritmos capazes de criar corpus paralelo, de modo a descobrir traduções desconhecidas. Como as BST codificam informação sobre o alinhamento, a extracção de frases paralelas torna-se bastante fácil e resume-se a atravessar a árvore, com uma política de pesquisa por profundidade primeiro. A Figura 3.6, adaptada de Munteanu et al. [MM02], mostra uns alinhamentos que se conseguem extrair do exemplo da Figura 3.5. Na pesquisa feita pela árvore, as arestas que são de interesse, acabam por ser aquelas representadas a tracejado na Figura 3.5, pois marcam o final do alinhamento entre os constituintes do corpus.

Existem muitos outros projectos na área da linguística, mas optámos por mostrar um pouco deste último, por utilizar a mesma estrutura de dados e por estar relacionada com o âmbito da dissertação. Além disso, é possível utilizar as árvores de sufixos bilingues, para definir um léxico a partir de corpus comparável e não paralelo [MKMK09]. No entanto, não é um protótipo que represente um léxico, nem que tenha uma operação como a de cobertura.

Porém, poderia ser interessante utilizar um léxico bilingue representado pelo LEXMAN, para tentar-se obter mais informação, ou pelo menos, obter a mesma mas de forma mais eficiente.

No capítulo seguinte, apresentamos a implementação do LEXMAN, tal como uma explicação de algumas decisões de implementação tomadas.

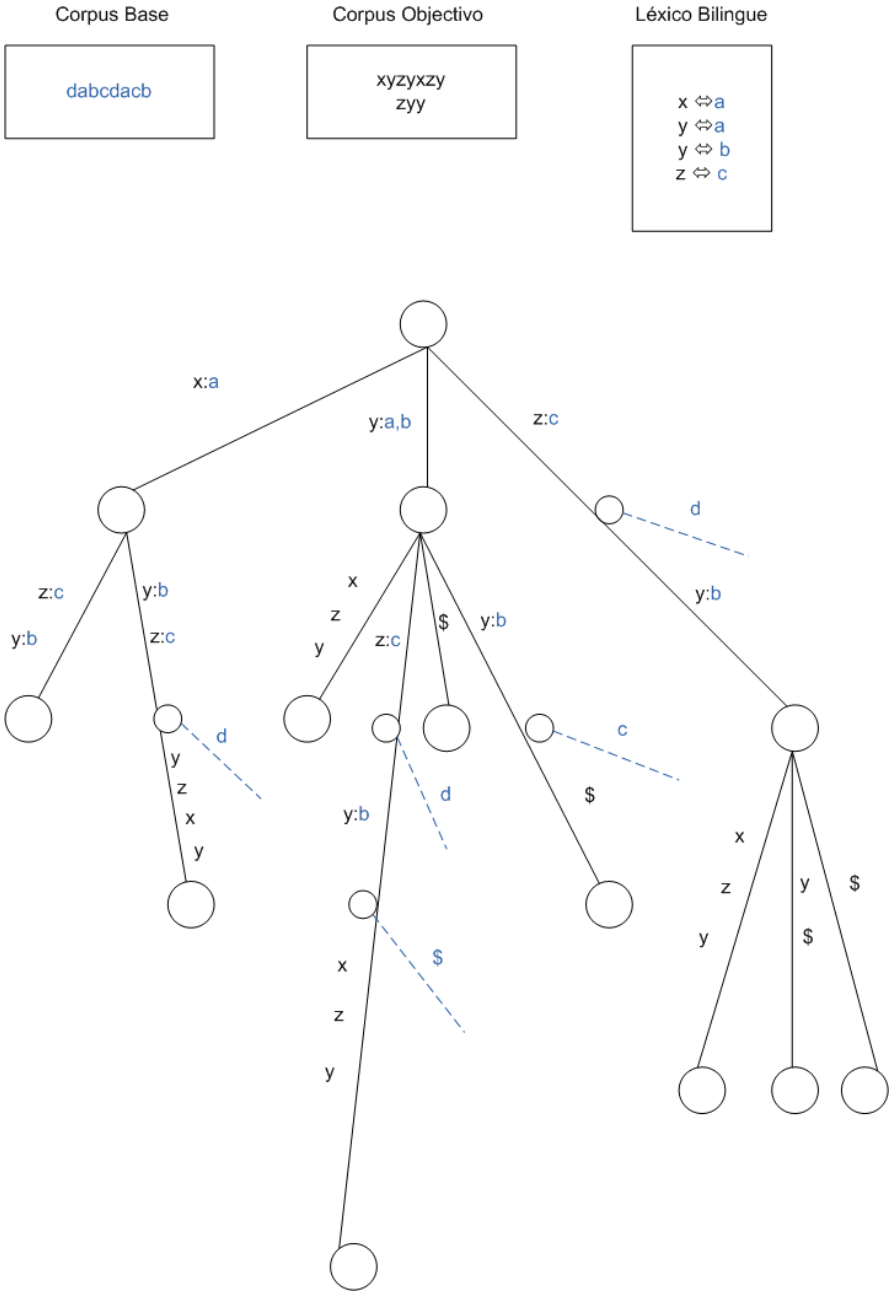

Figura 3.5: Árvore de Sufixos Bilingue

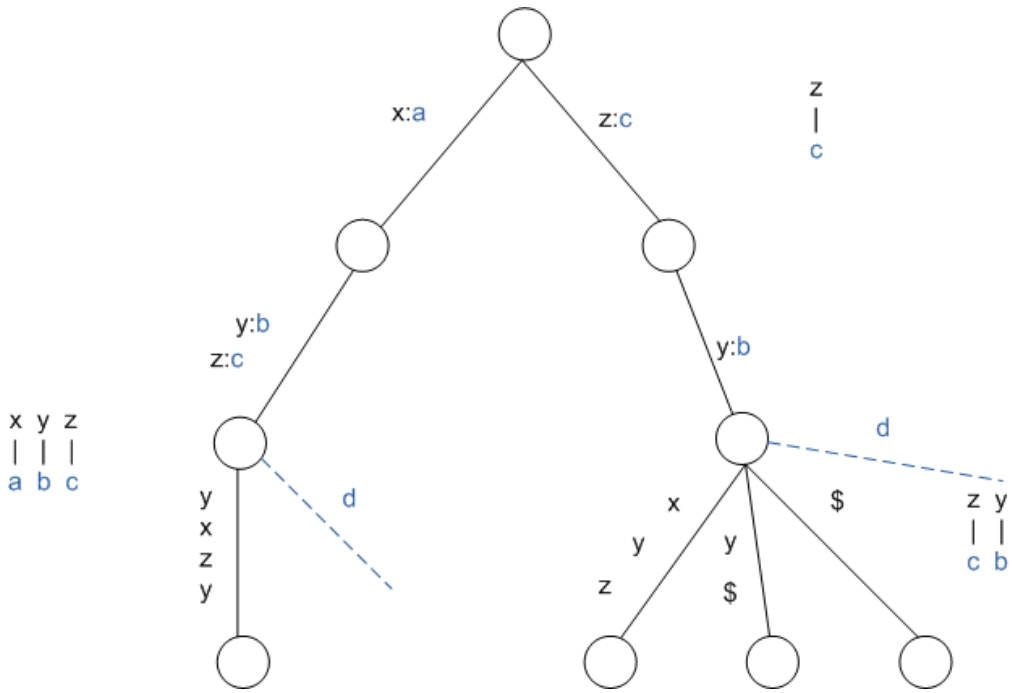

Figura 3.6: Exemplo de alinhamento com as BST

4

Implementação do Sistema LEXMAN

A implementação do LEXMAN utiliza duas árvores de sufixos generalizadas, uma para cada uma das línguas representadas. Durante a elaboração do projecto, as línguas utilizadas foram a inglesa e a portuguesa, pois são estas as línguas mais populares em Portugal. Os detalhes mais importantes da implementação, são explicados nas secções seguintes, começando pela opção da escolha das árvores de sufixos. Mostramos também as estruturas criadas ao nível da programação em C, que serão muito úteis para explicar como foram implementadas todas as operações.

As árvores estão implementadas ao nível do carácter, o que facilita a implementação e permite operações mais flexíveis. A alternativa seria implementar ao nível da palavra, mas isso levaria a uma menor flexibilidade para as operações pretendidas, nomeadamente no que toca às pesquisas na árvore e a seguir *suffix links*.

No Apêndice A apresentamos os comandos necessários para executar todas as operações, oferecidas pelo LEXMAN.

4.1 Árvores de Sufixos vs *Arrays* de Sufixos

Nos capítulos anteriores, foram introduzidos os dois índices de texto completo principais, as árvores de sufixos e os *arrays* de sufixos. Desde o início da dissertação, ficou estabelecido que a implementação seria em árvores de sufixos.

A opção de utilizar estas estruturas para representar o léxico bilingue, está relacionada com os *suffix links*, estrutura que não existe nos *arrays*. Quando é necessário fazer

pesquisas, é mais benéfico ter estes "atalhos", que permitem mudar de ramo da árvore, sem necessitar de voltar constantemente à raiz e voltar a descer. Principalmente na operação de cobertura, onde pode ser necessário efectuar pesquisas pelos vários segmentos de um termo. Como iremos mostrar, seguir *suffix links* é essencial para a eficiência do sistema. Num *array* teríamos de fazer várias pesquisas, para procurar por toda a informação que a cobertura necessita, enquanto que com as árvores basta apenas uma para resolver essa questão.

O léxico não exige muitos recursos de memória, pois não se espera que este tenha dimensões muito grandes. Portanto, o facto dos *suffix arrays* não utilizarem tanta memória, não se torna numa vantagem competitiva em relação às árvores de sufixos, neste caso específico. No Capítulo 5 obtemos resultados comparativos bastante esclarecedores da diferença entre as estruturas.

4.2 Estruturas

Para representar as árvores de sufixos e as ligações de correspondência entre termos, foram criadas algumas estruturas importantes ao nível da programação. As estruturas são úteis para armazenar os dados do sistema, assim como para ajudar a percorrer a árvore, sempre que é necessário. É importante apresentar estas estruturas, pois será bastante útil conhecer os seus campos e as suas finalidades, para explicar as operações implementadas. As estruturas são quatro e são as seguintes:

- Estrutura *Tree*
- Estrutura *Node*
- Estrutura *Point*
- Estrutura *List*

4.2.1 Estrutura *Tree*

Esta estrutura é a mais básica e tal como o nome indica, representa uma das árvores do sistema. A informação principal que esta estrutura contém, está relacionada com a sequência de caracteres armazenada, ou seja, a concatenação de todas as expressões no léxico. Além disso, a estrutura define o primeiro nó da árvore, a sua raiz.

Todos os termos presentes na árvores, são guardados no *array* de bytes T, sendo que o inteiro n representa o tamanho total da sequência de caracteres, representativa da concatenação de todos os termos. Todas as folhas apontarão para uma posição de T, de modo a saber qual o termo correspondente a essa folha.

Além da raiz da árvore, esta estrutura armazena ainda outro nó meramente auxiliar, para efeitos do algoritmo de Ukkonen. É um campo que representa o nó que foi criado anteriormente na árvore, sendo bastante importante na definição de *suffix links*. A estrutura, descrita em C, é a mostrada na Listagem 4.1.

Listing 4.1: Estrutura *Tree*

```
typedef unsigned char byte;
typedef struct tree *tree;
typedef unsigned int uint;

struct tree{
 byte* T;
 uint n;
 node root;
 node prevcreated;
};
```

4.2.2 Estrutura *Node*

De todas as estruturas criadas para este projecto, esta é a única independente de qualquer uma das outras, visto que se trata do elemento mais básico de uma árvore. A estrutura *Tree*, por exemplo, necessita de conhecer a estrutura *Node* para representar a raiz. A estrutura serve para representar um nó de uma árvore, nomeadamente toda a informação que o caracteriza.

Em primeiro lugar, um nó tem um pai, um filho e um irmão, todos também nós. O filho será sempre o descendente mais à esquerda, caso haja mais que um, enquanto o irmão, caso exista, encontra-se imediatamente à direita do nó na árvore.

De seguida, cada nó tem também a informação respectiva à sua etiqueta, nomeadamente a *sdep* e a *head*. O primeiro campo representa a *string-depth* do nó (Definição 2.5), ou seja, o tamanho da etiqueta que o caracteriza. O segundo campo trata-se de uma espécie de apontador para T, o *array* definido na estrutura anterior, indicativo da posição inicial do termo que o nó representa. Posteriormente, com a *path-label* do nó (Definição 2.4), consegue-se obter facilmente a cadeia de caracteres representada pelo nó em questão, em T.

Para o algoritmo de Ukkonen e posteriormente para as pesquisas, cada nó tem um *suffix link* associado que pode ser nulo. Para a operação de correspondência, cada nó tem também uma lista, com o objectivo de armazenar todos os nós da outra árvore com que forma uma ligação de correspondência. Deste modo, a lista tem todos os nós da outra árvore, para o qual o nó em questão tem uma ligação de correspondência.

Por fim, os dois números inteiros da estrutura, são as marcas temporais do Algoritmo de Pesquisa por Profundidade Primeiro (DFS) [Cor01]. Estas marcas são importantes, não só para distinguir os nós, mas também para determinar se um nó é antecessor de outro, técnica usada para a cobertura bilingue.

O código que define a estrutura é apresentado na Listagem 4.2 e a Figura 4.1 mostra um exemplo de uma árvore, com a representação da *string depth* de cada um dos nós.

Listing 4.2: Estrutura *Node*

```
1  typedef struct node *node;
2
3  struct node{
4   int fcount;
5   int scount;
6   int sdep;
7   int head;
8
9   node parent;
10  node child;
11  node brother;
12  node slink;
13
14  list corresp;
15 };
```

4.2.2.1 Algoritmo DFS

A pesquisa em profundidade primeiro é caracterizada por seguir um rumo de procura, que privilegia a profundidade da árvore, ou seja, sempre que um nó tenha um filho, a pesquisa continua para esse filho e assim sucessivamente. Quando não existem mais filhos num determinado nó, a pesquisa continua para o seu irmão direito, seguindo a mesma lógica anterior de procura pelos seus filhos. Caso não haja qualquer irmão, ou já se tenha percorrido todos, regressa-se ao pai e a pesquisa é direccionada para o seu irmão, prosseguindo a pesquisa exactamente da mesma forma, até o objectivo ser encontrado, ou se regressar à raiz sem a pesquisa ter tido sucesso.

As marcas temporais do algoritmo, indicam precisamente em que lugar um nó é acedido pela primeira vez, que na estrutura é representado por *fcount*, e quando é acedido pela segunda vez. O segundo acesso a um nó, acontece quando já foi feita a pesquisa para todos os seus filhos, voltando-se então ao nó, para prosseguir para os seus irmãos. Na estrutura *Node*, este valor é representado por *scount*. Na Figura 4.2

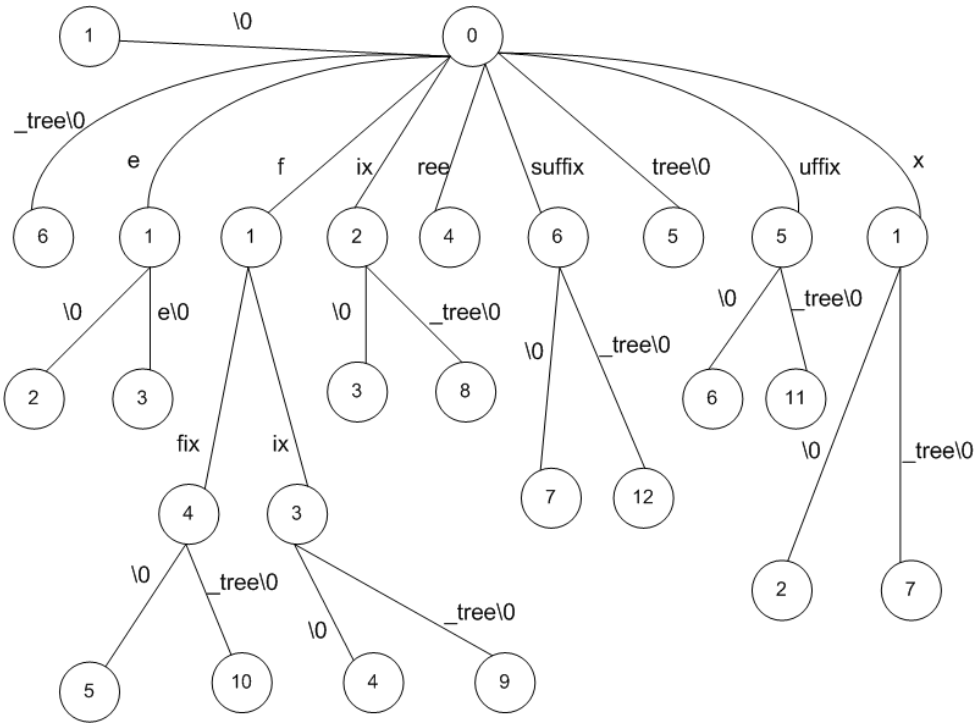

Figura 4.1: Árvore de sufixos de "suffix_tree", com o valor da *string depth* nos nós e com a representação dos terminadores.

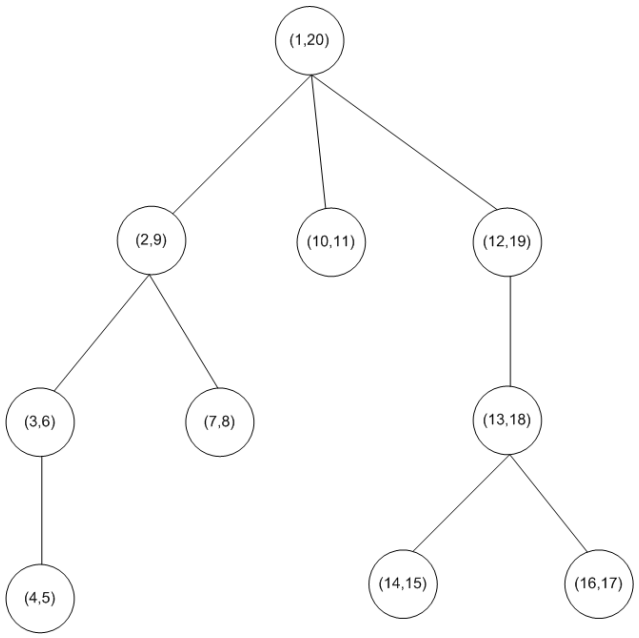

Figura 4.2: Exemplo de uma árvore com os marcadores temporais DFS

é mostrado um exemplo de uma árvore aleatória, com este tipo de marcas temporais, mostrando uma ordem de acesso aos nós segundo o algoritmo de DFS.

4.2.3 Estrutura *Point*

A estrutura *Point* tem uma grande importância para a implementação, pois é aquela que é utilizada para descer por uma dada aresta na árvore. Sempre que for preciso fazer uma pesquisa, seja descendo pelos nós e arestas, seja seguindo *suffix links*, é usado um *Point*. Um *Point* é uma espécie de ponteiro, que aponta sempre para uma dada zona de uma aresta ou para um nó, alterando-se o seu valor, sempre que é efectuada uma descida na árvore ou se segue um *suffix link*. Tal como um nó, um *Point* tem uma *string depth*, um pai e um filho, sendo o pai o nó imediatamente acima do ponteiro e o filho o nó imediatamente abaixo, ou o próprio caso se encontre num nó. A Listagem 4.3 tem o código da estrutura.

Listing 4.3: Estrutura *Point*

```
1 typedef struct point *point;
2
3 struct point{
4   node father;
5   int sdep;
6   node *son;
7 };
```

4.2.4 Estrutura *List*

Esta estrutura foi criada para apoiar a operação de correspondência, guardando os nós que têm ligação de correspondência com o nó em causa. Trata-se de uma lista com cabeça, cauda e em que cada elemento tem um nó e um apontador para o elemento seguinte. Para percorrer a lista sempre que necessário, foi criado um iterador para esse efeito. Pode ver-se a estrutura na Listagem 4.4.

4.3 Gestão dos Dados

4.3.1 Construção das Árvores

Para construir as duas árvores de sufixos, a implementação usa o algoritmo de Ukkonen. O algoritmo já foi explicado em pormenor no Capítulo 2, pelo que agora faremos uma abordagem mais especifica ao sistema implementado.

As duas árvores são criadas consecutivamente, imediatamente após o início do programa. Para cada uma das árvores define-se a raiz, que tem as mesmas características

Listing 4.4: Estrutura *List*

```
typedef struct elem *elem;
typedef struct list *list;
typedef elem *pos;
typedef struct iterator *iterator;

struct elem{
 node n;
 elem next;
};

struct list{
 elem head;
 pos tail;
};

struct iterator{
 pos i;
};
```

em ambas. A *head* da raiz é sempre 1, pois é o nó que não tem qualquer tipo de elemento associado, apontando assim para a posição do primeiro sufixo no *array* T.

De seguida, inicializam-se os marcadores temporais DFS com os valores 100 e 200. Normalmente a opção para estes marcadores é começar pelo 1, fazendo-se um incremento de 1 em 1, para definir os outros marcadores. Porém, por razões explicadas na secção seguinte, optámos por estes valores de maior dimensão. Por fim, define-se o *suffix link* da raiz e o *array* T, de modo a estar preparado a receber novos termos.

Neste ponto, as árvores de sufixos não têm qualquer tipo de termo associado, pois apenas foram criadas as estruturas necessárias para o programa começar a receber dados. Na secção seguinte, são apresentadas as operações que permitem carregar novos dados para o léxico.

4.3.2 Adição de Dados

Com as árvores construídas, o método add_pairs é sempre executado uma vez, no início do programa, de modo a compor o sistema com os pares que já são conhecidos. A explicação da operação para adicionar vários pares, encontra-se separada da operação para adicionar apenas um, devido a existirem algumas diferenças entre as duas.

4.3.2.1 Adicionar Vários Pares

Para adicionar vários pares ao sistema, é necessário ler os termos de um ficheiro. Este é caracterizado por ter o formato seguinte, não necessitando de qualquer tipo de ordenação entre os termos. Assumindo a língua inglesa e portuguesa:

Termo Inglês '$\backslash t$' Termo Português '$\backslash n$'

Após ser feita a leitura de um par, é seguido um processo semelhante para cada um dos termos. Em primeiro lugar, actualiza-se o T de cada uma das árvores, com o termo da língua respectiva. Após adicionar o termo a T, é acrescentado um terminador no final, de modo a indicar o fim do termo em T.

Como se trata de uma árvore de sufixos generalizada, todos os termos devem ter um terminador distinto. Como podemos ter muitos milhares de termos, seria impossível arranjar símbolos diferentes para cada um deles. A situação ainda se torna mais incomportável, com a condicionante de qualquer cadeia de caracteres adicionada, poder ter qualquer letra do abecedário, como também símbolos de pontuação e algarismos. Por isso, convenciona-se o '$\backslash 0$' como terminador universal. Para simular um terminador sempre único, definimos uma função de comparação de caracteres, onde se define que o '$\backslash 0$' é sempre distinto, mesmo quando comparado com o próprio.

De seguida, é adicionado o termo à árvore, assim como todos os seus sufixos. Usando o algoritmo de Ukkonen, tenta-se descer na árvore de sufixos, por cada um dos caracteres do primeiro sufixo (que é igual ao termo). Caso seja possível descer, não há nada a adicionar, continuando-se para o carácter seguinte. Caso não seja possível descer, então tem de se acrescentar o carácter, que pode levar a adicionar um novo ramo e um novo nó na árvore. Este pode ser um nó interno, o que quer dizer que existe um nó abaixo do ponto onde a descida parou, ou pode ser uma folha, quando se conseguiu descer até ao final da árvore, mas ainda havia mais caracteres do termo para ler. De seguida, seguem-se *suffix links* para definir os restantes sufixos do termo principal, seguindo a partir daí uma metodologia exactamente igual à explicada para o primeiro sufixo.

Uma situação importante, que nos permite manter o sistema com características lineares, trata-se de evitar criar uma nova folha, caso surja um termo exactamente igual a outro que exista na árvore. Como exemplo, assume-se o termo "abandon" que já se encontra no sistema com o par "abandonado". Pode acontecer que "abandon" surja novamente, mas desta vez com um par diferente, como por exemplo "abandonada". Como todos os termos têm o terminador único no final, quando se chegasse ao último carácter de "abandon", o '$\backslash 0$', não se conseguia descer na árvore (pois o terminador é sempre distinto de qualquer outro carácter), levando a criar uma nova folha. Como

nos casos das formas verbais, existem muitas traduções em português para uma palavra em inglês, isso levaria a uma repetição elevada de termos na língua inglesa. Deste modo, o sistema criaria sempre novas folhas, consoante o número de termos "abandon" que surgisse, o que obrigaria a um consumo muito maior de espaço. A performance do sistema também iria diminuir consideravelmente, no que toca a percorrer a árvore, nomeadamente na adição de novos termos.

Esta situação é evitada, comparando sempre o último carácter do termo com '\0' de forma directa (com !=), não utilizando a função de comparação que é usada normalmente. Caso se confirme um terminador, então a folha não é adicionada, pois já existe uma para esse termo. Caso não seja um terminador, então é necessário adicionar uma nova folha à árvore, pois trata-se de um termo novo.

A operação add_pairs faz precisamente este processo para todos os termos no ficheiro, podendo ser repetida várias vezes, mesmo que o ficheiro tenha exactamente os mesmos termos que já tenham sido adicionados. Nesse caso, nada seria alterado nas estruturas de dados, mas o sistema não origina qualquer tipo de erro. No entanto, basta haver uma pequena diferença no ficheiro, para a árvore ser alterada, com essa nova informação. No final, o sistema informa que os pares foram inseridos, com a mensagem "Pairs added to the system".

Sempre que um método add_pairs é executado, são calculados os valores dos marcadores temporais DFS para toda a árvore, com uma adaptação do algoritmo DFS. Assim, após se inserir vários pares, é feita uma reestruturação também do valor dos marcadores da árvore, podendo alguns valores serem modificados, já que é feita uma iteração completa do algoritmo DFS. Os valores dos marcadores, são sempre aumentados de 100 em 100 valores, devido a um pormenor de implementação da operação explicada na Secção 4.3.2.2.

4.3.2.2 Adicionar um Único Par

A operação add_pairs, podia ter sido implementada de forma a utilizar a implementação da adição de um único par, apresentada nesta secção. Porém, é importante que após qualquer alteração, os marcadores temporais DFS sejam actualizados. Como na operação anterior adicionam-se vários termos, é plausível que seja feito o cálculo dos marcadores para toda a árvore, no final de ser tudo inserido. No entanto, só pela adição de um único termo, isso torna-se bastante ineficiente, pois as mudanças na árvore podem não ser muito significativas.

A ideia seguida foi adaptar os marcadores temporais do novo nó, de acordo com os outros já existentes ao seu redor, sem ter de os alterar. A atribuição destes valores é feita, verificando os nós directamente mais próximos, sejam irmãos ou o pai, dividindo

a diferença entre os valores mais próximos por 4, no caso de serem folhas, ou por 2, no caso de serem nós internos. O primeiro marcador ficará com o limite inferior + valor calculado, enquanto o segundo marcador será igual ao limite superior - valor calculado. Na Figura 4.3, no primeiro caso, temos como limite inferior 300 e como superior 400. (400-300)/4 = 25, logo 300+25 = 325 e 400-25 = 375, que são os marcadores do novo nó do primeiro exemplo da Figura.

Figura 4.3: Exemplos de novos nós criados (a tracejado), com os seus marcadores DFS

Esta é a razão para se utilizarem valores para os marcadores, na ordem das centenas, pois consegue evitar-se por algum tempo actualizar os marcadores dos nós todos, sempre que um novo nó é adicionado. O valor na ordem das centenas foi o escolhido, mesmo com a hipótese de escolher um de ordem superior, para não correr o risco de atingir o limite máximo representável dos números inteiros.

Quando se dá a situação de não se conseguir dividir mais os marcadores temporais, por existirem valores muito próximos, não há outra alternativa que não fazer um novo cálculo para todos os nós da árvore, à semelhança do add_pairs.

O resto da operação é bastante semelhante à adição de um par na add_pairs, adicionando-se o termo ao *array* T e à árvore, exactamente da mesma forma. Após o par ser inserido, o sistema apresenta a mensagem: ”Pair added to the system”. Caso este já exista, a mensagem é a seguinte: ”Pair already exists”.

Tanto esta operação, como a add_pairs, tem ainda um passo final que está relacionado com a definição da relação de correspondência. Na Secção 4.3.4 explicamos ao pormenor, como é feito esse passo, ou seja, como definimos as ligações de correspondência entre dois termos de duas árvores diferentes.

4.3.3 Obter Todos os Pares

O método all_pairs, pode ser utilizado para definir uma espécie de cópia de segurança para os dados do léxico. No add_pairs, os dados são lidos de um ficheiro e carregados para a árvore. No all_pairs, a situação é a inversa. A ideia é percorrer a árvore e, para cada termo representado (e não os seus sufixos), imprimir os seus pares no formato seguinte, segundo uma ordem lexicográfica dos termos na primeira língua. Voltando a assumir o inglês e o português:

Termo Inglês '$\backslash t$' Termo Português '$\backslash n$'

Pode acontecer o mesmo termo surgir várias vezes no ficheiro, pois se um termo na primeira língua tiver várias ligações de correspondência, este aparece no ficheiro o número de vezes igual ao número de ligações que tem, e de forma consecutiva.

A informação sobre actualizações na árvore é também mostrada de cada vez que este método é executado, seja pela inexistência de pares previamente presentes na árvore, ou pela existência de novos pares, que não faziam parte da versão anterior do léxico. Este ficheiro poderá ser carregado normalmente, numa futura execução do sistema, com o método add_pairs.

A implementação desta operação baseia-se numa adaptação do algoritmo de pesquisa por profundidade primeiro, retirando todos os nós por essa ordem. Para cada nó retirado, é verificado se tem alguma correspondência. Caso tenha, os termos respectivos à *path-label* dos nós, o retirado e o de correspondência, são escritos no ficheiro. Caso contrário passa-se ao nó seguinte.

Basta fazer a pesquisa numa árvore, pois através da lista de correspondências dos nós, consegue obter-se a informação da outra árvore. Depois de o ficheiro estar escrito, o LEXMAN devolve a mensagem: "Data printed into file".

4.3.4 Correspondência

Uma ligação de correspondência entre dois termos, marca-os como tradução um do outro. Quando um novo par é adicionado, o sistema assume automaticamente que se trata de um par de tradução, pelo que é feita a ligação entre os dois termos.

Imediatamente a seguir aos dois termos serem adicionados às árvores, obtêm-se os nós criados e respectivos ao maior sufixo adicionado, ou seja, ao nó que tem como *path-label* o termo completo. Assume-se que X é o nó da árvore com os termos em inglês e Y o nó da árvore com os termos em português. Cada nó tem uma lista associada, tal como é mostrado na Listagem 4.2, que armazena os nós com os quais tem uma ligação de correspondência. Portanto, para o nó X é feita uma pesquisa à sua lista, para verificar se Y já lá se encontra representado. Se não estiver, então adiciona-se Y à lista de X. O mesmo processo é repetido para Y. No final, existe uma relação de correspondência entre X e Y, pelo que os termos respectivos aos nós X e Y, passam a formar um par.

Além do sistema ter as suas estruturas preparadas para comportar estas ligações virtuais, também oferece uma operação que permite que seja perguntado, se dois termos formam um par de tradução conhecido pelo léxico. Esse método é o is_pair.

Para verificar se dois termos formam um par, é necessário pesquisar pelos nós com *path-label* igual aos termos, sendo que para a árvore com termos em inglês, procura-se pelo termo em inglês e o mesmo acontece para a árvore e termo em português. Usando mais uma vez a terminologia do nó X e Y, basta verificar se Y se encontra na lista de correspondências de X e vice-versa, para descobrir se existe uma ligação entre ambos. Essa verificação é feita usando os marcadores temporais DFS, que acabam por ser também identificadores únicos, para distinguir os vários nós. Caso o par seja confirmado, o programa devolve ”YES”. Caso contrário, a resposta do sistema é ”NO”, indicando que esse par não está definido.

4.3.5 Remover um Par

A opção de implementação tomada para remover um par do sistema, foi eliminar apenas a ligação de correspondência entre eles. Para eliminar os dois termos das árvores, levaria a uma implementação bastante complicada, pois poderia ser necessário modificar muitos apontadores, para as estruturas ficarem consistentes. Adicionalmente, o termo teria de ser eliminado do *array* T. Com tamanhos muito elevados de T, ou seja, para muitos termos existentes no sistema, não se tornaria muito eficiente.

Eliminando apenas as ligações de correspondência, não há o risco do sistema ficar incoerente. Cada árvore representa o léxico de uma linguagem, pelo que não faria muito sentido eliminar um termo de uma linguagem, só porque ficou sem par de tradução. Caso se pretenda adicionar de novo o par que se acabou de remover, basta voltar a acrescentar essa ligação, com a operação add_pair.

Para remover um par com o método delete_pair, faz-se uma pesquisa pelos nós cuja *path-label* é igual aos termos, tal como no método is_pair. De seguida, é feita uma

pesquisa pelas listas de correspondência de cada nó, tal como no is_pair, com a única diferença que desta vez o nó Y é eliminado da lista de X e vice-versa. Após remover um par, o LEXMAN imprime a mensagem "Pair deleted". Se o par não existir, não havendo por isso qualquer dado removido, a mensagem apresentada é "Inexistent Pair".

Na Secção 4.5, mostramos alguns exemplos de modificações nas árvores de sufixos, após a execução destas operações de gestão.

4.4 Cobertura

Como já foi mostrado, existem dois tipos de cobertura: a monolingue e a bilingue.

4.4.1 Cobertura Monolingue

A cobertura monolingue encarrega-se de descobrir, que segmentos de um dado termo não estão cobertos na respectiva árvore do sistema. Pode existir cobertura total, caso todo o termo ou todos os seus segmentos existam, como pode existir cobertura parcial, caso alguns dos seus segmentos não existam. O código utilizado é independente da árvore em que é feita a pesquisa, bastando por isso apenas indicar a língua do termo do qual se pretende descobrir a cobertura, de modo a distinguir a árvore a usar. Na Listagem 4.5 é apresentado o pseudo-código desta operação.

Listing 4.5: Cobertura Monolingue

```
1  nterm = pre_process(term);
2  int acumulatedSpace, i, j,len,nlen;
3  acumulatedSpace = i = 1;
4  j = 0;
5
6  for (; i < length(nterm); i++){
7      if(nterm[i] == '␣')
8             acumulatedSpace++;
9      while(!descend(nterm[i])){
10            if(nterm[j] == '␣'){
11                  if(acumulatedSpace < 2){
12                        //segment = segmento entre j e i
13                        write(segment);
14                  }
15                  acumulatedSpace--;
16            }
17            j++;
18            follow_suffix_link(nterm);
19     }
20     descend(nterm[i]);
21 }
```

A técnica utilizada para determinar a cobertura monolingue é bastante simples. Em primeiro lugar são necessários dois índices, i e j, onde i representa o carácter do termo por onde se está a tentar descer na árvore, e j que só incrementa quando não é possível descer. Com o i e o j, conseguimos definir se um segmento do termo a pesquisar, está coberto ou não pelo léxico. Para isso, precisamos de fazer um pré-processamento do termo, acrescentando um espaço no princípio do termo e um carácter único ($) no final, para podermos obter um termo semelhante ao representado na Figura 4.4.

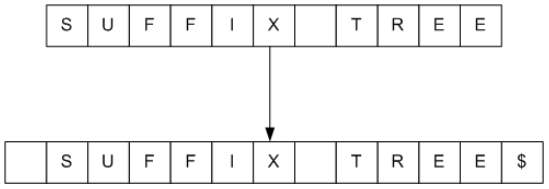

Figura 4.4: Termo "suffix tree" após o pré-processamento para a cobertura monolingue

Para fazer a pesquisa na árvore, tenta-se descer normalmente por todos os caracteres do termo, utilizando o índice i, que começa no primeiro carácter do termo. O índice j começa no primeiro espaço, criado antes da pesquisa dar início. Sempre que se passa para o carácter seguinte do termo, o índice i é incrementado. Sempre que não se consegue descer, incrementa-se o índice j e seguem-se *suffix links* até se conseguir descer na árvore pelo carácter em questão. Quando são seguidos os *suffix links*, a lógica para j mantém-se, pois sempre que não se consiga à mesma descer e seja necessário seguir outro *suffix link*, o j continuará a incrementar.

Para confirmar se um termo está coberto, basta verificar na altura em que é impossível descer, se j e i se encontram num espaço ou no $, ou se o número de espaços acumulados (*accumulatedSpace*) entre os índices é 2, o que acontece com j num espaço e com i a ter apontado recentemente para um espaço ou $. Podem ver-se vários exemplos na Figura 4.5.

O resultado da função são os segmentos do termo em pesquisa, que não se encontram cobertos. Cada segmento é representado numa linha, seguindo um formato semelhante a este:

Segmento1
Segmento2
...

Esta opção foi tomada, porque para os protótipos de extracção de traduções e de alinhamento, é mais interessante descobrir logo o que não se conhece, do que descobrir

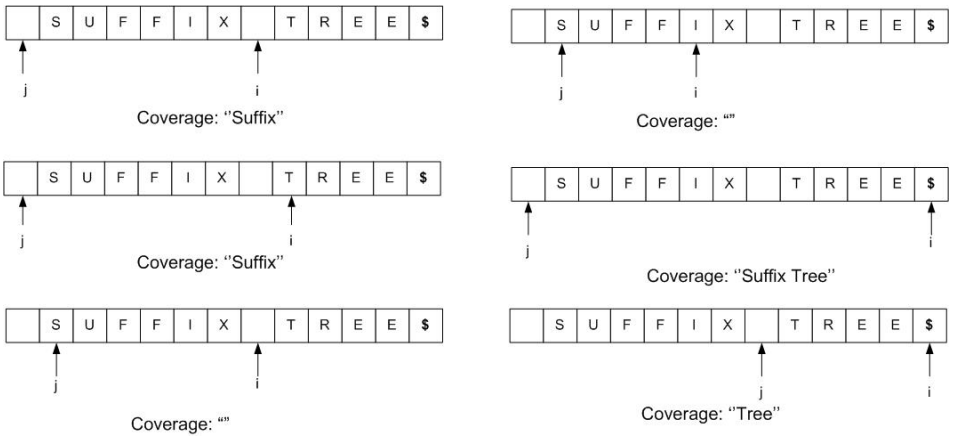

Figura 4.5: Vários exemplos de verificação de cobertura com os índices i e j

o que é conhecido, obrigando a calcular as partes em falta. Na Secção 4.5, apresentamos alguns exemplos de resultados da cobertura monolingue.

4.4.2 Cobertura Bilingue

A operação de cobertura bilingue é a mais complexa, pois exige muitas verificações e tratamento de muitos dados. Além das estruturas mais genéricas que mostrámos na Secção 4.2, apresentamos na Listagem 4.6 a estrutura *Cov*, específica para ajudar na operação de cobertura bilingue. Esta estrutura é utilizada na pesquisa pelos nós associados a um termo, que é feita em cada árvore, antes de se verificar se existe cobertura.

Listing 4.6: Estrutura *Cov*

```
1 typedef struct cov *cov;
2 struct cov{
3     int index;
4     int* startpos;
5     node* nodes;
6 };
```

O *index* indica o número de elementos no *array nodes* e no *array startpos*. O primeiro *array* armazena os nós devolvidos na pesquisa por um dos termos, enquanto que o segundo armazena a posição inicial nesse termo, da *path-label* de cada nó armazenado em *nodes*. Como exemplo, se tivermos o termo "regresso ao futuro", para um nó com *path-label* igual ao termo completo, é armazenado o número 1 em *startpos*. Se devolver apenas um nó com *path-label* "futuro", então o valor associado a este nó, que é guardado em *startpos*, é o 13. Na Listagem 4.7 pode ver-se o pseudo-código geral da

Listing 4.7: Cobertura Bilingue

```
cov fcov = get_nodes(tree1,term1);
cov scov = get_nodes(tree2,term2);
int i = 0;
for ( ; i < scov->index; i++){
  check_correspondence(scov->nodes[i], scov->startpos[i], fcov,tree1,tree2);
}
write_term(term1);
write_term(term2);
```

operação de cobertura bilingue.

No início da operação de cobertura bilingue, percorrem-se as duas árvores, à procura dos nós respectivos ao termo. Se o termo existir na totalidade, é retirado o nó com essa *path-label*. Para fazer este tipo de verificação, é utilizado um algoritmo semelhante ao da cobertura monolingue, utilizando os dois índices i e j. Caso o termo não exista na totalidade, existindo apenas alguns segmentos, o resultado pode ser mais que um nó, todos eles com *path-label* igual a algum dos segmentos encontrados.

O passo seguinte já está relacionado com a verificação da cobertura. Para todos os nós retirados da segunda árvore, percorremos a sua lista de correspondências. Para cada um dos nós dessa lista, que são todos elementos da primeira árvore, será verificada a cobertura, com os nós retirados da pesquisa nessa mesma árvore (*get_nodes*). Para existir cobertura, um nó tem que ser antecessor de outro. Assumindo que um nó da lista de correspondências do termo (ou dos seus segmentos) em português é X, então X terá que ser antecessor de um dos nós obtidos da pesquisa pela primeira árvore. Seja Y um desses nós, X é antecessor de Y se: $(X->fcount <= Y->fcount)$ e $(X->scount >= Y->scount)$. Desta forma, a verificação de se um nó é antecessor de outro, é feita em tempo constante, $O(1)$. De notar, que a pesquisa por um antecessor é feita com o uso de uma pesquisa binária, para diminuir a complexidade da operação.

Quando não é encontrada uma correspondência para um nó, sobe-se ao pai deste e repete-se o processo, para verificar outros subsegmentos do segmento que está a ser utilizado. Imaginando que "suffix tree" se encontra na árvore, mas não tem correspondência, "suffix" pode ter, pelo que a subida ao pai pode encontrar essa ligação. Uma amostra do código da verificação de correspondência apresentada, encontra-se na Listagem 4.8.

À semelhança da cobertura monolingue, o resultado final não são os segmentos cobertos, mas sim os não cobertos. Como neste procedimento, a implementação utiliza nós existentes na árvore obtidos através das pesquisas previamente mencionadas, utilizámos um procedimento adicional para que a informação seja obtida, da forma

Listing 4.8: Verificação da Existência de Correspondência

```
int k = -1;
iterator it = create_iterator(n->corresp);
while(has_next(it)){
    elem e = next(it);
    node nxt = e->n;
    k = check_ancestor(fcov->nodes, nxt);
    if (k != -1)
         break;
}
free_iterator(&it);
if (k == -1 && n->sdep > 2){
      node new = n->parent;
      if (new->child != NULL && new->child != n)
              new = new->child;
      k = check_correspondence(new,i,fcov, eng, pt, bt1, bt2);
}
```

pretendida.

Utilizámos dois *arrays* de inteiros, um para cada termo. A ideia é sempre que encontramos cobertura para algum segmento, preenchemos estes *arrays* com 1's, nas posições correspondentes aos segmentos, cuja existência de cobertura foi confirmada. Deste modo, depois de feitas todas as verificações, imprimem-se os caracteres dos termos, cujas posições respectivas nos *arrays* de inteiros, contêm um zero.

No *output*, após cada segmento, existe uma separação por um *tab*, para esperar por outro segmento. O formato é o seguinte, assumindo de novo o inglês e o português:

SegmentoInglês1 \t SegmentoInglês2 \t ...
SegmentoPortuguês1 \t SegmentoPortuguês2 \t ...

Na secção seguinte, mostramos alguns resultados das duas coberturas, para exemplificar um pouco melhor o *output* de cada função.

4.5 Exemplos de Resultados

4.5.1 Operações de Cobertura

A Tabela 4.1 mostra um exemplo de um léxico, com o qual exemplificamos alguns resultados das coberturas. Esses resultados são divididos por duas tabelas, a primeira (4.2) para a cobertura monolingue e a segunda (4.3) para a cobertura bilingue. O *output* mostrado nas tabelas não está exactamente igual ao produzido pelo LEXMAN. A ideia

é apenas mostrar os segmentos que o programa devolveria como resultado e não o formato exacto dos mesmos.

Os resultados obtidos na Tabela 4.2, tal como foi explicado, são os segmentos sem cobertura. No terceiro exemplo, pode ver-se que não é necessário, para existir cobertura total, que todo o termo esteja no léxico. Basta os seus diferentes segmentos existirem, pelo que nada é devolvido no resultado final. No quarto exemplo, se ”somar onze valores” fosse ”somar valores onze”, então ”somar valores” seria considerado um segmento único sem cobertura, mas como o ”onze” surge no meio, isso não acontece. De notar, que a língua no *input* é essencial, tal como se pode ver no último exemplo. O termo existe, mas é na outra língua representada.

Na Tabela 4.3, a situação é bastante idêntica à da cobertura monolingue. No primeiro exemplo, mesmo que "nice” ou "lindos” fossem conhecidos, como não haveria ligação de correspondência à mesma, o resultado final conteria à mesma os dois termos. De notar que aqui, temos os termos de línguas diferentes encontram-se separados por uma mudança de linha. No quarto exemplo, com um resultado que consiste nos termos ”number”, "thousand”, ”numero” e "mil”, no programa a separação dos dois segmentos da mesma língua devolvidos é por um *tab*. Neste exemplo, o resultado são dois pares de termos separados, pois no meio temos o par "11” <=> "onze”, conhecido pelo léxico.

Tabela 4.1: Pequeno exemplo de um léxico

Inglês	**Português**
11 months	onze meses
11	onze
12	doze
months	meses

Tabela 4.2: Exemplos de cobertura monolingue

Exemplo	**Resultado**
mono_coverage(”eng”, ”11 nice months”)	”nice”
mono_coverage(”pt”, ”doze noites”)	”noites”
mono_coverage(”eng”,”11 12”)	
mono_coverage(”pt”, ”somar onze valores”)	”somar” ”valores”
mono_coverage(”eng”, ”onze”)	”onze”

Tabela 4.3: Exemplos de cobertura bilingue

Exemplo	**Resultado**
bili_coverage(”11 nice months”, ”onze lindos meses”)	”nice” ”lindos”
bili_coverage(”12 nights”, ”doze noites”)	”nights” ”noites”
bili_coverage(”11 12”, ”onze doze”)	
bili_coverage(”number 11 thousand”, ”numero onze mil”)	”number” e ”thousand” ”numero” e ”mil”
bili_coverage(”onze”, ”11”)	”onze” ”11”

4.5.2 Operações de Gestão

Nesta secção mostramos algumas figuras com as árvores resultantes, após serem efectuadas algumas das operações de gestão explicadas anteriormente.

Na Figura 4.6, surgem as duas árvores de sufixos após a execução do método para adicionar vários pares (add_pairs), para o pequeno ficheiro de *input* mostrado na figura. As árvores resultantes armazenam os termos respectivos, são criadas as ligações de correspondência entre os pares e os valores dos marcadores DFS são calculados.

Na Figura 4.7, pode ver-se as mesmas árvores com a adição de um novo par, neste caso o ”after” <=> ”day”, assim como o ficheiro de *output* produzido pelo método all_pairs, que já conta com o último par a ser inserido. Alguns dos valores dos marcadores DFS, têm de ser recalculados, pela criação de novos nós.

Por fim mostramos na Figura 4.8, as mesmas árvores após a eliminação do par ”night” <=> ”noite”, onde se verifica que esses termos mantém-se nas árvores respectivas, mas a ligação de correspondência que os marca como um par, é eliminada. Como a estrutura da árvore não é alterada aqui, os marcadores DFS mantêm os mesmos valores da Figura 4.7.

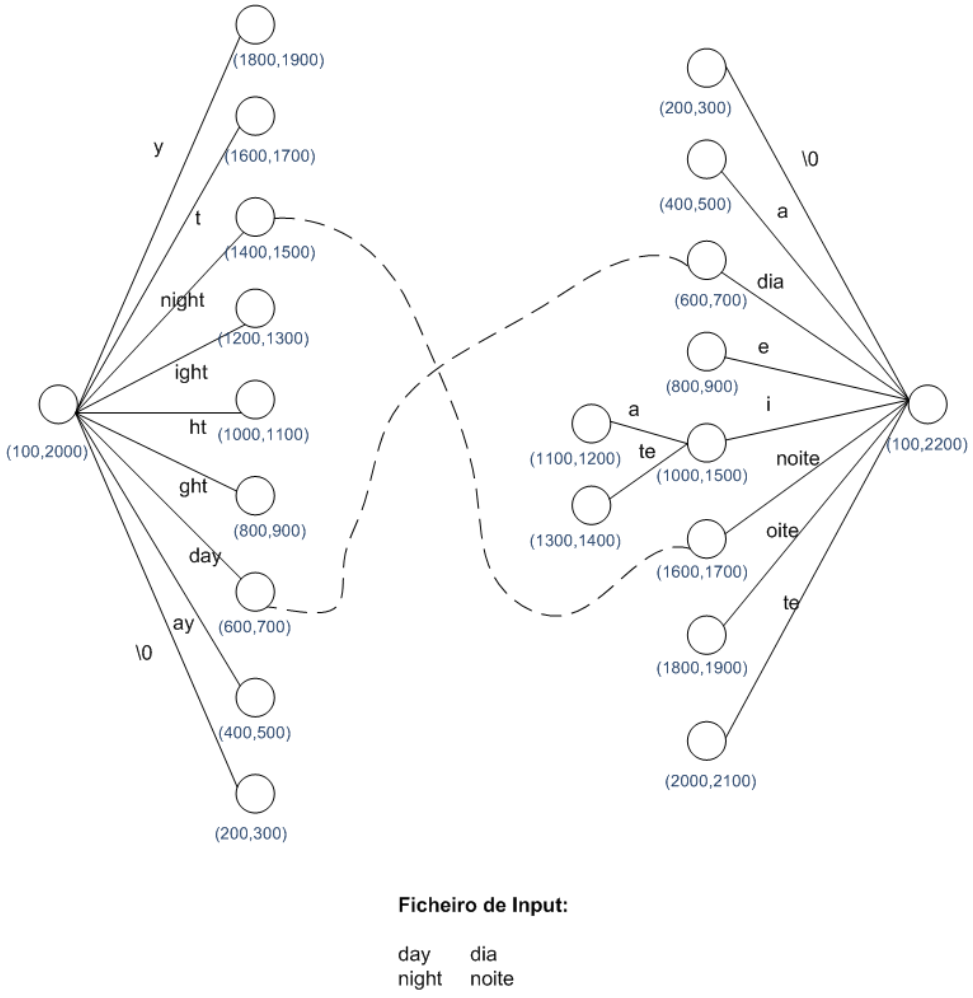

Figura 4.6: Árvores após a execução do método add_pairs.

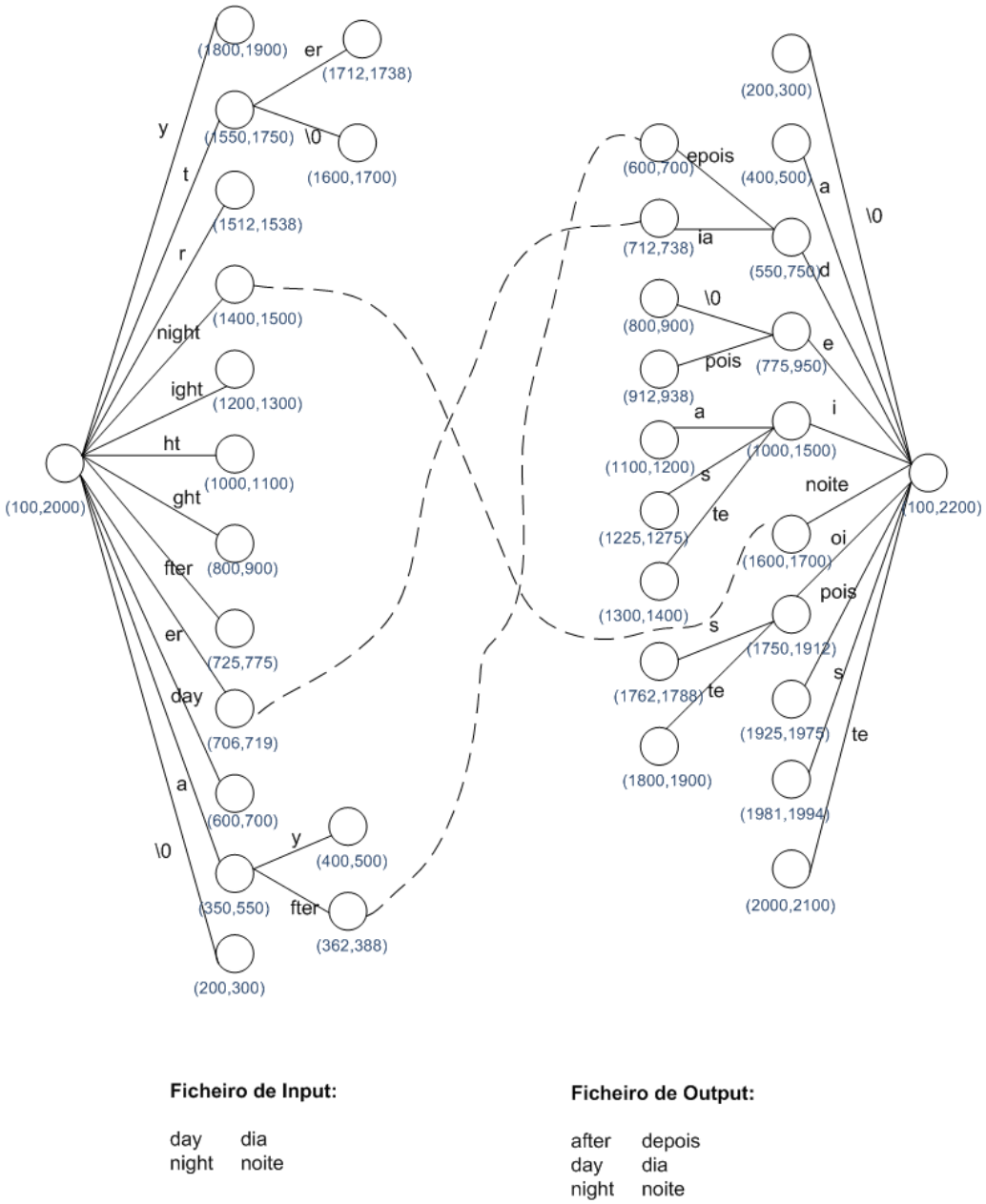

Figura 4.7: Árvores após a execução do método add_pair(”after”,”depois”)

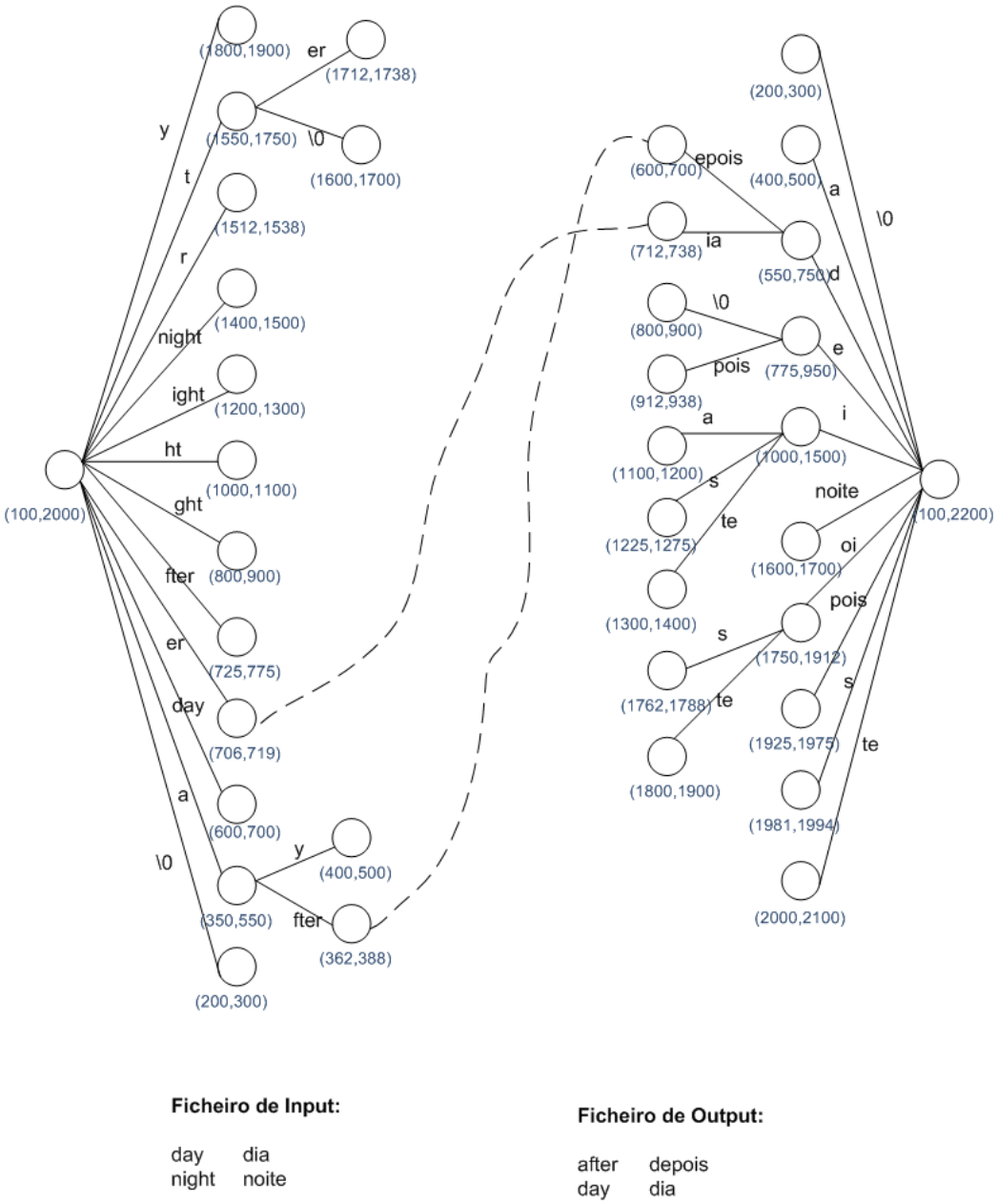

Figura 4.8: Árvores após a execução do método delete_pair(”night”,”noite”)

5

Apresentação de Resultados

5.1 Complexidades Temporais

- $\mathbf{t}_1$ – Termo na primeira língua representada.
- $\mathbf{t}_2$ – Termo na segunda língua representada.
- **l** – Língua do termo a pesquisar na cobertura monolingue.
- **t** – Termo na língua 'l'.
- **f** – Nome do ficheiro de *input*
- **s** – Tamanho do ficheiro 'f'.
- $\mathbf{w}_1$ – Número de palavras do termo da árvore 1.
- $\mathbf{l}_2$ – Número de ligações dos nós da árvore 2, para a árvore 1.
- ε – Factor que pode ser igual a 1 ou 2.

A generalidade das operações tem uma complexidade linear, fruto das características das árvores de sufixos, no que toca à construção da própria e das operações de pesquisa que suporta. Todas as operações cujas complexidades dependem dos termos em argumento, têm tempos lineares dessa ordem, devido às pesquisas realizadas na árvore. No caso do add_pair, delete_pair e is_pair, são feitas duas pesquisas pelos dois

Tabela 5.1: Complexidades temporais das operações

Operação	**Complexidade**
add_pair(t_1,t_2)	$O(\|t_1\| + \|t_2\|)$
add_pairs(f)	$O(s)$
all_pairs(f)	$O(s)$
delete_pair(t_1,t_2)	$O(\|t_1\| + \|t_2)$
is_pair(t_1,t$_2$)	$O(\|t_1\| + \|t_2\|)$
mono_coverage(l,t)	$O(\|t\|)$
bili_coverage(t_1,t_2)	$O(\|t1\| + \|t_2\| + (l_2^{\varepsilon} \times log(w1))$

termos em argumento, pelo que a complexidade da operação depende do tamanho dos dois termos. Assim, temos $O(|t_1|)$ na pesquisa pelo primeiro termo, na respectiva árvore, e $O(|t_2|)$ pelo segundo termo. Deste modo, a complexidade é $O(|t_1| + |t_2|)$. Na cobertura da monolingue temos apenas uma pesquisa, pelo que só conta o único termo em argumento.

Quanto às operações que exigem a leitura ou a escrita de dados de um ficheiro, a sua complexidade amortizada está dependente do tamanho do ficheiro. Isso acontece pois o tempo para inserir cada par de termos no sistema, ou para escrever os pares no ficheiro, acaba por não ter um peso significativo, que faça com que a operação deixe de ter comportamento linear, com dependência do tamanho do ficheiro.

No que toca à cobertura bilingue, como se trata da operação mais complexa, a sua complexidade temporal acaba por ser a mais difícil de estimar e a menos eficiente. A soma dos tamanhos dos textos, equivalem à pesquisa de ambos nas respectivas árvores. O resto da expressão corresponde a percorrer o número de nós obtidos da árvore em português, que é dado pelo número de palavras desse termo e, para cada um desses termos, fazer-se a verificação da cobertura com os nós da árvore em inglês. No entanto, como é utilizada uma pesquisa binária e recorre-se à própria árvore, para descobrir outros segmentos do termo, evita-se uma potencial complexidade quadrática, obtendo ordem logarítmica: $(l_2 \times log(w1))$. O factor ε, deve-se ao facto de na pesquisa por uma correspondência, por vezes ser necessário subir ao pai do nó encontrado na pesquisa, quando não se encontra uma ligação para esse nó.

5.2 Avaliação Temporal da Construção das Árvores

Nas secções seguintes, apresentamos alguns resultados importantes do sistema LEXMAN. As medidas temporais e de memória, foram retiradas usando um computador com processador Intel Core2 Duo P8700 2.53GHz, memória de 6 GB e num sistema

operativo Unix, o Ubuntu 9.10.

No arranque do programa, é sempre feito um carregamento de dados de um ficheiro de *input*. Para testar a natureza linear do sistema, definimos vários ficheiros de *input* de tamanhos diferentes, medindo de seguida o tempo de execução da operação add_pairs, para cada um desses ficheiros. Na Figura 5.1 pode ver-se um gráfico da evolução temporal em segundos, em função do aumento do número de termos do ficheiro de *input*, enquanto que na Tabela 5.2 mostramos os valores concretos, com que foram feitas as medições.

Tabela 5.2: Tempos obtidos na construção das árvores

Número de Termos	**Tempo**
10000	0.284 segundos
20000	0.512 segundos
30000	0.752 segundos
40000	0.988 segundos
50000	1.212 segundos
60000	1.472 segundos
70000	1.688 segundos
80000	1.879 segundos
90000	2.092 segundos
100000	2.296 segundos
200000	4.404 segundos

5.3 Recursos Temporais e de Memória Consumidos

Para esta secção, dividimos os testes em dois. Primeiro, para operações que envolvem o *input* e *output* dos termos, que são bastante mais demoradas, determinámos o tempo em segundos de cada uma das operações. Os resultados são mostrados na Tabela 5.3. Segundo, para as operações praticamente instantâneas, apresentamos na Tabela 5.4 uma média do número de operações que são executadas em 1 segundo.

Todos os valores de memória apresentados nas duas tabelas estão em Megabytes e contam com a operação de add_pairs inicial que o sistema executa. Quando surgem valores iguais aos da operação add_pairs, significa que não exigiram mais consumo de memória. O ficheiro de *input* para esta operação inicial tem 202534 pares e os termos utilizados para pesquisa, quando necessário, tinham três palavras de comprimento.

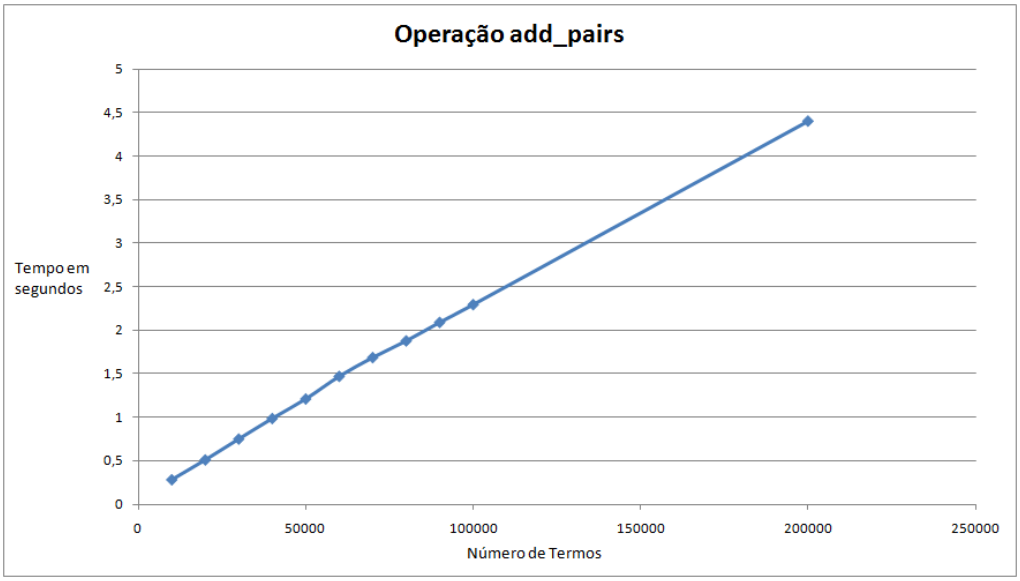

Figura 5.1: Gráfico com resultados da Tabela 5.2

Tabela 5.3: Tempos e memória consumida para as operações mais lentas

Operação	**Tempo Média p/ Operação**	**Memória Consumida**
add_pairs(f)	5.02 segundos	380 MB
all_pairs(f)	0.78 segundos	380 MB

Pelos dados da Tabela 5.3, percebe-se que não é a leitura nem a escrita dos termos, que ocupa a maior parte do tempo. A operação all_pairs é bastante mais rápida que add_pairs, pois esta última necessita de construir novos ramos e descer nas árvores, terminando com dois algoritmos DFS para definir as marcas temporais. No entanto, como foi provado na secção anterior, isso não afecta a natureza linear da operação, baseada no tamanho do ficheiro de *input*. No caso do all_pairs, apenas é necessário percorrer a árvore e seleccionar os nós que têm correspondências.

Tabela 5.4: Tempos e memória consumida para as operações mais rápidas

Operação	**Número de Operações/1s**	**Memória Consumida**
add_pair(t_1,t_2)	56253	447 MB
delete_pair(t_1,t_2)	175976	380 MB
is_pair(t_1,t_2)	242127	380 MB
mono_coverage(l,t)	62813	380 MB
bili_coverage(t_1,t_2)	23563	380 MB

No que toca à Tabela 5.4, os valores do número de operações realizadas demonstram bem a sua eficiência. Pela mesma ordem de ideias do add_pairs, o método add_pair é o mais demorado, devido à criação de novos nós e ao cálculo DFS. As operações is_pair e delete_pair, como basta descer na árvore até chegar ao fim do termo ou não se conseguir descer mais, ou seja, são simples pesquisas, tornam-se muito mais rápidas que todas as outras.

A cobertura monolingue, como por vezes exige seguir vários *suffix links* e ter um *output* mais extenso, acaba por ser uma operação bem mais lenta que as anteriores. A cobertura bilingue, pela sua maior complexidade devido às verificações necessárias, é naturalmente a operação mais lenta de todas.

5.4 Comparação de Implementações da Cobertura

Na Secção 4.1, explicámos o porquê da escolha das árvores de sufixos, em vez de *arrays* de sufixos. Para termos uma base de comparação, definimos uma implementação da cobertura monolingue e bilingue em árvores de sufixos, que simulasse uma implementação com *arrays* de sufixos. Basicamente, não são seguidos *suffix links*, o que obriga a voltar à raiz e voltar a descer, sempre que seja impossível prosseguir a pesquisa num dado ramo. Deste modo, os atalhos entre arestas que os *suffix links* oferecem, não são aproveitados. De seguida, apresentamos os resultados obtidos.

5.4.1 Resultados da Cobertura Monolingue

Na Tabela 5.5 apresentamos os resultados temporais para as duas versões da cobertura monolingue, assim como um gráfico com esses mesmos dados na Figura 5.2. A medida utilizada foi o número de operações executadas, em X segundos e para 202534 pares.

Tabela 5.5: Número de operações da cobertura monolingue, por unidade de tempo

Tempo Limite	**Nº Operações c/ *Suffix Links***	**Nº Operações s/ *Suffix Links***
10 segundos	732080	42680
20 segundos	1460602	85819
30 segundos	2193480	128542
40 segundos	2922737	171735
50 segundos	3647994	214947
60 segundos	4391173	258265

Na Tabela 5.6 apresentamos resultados semelhantes, mas com variação do tamanho

do léxico, com o gráfico 5.3 a mostrar a evolução dos valores. Os termos utilizados para estes testes foram: "after verifying that the information supplied in the exporter ' s application agrees with“ e "depois de terem verificado a conformidade de os elementos de o pedido de o exportador com“, um par existente no léxico.

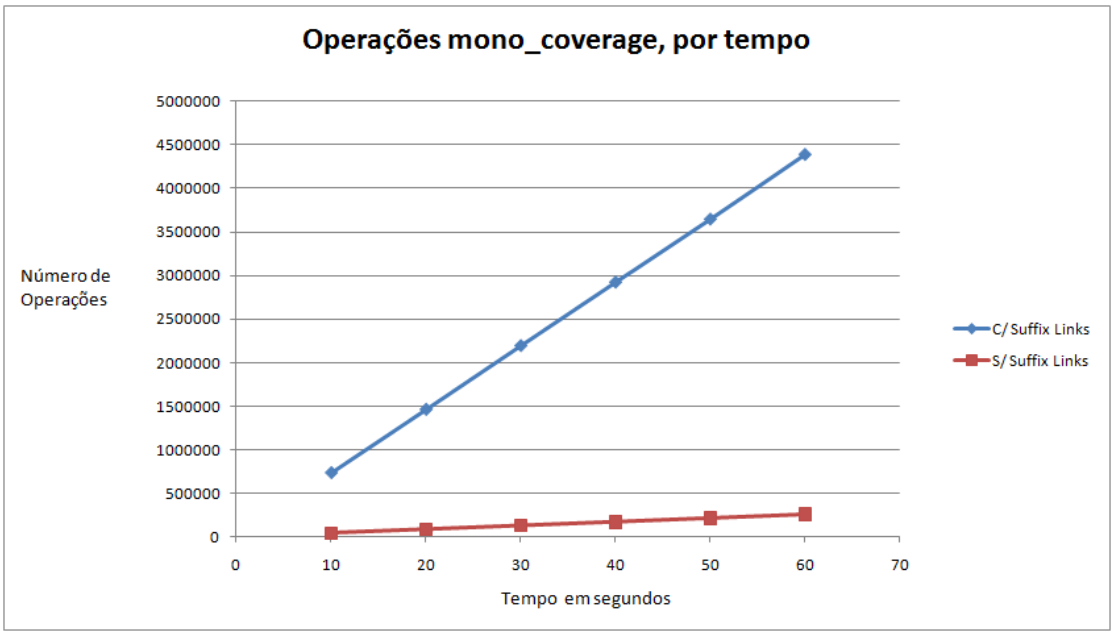

Figura 5.2: Gráfico com resultados da Tabela 5.5

Pelo gráfico 5.2 é perceptível que os *suffix links* são uma técnica de implementação muito útil. Voltar sempre à raiz torna-se muito mais ineficiente, pois estão se a evitar os atalhos entre ramos da árvore, que se mostram essenciais para a eficiência das operações de cobertura. A diferença entre os resultados das duas implementações é caracterizado por um factor de 17, ou seja, a implementação com *suffix links* é 17x melhor, mostrando as vantagens das árvores de sufixos para este tipo de operações.

Tabela 5.6: Operações da cobertura monolingue em 10 seg, por tamanho do léxico

Número de Termos	**Nº Operações c/ *Suffix Links***	**Nº Operações s/ *Suffix Links***
10000	743202	52536
20000	761570	50612
50000	740415	46964
100000	739338	44576
200000	709713	43394

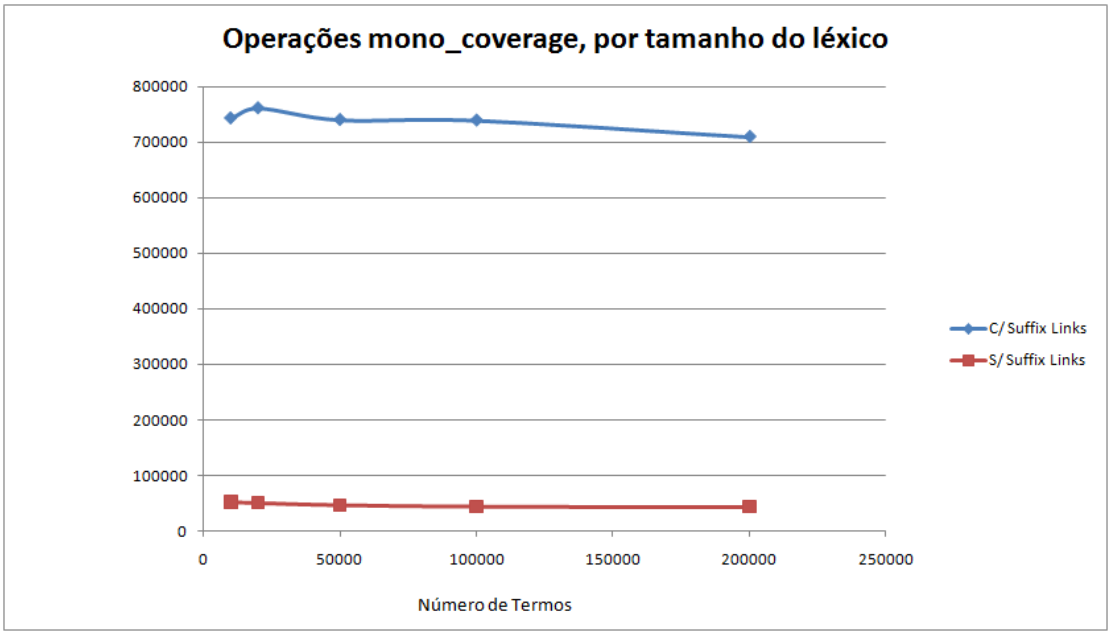

Figura 5.3: Gráfico com resultados da Tabela 5.6

Pelos dados da Tabela 5.6, pode verificar-se que a diferença entre implementações mantém-se, porém a variação do número de termos, não causa uma grande influência nos resultados. As operações de pesquisa nas árvores de sufixos, são mais dependentes do tamanho da expressão a pesquisar, pelo que as variações pequenas nos resultados são perfeitamente compreensíveis.

A partir de um determinado número de termos, a curva torna-se estritamente decrescente devido ao tamanho da árvore, em qualquer uma das curvas. A árvore ao ir crescendo, causa essa descida nos resultados, apesar de ser uma variação pequena. A razão para isso acontecer, deve-se ao crescimento inerente da árvore, o que faz com que haja mais hipóteses de descida a ter em conta e um consequente seguimento de mais *suffix links*. Mesmo assim, a diferença não é de todo muito considerável.

5.4.2 Resultados da Cobertura Bilingue

À semelhança da cobertura monolingue, também apresentamos aqui várias tabelas e gráficos para a cobertura bilingue. Todas seguem as características explicadas na secção anterior, com apenas as diferenças a mostrarem-se nos resultados, que explicitam bem a maior complexidade desta operação.

Analisando o gráfico 5.4, pode ver-se mais uma vez a vantagem do uso dos *suffix links*, pois é notória a melhoria em termos de eficiência de uma implementação para

Tabela 5.7: Número de operações da cobertura bilingue, por unidade de tempo

Tempo Limite	**Nº Operações c/ *Suffix Links***	**Nº Operações s/ *Suffix Links***
10 segundos	241050	18393
20 segundos	483375	37529
30 segundos	719631	55711
40 segundos	962218	73050
50 segundos	1197465	93396
60 segundos	1445280	112163

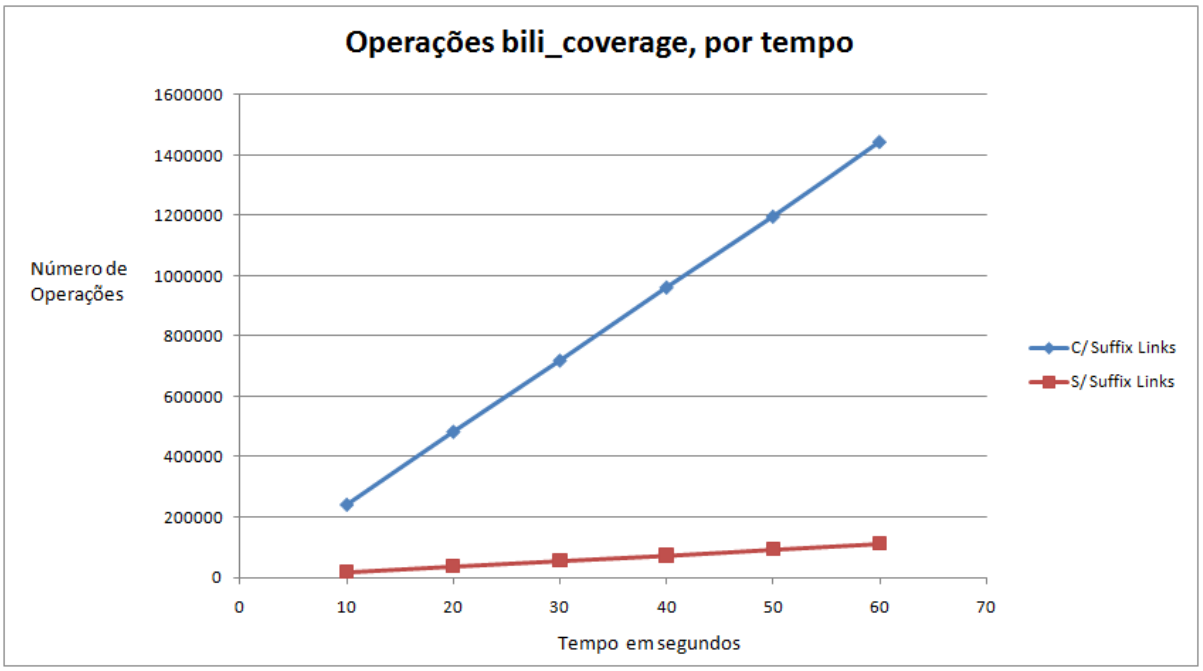

Figura 5.4: Gráfico com resultados da Tabela 5.7

outra. De notar que a implementação dos *suffix links*, só se verifica na pesquisa pelos nós nas duas árvores, mas mesmo assim, a diferença entre implementações é notória. A solução com *suffix links*, demonstra ser 13x melhor que sem *suffix links*.

Pelos dados da Tabela 5.8, encontramos uma situação semelhante à da cobertura monolingue. A diferença entre o número de operações executadas, entre as duas implementações é bastante grande. A variação entre os resultados, com o aumento do léxico, mostra-se muito pequena, sendo que a implementação sem *suffix links* continua a ser caracterizada por uma curva estritamente decrescente. Isto mostra que em ambos os casos, independentemente de uma curta variação de tamanho das árvores, causa sempre um impacto negativo em termos de resultados.

Tabela 5.8: Operações da cobertura bilingue em 10 seg, por tamanho do léxico

Número de Termos	**Nº Operações c/ *Suffix Links***	**Nº Operações s/ *Suffix Links***
10000	244326	25534
20000	244895	24319
50000	251884	22182
100000	249987	20241
200000	242471	19302

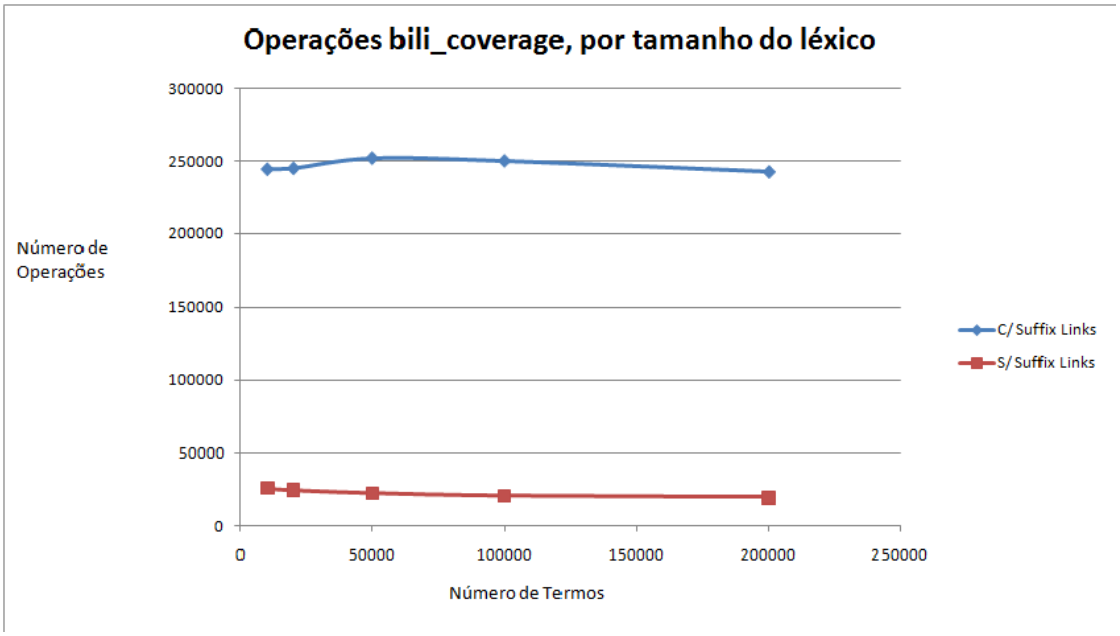

Figura 5.5: Gráfico com resultados da Tabela 5.8

Desta forma, é visível a vantagem do uso das árvores de sufixos, em relação aos *arrays* de sufixos, pois a diferença nos resultados é muito grande. É certo que a segunda implementação definida, apenas simula o comportamento dos *arrays*, mas não foge muito à realidade do que seria uma implementação nessas estruturas de dados.

Em termos de outros protótipos, não foi feita nenhuma comparação por não existirem outros com árvores de sufixos ou outras estruturas, segundo o nosso conhecimento, que possuam o mesmo tipo de funcionalidades.

No Capítulo 3 apresentámos as árvores de sufixos bilingues [MM02], que conseguem definir um alinhamento entre vários corpus, através das correspondências conhecidas no léxico. No LEXMAN, além de representarmos e fazermos a gestão do léxico bilingue, o que não acontece com a implementação de Munteanu et al., temos a

operação de cobertura que consegue determinar os segmentos que não têm correspondência de um par de termos. Além de oferecer informação importante que as árvores de sufixos bilingues não conseguem, a implementação está a um nível mais específico, que é o do termo, ao invés de um corpus completo.

Sendo assim, os testes foram feitos apenas ao nível das estruturas de dados, nomeadamente comparando as operações de cobertura.

6

Considerações Finais

6.1 Trabalho Futuro

Em capítulos anteriores deste relatório, foi referido que as árvores de sufixos não tinham sido muito utilizadas em áreas relacionadas com a linguística. Este trabalho, mostrou contudo, que é benéfico utilizar estas estruturas nesta área, de modo a ser possível aproveitar as características das mesmas, em prol dos projectos que poderão ser desenvolvidos no futuro.

6.1.1 Compressão

O LEXMAN, apesar de pensado especificamente para esta aplicação, pode ser adaptado para outro tipo de projectos. Para isso ser feito de forma eficiente, é natural que seja necessário reduzir ao máximo os recursos consumidos, nomeadamente em termos de memória. As árvores de sufixos requerem geralmente um espaço em memória considerável, além de que para serem úteis, necessitam de estar armazenadas em memória principal. No caso do LEXMAN, como o léxico bilingue não exige muito espaço, são estruturas interessantes. Mas para outro tipo de projectos, que trabalhem com uma quantidade de dados muito maior, esta situação pode ser prejudicial.

Uma das hipóteses para contornar o problema, poderia estar relacionada com a eliminação de alguma informação, que foi necessária para desenvolver esta implementação, mas que poderia não ser tão interessante para outros projectos. No entanto, isso pode implicar mudanças profundas na implementação. Por isso, poderia ser bastante

interessante utilizar uma forma de compressão [Sad07], que permitisse consumir menos recursos de memória, de forma eficiente. Seguindo esta direcção, as novas árvores de sufixos comprimidas [RNO08] podem ser muito úteis.

Com projectos relativamente recentes, que combinam compressão de dados com estas estruturas sucintas, atingiram-se excelentes resultados. Um exemplo de sucesso é o trabalho de Ferragina et al. [FMMN07], com a apresentação de um *array* de sufixos comprimido. Usando este tipo de estruturas comprimidas e com o auxílio de outras estruturas auxiliares, consegue-se representar as operações características das árvores de sufixos, surgindo desta forma as árvores de sufixos comprimidas [RNO08].

6.1.2 Aplicações a Alto Nível

No primeiro capítulo deste relatório, abordaram-se as contribuições que o LEXMAN pode trazer a outras aplicações. Uma delas, está relacionada com a criação de novas aplicações a alto nível, usando a implementação do LEXMAN.

Um exemplo de uma aplicação possível, pode ser uma ferramenta para sugerir novas traduções. Se um léxico não conhecer um segmento de uma expressão multi-palavra, é possível que exista uma forma de propor um possível par de tradução para esse segmento desconhecido, usando todos os outros bocados conhecidos.

Assumindo que X, Y, Z e W são segmentos que compõem uma expressão em inglês, e X', Y', Z' e W' são também quatro segmentos de uma expressão multi-palavra, mas desta vez em português, é possível obter o exemplo da Figura 6.1. Na Figura, as linhas a tracejado mais fino representam ligações de correspondência entre os segmentos, enquanto as linhas horizontais representam uma cobertura monolingue desses segmentos, no léxico da respectiva linguagem. Pelo exemplo, mesmo não conhecendo Z, mediante a correspondência que existe entre todos os outros segmentos e conhecendo-se Z', como seria a parte que faltava ter ligação de correspondência, podia ser feita a sugestão da ligação Z <=> Z', guardando-se essa informação no léxico. Esse par inferido seria posteriormente verificado no passo de validação.

6.1.3 Base de Dados + LEXMAN

A gestão do léxico, antes da criação do sistema, era feita através de uma base de dados relacional. No entanto, não seria muito vantajoso para um sistema desse tipo com as três tabelas, ter uma implementação da operação de cobertura e permitir uma maior extensibilidade para operações futuras. Por isso, o LEXMAN é vantajoso nesse aspecto. Porém, em termos de operação de pesquisa, as bases de dados conseguem ter mais funcionalidades disponíveis do que o LEXMAN, como por exemplo, as pesquisas por

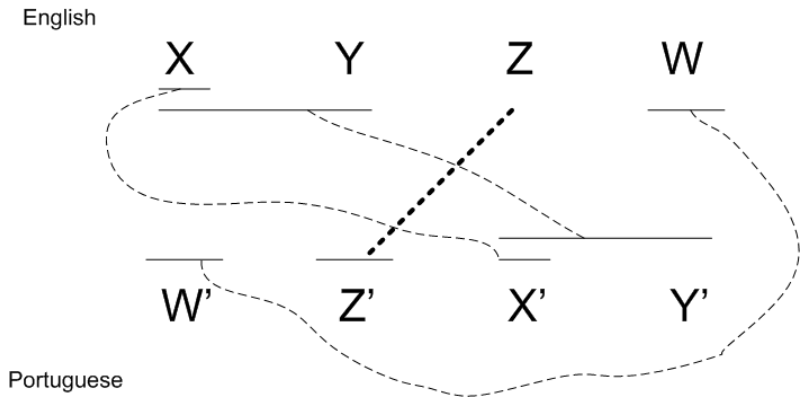

Figura 6.1: Exemplo de sugestão do par Z<=>Z'

texto completo (*full text search*). Este tipo de pesquisa examina todos os termos numa base de dados, à medida que tenta encontrar aqueles que emparelham com as palavras fornecidas pelo utilizador no arranque.

Comparar os dois sistemas não é uma boa opção, pois oferecem um conjunto de opções distintas. Portanto, poderia ser feito um sistema híbrido que juntasse o potencial das bases de dados e do LEXMAN. Claro que isto só seria viável, se valesse a pena manter a informação replicada pelos dois sistemas, mas cremos que poderia ser uma hipótese a estudar. Uma hipótese de sucesso poderia ser o LEXMAN fornecer a cobertura bilingue e monolingue para ajudar na extracção de traduções e alinhamento, enquanto a base de dados poderia dar um suporte para o processo de validação de traduções, através das seguintes funcionalidades:

- Pesquisa, para listar traduções que contenham uma dada expressão X.
- Alteração do estado de cada tradução, podendo indicar se uma dada tradução foi aceite, rejeitada ou ainda se encontra por verificar.
- Guardar dados adicionais, associados a cada tradução, como por exemplo, o resultado obtido pelo classificador no passo de validação, a fonte da tradução onde esta foi extraída, o utilizador que validou a tradução, etc.

6.1.4 Outra Abordagem Para Extracção de Traduções

Actualmente a operação de cobertura, permite descobrir quais os segmentos dos termos não cobertos pelo léxico, podendo-se na extracção tentar extrair esses mesmos pares desconhecidos. Porém, existem outras possibilidades de resultados que se podem obter com a cobertura, de modo a abordar o processo de extracção de traduções de formas diferentes.

Em primeiro lugar, podemos emparelhar os segmentos sem cobertura. Supondo que de um lado estão dois segmentos não cobertos e do outro três, podemos indicar que temos $2x3$ pares de segmentos. Usando o exemplo da Figura 6.1, pode-se tentar extrair novas traduções desses pares de segmentos, usando o que já é conhecido.

Outra metodologia que se pode seguir, é através do cálculo de uma medida de associação para cada par. Essas medidas podem ser o coeficiente de Dice [Dic45], a SCP (probabilidade condicional simétrica) ou uma simples probabilidade condicionada:

- $$Dice(a,b) = \frac{2 * F(a,b)}{(F(a) + F(b))}$$
- $$SCP(a,b) = \frac{F(a,b)^2}{(F(a) + F(b))}$$
- $$P(a|b) = \frac{F(a,b)}{F(b)}$$
- $$P(b|a) = \frac{F(a,b)}{F(a)}$$

onde F(a,b) é a frequência do par (a,b), F(a) é a frequência do segmento a e F(b) a frequência do segmento b.

Estas medidas de associação têm um valor entre 0 e 1, sendo que um valor alto (0.8 por exemplo) indica que existe uma forte possibilidade de serem traduções.

As probabilidades condicionadas, normalmente são mais adequadas do que uma medida simétrica (Dice, SCP), para os casos em que uma expressão tem várias traduções possíveis. Como exemplo, ”European” pode ser traduzida como ”Europeu”, ”Europeia”, ”Europeus”, ”Europeias”, ”da Europa”, etc. No entanto, ”Europeu”, ”Europeia”, ”Europeus”, ”Europeias”, ”da Europa”, são quase sempre traduzidos como ”European”. Assim, a probabilidade condicionada P(”European” | ”Europeu”) daria um valor muito alto, mas P(”Europeu” | ”European”) originaria um valor bem menor.

6.2 Conclusões

O principal objectivo proposto para esta dissertação era a implementação de um sistema, usando árvores de sufixos, para representar um léxico bilingue. Esta implementação teria de permitir a gestão do léxico com tempos lineares, assim como teria de

fornecer algumas operações que pudessem oferecer informação importante, nomeadamente à extracção de traduções e aos outros passos do processo iterativo, do qual o LEXMAN fará parte.

Deste modo, analisando os resultados apresentados, tanto em termos de recursos consumidos, como em termos de complexidade e em termos de resultados comparativos mostrados, pode concluir-se que os objectivos foram cumpridos. À excepção da cobertura bilingue, todas as operações fornecidas, com principal destaque para as de gestão do léxico, têm todas complexidades lineares. No que toca à cobertura bilingue era impossível, com as características das árvores de sufixos e o número de verificações a fazer, conseguir que esta tivesse complexidade mais eficiente. Mesmo assim, os resultados apresentados para esta operação são bastante satisfatórios.

Os benefícios que o sistema traz ao processo iterativo explicado no início do relatório, também são bastante importantes, principalmente pela informação que pode fornecer através da das operações de cobertura, nomeadamente à extracção de traduções e indirectamente ao alinhamento. No que toca à extracção de traduções, os benefícios que a operação de cobertura pode trazer é notório, devido à capacidade que esta operação tem em devolver informação de traduções em falta, permitindo assim que o protótipo de extracção, procure colmatar essas falhas. Melhorando este passo, tanto o alinhamento como a validação saem beneficiados.

É também importante notar todas as hipóteses possíveis de trabalho futuro, que esta implementação permite, mostrando que pode não ser só importante exactamente como está e no presente, como também pode contribuir para novas aplicações e novos projectos inovadores.

Bibliografia

[ABM08] D. Adjeroh, T. Bell, e A. Mukherjee. *The burrows-wheeler transform: data compression, suffix arrays, and pattern matching*. Springer-Verlag New York Inc, 2008.

[ALG09] Aires, G. P. Lopes, e Luis Gomes. Phrase translation extraction from aligned parallel corpora using suffix arrays and related structures. In P. Mariano L. Seabra Lopes, N. Lau e L. M. Rocha, editores, *Progress in Artificial Intelligence*, number 5816 in Lecture Notes in Computer Science, pág. 588–597. Springer-Verlag, 10 2009.

[AMHT00] L. Ahrenberg, M. Merkel, A.S. Hein, e J. Tiedemann. Evaluation of word alignment systems. In *Proceedings of LREC 2000*. Citeseer, 2000.

[BRCR94] P. Bieganski, J. Riedl, JV Cartis, e EF Retzel. Generalized suffix trees for biological sequence data: applications and implementation. In *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences, 1994. Vol. V: Biotechnology Computing*, volume 5, 1994.

[BSTU09] M. Barsky, U. Stege, A. Thomo, e C. Upton. Suffix trees for very large genomic sequences. In *Proceeding of the 18th ACM conference on Information and knowledge management*, pág. 1417–1420. ACM, 2009.

[CBBS05a] C. Callison-Burch, C. Bannard, e J. Schroeder. A compact data structure for searchable translation memories. In *10th European Conference of the Association for Machine Translation (EAMT)*, pág. 59–65. Citeseer, 2005.

[CBBS05b] C. Callison-Burch, C. Bannard, e J. Schroeder. Scaling phrase-based statistical machine translation to larger corpora and longer phrases. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pág. 262. Association for Computational Linguistics, 2005.

[Cha] Tyler Chambers. The internet dictionary project. `http://www.june29.com/IDP/`.

[Cor01] T.H. Cormen. *Introduction to algorithms*. The MIT press, 2001.

[Dic45] L.R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945.

[Far97] M. Farach. Optimal suffix tree construction with large alphabets. In *focs*, pág. 137. Published by the IEEE Computer Society, 1997.

[FMMN07] P. Ferragina, G. Manzini, V. Mäkinen, e G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):20, 2007.

[Fre60] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[FS96] P. Flajolet e R. Sedgewick. *An introduction to the analysis of algorithms*. Addison-Wesley Reading, MA, 1996.

[GAL09] Luís Gomes, Aires, e G. P. Lopes. Parallel texts alignment. In P. Mariano ad L.M. Rocha L. Seabra Lopes, N. Lau, editor, *New Trends in Artificial Intelligence, 14th Portuguese Conference in Arificial Intelligence, EPIA 2009, Aveiro, October, 2009, Proceedings*, pág. 513–524. Universidade de Aveiro, 10 2009.

[GBYS92] G. Gonnet, R. Baeza-Yates, e T. Snider. Information Retrieval: Data Structures and Algorithms, chapter 3: New indices for text: Pat trees and Pat arrays. *Prentice Hall, Upper Saddle River, New Jersey*, 7458:66–82, 1992.

[Gom09] L.M.S. Gomes. Parallel texts alignment. 2009.

[Gus97] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge Univ Pr, 1997.

[Hje] H. Hjelm. Identifying cross language term equivalents using statistical machine translation and distributional association measures. In *Proceedings of Nodalida 2007, the 16th Nordic Conference of Computational Linguistics*. Citeseer.

[IAH07] M. Itagaki, T. Aikawa, e X. He. Automatic validation of terminology translation consistency with statistical method. *Proceedings of MT summit XI*, pág. 269–274, 2007.

[IL05] T. Ildefonso e G. Lopes. Longest sorted sequence algorithm for parallel text alignment. *Computer Aided Systems Theory–EUROCAST 2005*, pág. 81–90, 2005.

[IT99] H. Itoh e H. Tanaka. An efficient method for in memory construction of suffix arrays. In *Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware*, pág. 81. IEEE Computer Society, 1999.

[KLA+01] T. Kasai, G. Lee, H. Arimura, S. Arikawa, e K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching*, pág. 181–192. Springer, 2001.

[Knu73] D.E. Knuth. The art of computer programming. Vol. 3: sorting and searching. *Atmospheric Chemistry & Physics*, 1973.

[LS07] N.J. Larsson e K. Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258–272, 2007.

[McC76] E.M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976.

[Mel99] I.D. Melamed. Bitext maps and alignment via pattern recognition. *Computational Linguistics*, 25(1):130, 1999.

[Mel00] I.D. Melamed. Models of translational equivalence among words. *Computational Linguistics*, 26(2):221–249, 2000.

[MF04] G. Manzini e P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.

[MKMK09] D. Marcu, K. Knight, D.S. Munteanu, e P. Kohen. Building A Translation Lexicon From Comparable, Non-Parallel Corpora, 2009. US Patent App. 12/576,110.

[MM90] U. Manber e G. Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pág. 319–327. Society for Industrial and Applied Mathematics, 1990.

[MM02] D.S. Munteanu e D. Marcu. Processing comparable corpora with bilingual suffix trees. In *Proceedings of the ACL-02 conference on Empirical methods in*

natural language processing-Volume 10, pág. 295. Association for Computational Linguistics, 2002.

[MW02] D. Marcu e W. Wong. A phrase-based, joint probability model for statistical machine translation. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pág. 139. Association for Computational Linguistics, 2002.

[NM07] G. Navarro e V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys (CSUR)*, 39(1):2, 2007.

[ON03] F.J. Och e H. Ney. A systematic comparison of various statistical alignment models. *Computational linguistics*, 29(1):19–51, 2003.

[RDLM01] A. Ribeiro, G. Dias, G. Lopes, e J. Mexia. Cognates alignment. *Machine Translation Summit VIII, Machine Translation in The Information Age*, 287292, 2001.

[RNO08] L. Russo, G. Navarro, e A. Oliveira. Fully-compressed suffix trees. *LATIN 2008: Theoretical Informatics*, pág. 362–373, 2008.

[RPLM00] A. Ribeiro, G. Pereira Lopes, e J. Mexia. Extracting equivalents from aligned parallel texts: Comparison of measures of similarity. *Advances in Artificial Intelligence*, pág. 339–349, 2000.

[Sad07] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

[SS07] K.B. Schürmann e J. Stoye. An incomplex algorithm for fast suffix array construction. *Software: Practice and Experience*, 37(3):309–329, 2007.

[Ukk95] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[Wei73] P. Weiner. Linear pattern matching algorithms. In *IEEE Conference Record of 14th Annual Symposium on Switching and Automata Theory, 1973. SWAT'08*, pág. 1–11, 1973.

Comandos das Operações do Sistema

Quando o LEXMAN é executado, pode receber vários comandos. Após as árvores serem criadas e compostas com termos carregados pelo método add_pairs, o sistema continua a sua execução, à espera de novos comandos. Definimos por isso uma *shell*, que espera por um número ilimitado de comandos, que permitem executar as várias operações que o LEXMAN oferece. Para correr o sistema, é necessário dar o nome de um ficheiro em argumento, para ser efectuado o primeiro carregamento de dados para o léxico. Os comandos são os seguintes:

- **add_pair(t_1,t_2) –** Comando para executar a operação para adicionar um único par ao sistema. Após o comando, a *shell* espera por um espaço antes do primeiro termo e um *tab* antes do segundo. No final, o sistema espera uma mudança de linha, para executar o método e esperar pelo comando seguinte.

  $$\text{add\_pair } t_1 \ \backslash t \ t_2 \ \backslash n$$

- **delete_pair(t_1,t_2) –** Comando para executar a operação que remove um par do sistema. A *shell* espera por uma estrutura semelhante à do comando anterior, ou seja, comando, espaço, primeiro termo, *tab*, segundo termo e mudança de linha.

  $$\text{delete\_pair } t_1 \backslash t \ t_2 \ \backslash n$$

- **is_pair(t_1,t_2) –** Comando para executar a operação que pergunta ao sistema, se dois termos formam um par de tradução. O comando é idêntico aos anteriores.

is_pair t_1 \t 2 \n

- **add_pairs(fn) –** Comando que permite adicionar vários pares ao sistema, a partir de um ficheiro de *input*. Após o comando é esperado um espaço e logo a seguir o nome do ficheiro.

add_pairs fn \n

- **all_pairs (fn) –** Comando que executa a operação de escrita dos termos para um ficheiro. A sua estrutura é idêntica ao anterior.

all_pairs fn \n

- **mono_coverage(lang, term) –** Comando que executa a operação de cobertura monolingue. A seguir ao comando, o sistema espera um espaço e a língua do termo a pesquisar. Os comandos previstos são "one", indicando que o termo está em na primeira língua definida, e "two", para indicar que o termo se encontra na segunda língua. De seguida é esperado um *tab*, seguido do termo e da mudança de linha.

mono_coverage lang \t term \n

- **bili_coverage(t_1,t_2) –** Comando que executa a operação de cobertura bilingue, cujo formato é semelhante ao do add_pair.

bili_coverage t_1 \t t_2 \n

- **quit –** Termina a *shell* e a execução do LEXMAN, após ser feita uma mudança de linha a seguir ao comando.