Relatório final

Algoritmos e Estrutura de Dados

Tiago Roxo, Nº 37032, Eng. Informática

1. Objetivos

Este trabalho consiste em desenvolver um programa que, dado um ficheiro binário de número indeterminado de mensagens, consiga determinar quantas são e, dentro destas, procurar por potenciais palavras perigosas, palavras essas fornecidas por um ficheiro de texto. Para determinar a existência de palavras serão considerados todos os espaçamentos possíveis entre as letras desta, tendo sempre em consideração que o espaçamento tem que ser admissível que a palavra exista dentro da mensagem (não exceder o tamanho da mensagem), e a perigosidade da mensagem será determinada pelo número de palavras perigosas que existem nesta com espaçamento comum.

Os principais objetivos com este trabalho são incutir ao aluno a noção de organização de informação extraída de ficheiros, otimização de código e estimular o aluno a pensar qual a melhor abordagem para conseguir, em tempo útil, mostrar a informação requerida.

A manipulação de ficheiros, em especial binários, armazenamento da informação de ficheiros e, principalmente, gestão da quantidade imensa de espaçamentos possíveis que uma palavra pode tomar e organização de perigosidade de mensagens são alguns dos desafios que o trabalho impôs. Em particular este penúltimo ponto (espaçamento de palavras) requer uma estratégica apropriada, pois a abordagem de força bruta não é abordagem possível (para além de não ser inteligente), devido à quantidade de possibilidades que cada palavra pode ter para existir numa mensagem.

2. Motivação

Este trabalho tem principalmente um interesse académico. No entanto, o conceito de analisar um ficheiro, e consequentemente as mensagens deste, e conseguir encontrar palavras-chave que dificilmente seriam encontradas sem um auxílio de um computador é uma noção importante a ter, nomeadamente no âmbito de investigações criminais. Analisar inúmeras mensagens e conseguir determinar que uma em particular contém um número elevado de palavras-chave com espaçamento mútuo, pode ser indicativo de que é uma mensagem entre criminosos ou terroristas para preparem um ataque. Esta análise pode ser suficiente para analisar os registos criminais entre o recetor e emissor da mensagem e, potencialmente evitar um ataque. Todas as aplicações que possam facilitar o trabalho das autoridades e promover uma maior segurança (neste caso, evitando possíveis atentados ou ataques) é sempre um fator importante, e este trabalho tem este fator associado.

3. Descrição

3.1 Funcionalidades

As funcionalidades do programa, relativamente ao menu, são as esperadas: possibilidade de escolher o ficheiro de mensagens a analisar, possibilidade de escolher o ficheiro de texto com as palavras-chave e pesquisa de mensagens. Após a pesquisa de todas palavras em de todas as mensagens será mostrado um ranking ao utilizador, cujo valor do top (se quer ver as 3 mais perigosas, por exemplo) é à escolha deste. Na secção dos resultados experimentais foi representado o top 5, a título de exemplo.

Para melhor organização do programa, foi usado um ficheiro de extensão .h, onde ficaram guardadas todas as funções usadas ao longo do programa.

3.2 Estrutura do Programa e detalhes da implementação do mesmo

Primeiramente será analisado o ficheiro binário das mensagens. Esta análise será feita pela função *total_mensagens_ficheiro* que receberá o nome do ficheiro a analisar e devolverá uma lista de "Mensagem" (estrutura criada). Uma "Mensagem" tem, entre outras variáveis, um inteiro que guarda a posição da mensagem no ficheiro (informação obtido via ftell(f), sendo f o ficheiro das mensagens), e um inteiro para tamanho da mensagem. Será melhor explicado o funcionamento da função por recurso a um esquema:

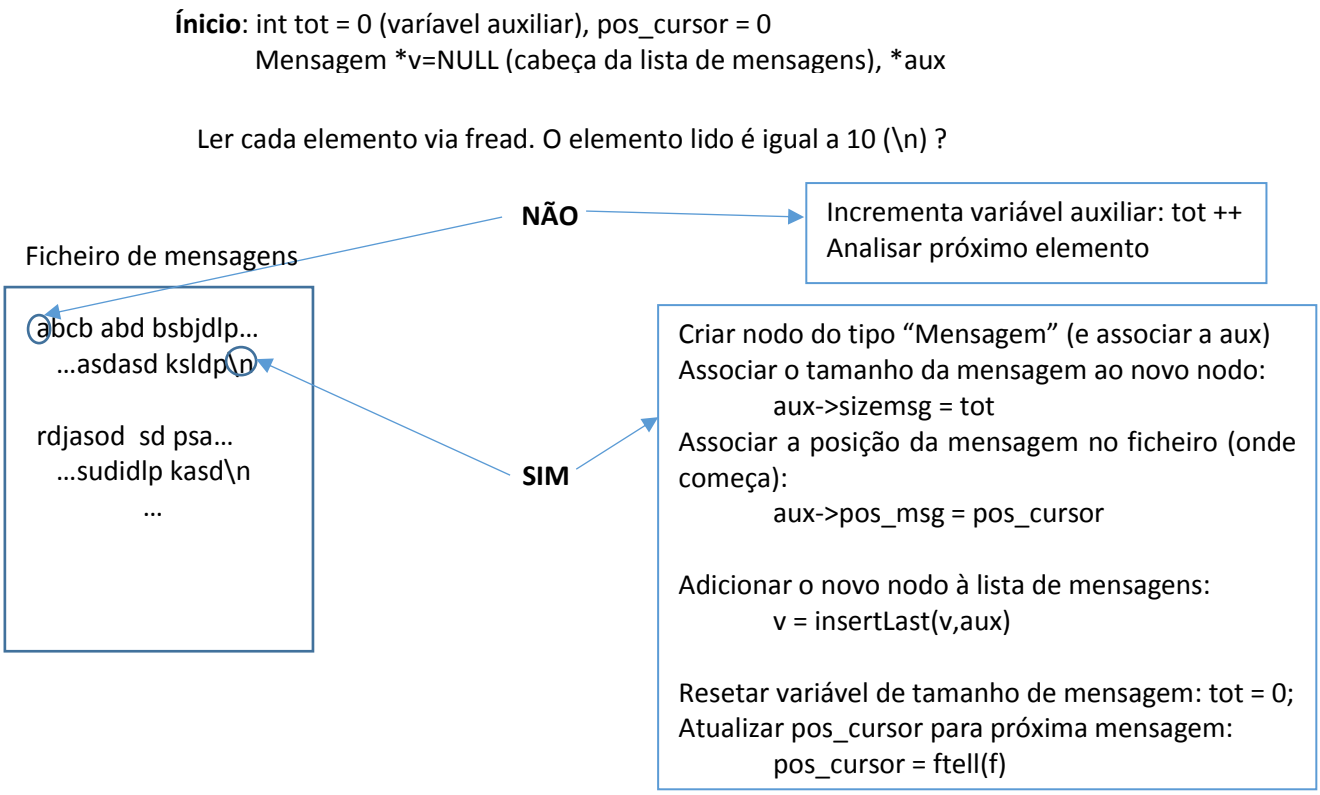


Fig.1 – Esquematização do funcionamento da função de leitura de ficheiro de mensagens

Resumidamente, por ser um ficheiro binário teremos que recorrer à leitura por fread, tendo optado por ler elemento a elemento até encontrar o terminador de mensagem (\n). Quando for encontrado será criado uma nova "Mensagem", e será associado o tamanho da mesma e a sua posição inicial no ficheiro a variáveis desta. Qual a vantagem de guardar estas variáveis e não guardar a mensagem integralmente na estrutura? Principalmente memória. Esta abordagem tem maior importância porque uma das estratégias tomadas para encontrar espaçamentos possíveis de palavras passou por criar um "dicionário" que, resumidamente, guarda todas as posições de cada letra da mensagem (será explicado em maior detalhe mais à frente). Portanto para cada mensagem iriemos sempre precisar de um "dicionário", e se, para além desta matriz de inteiros, guardássemos toda a mensagem também na estrutura, poderia ser problemático em termos de memória se as mensagens fossem muitas, ou muito grandes. Levanta-se as questões: se quisermos analisar uma determinada mensagem precisamos de ir ao ficheiro e extraí-la de lá? Isso não tornará a procura muito mais lento? Sim, teremos que ir ao ficheiro, mas o acesso à mensagem está bastante facilitado pelas variáveis guardadas na estrutura "Mensagem". Simplesmente fazemos um fseek (desde o início do ficheiro até a posição "pos_msg" da estrutura) e guardamos a mensagem de tamanho "sizemsg" (informação guardada numa variável da estrura) num vetor de char temporário, via fread (instruções reunidas numa só função, *le_msg_ficheiro*). Como não foram notadas diferenças significativas entre esta abordagem e o armazenamento integral da mensagem na estrutura, optou-se por tomar a abordagem que seria mais responsável em termos de memória.

Ainda relativamente à figura 1, enquanto forem encontradas mensagens, serão criados novos nodos "Mensagem" sendo continuamente inseridos na lista de mensagens pela ordem de aparecimento no ficheiro. A função para tal foi a insertLast, muito usada ao longo do semestre nas aulas práticas e utilizada neste trabalho com a finalidade de (como o nome sugere) inserir um novo elemento no final da lista.

Outro ficheiro a analisar é o ficheiro das palavras-chave. O esquema abaixo mostra o funcionamento da função com essa finalidade:

Ínicio: char **v = NULL, temp[SIZE_WORD] (#define SIZE_WORD 30)
Int tot=0;

Ficheiro de palavras-chave

ataque\n
explosao\n
…

Procedimento:
Enquanto conseguir extrair palavras (via fgets para temp)
Incremento variável auxiliar: tot++;

Terminado o ficheiro já sabemos o número de palavras-chave. Criar um vetor de vetores de chars de tamanho tot (número de mensagens). Cada vetor de chars terá tamanho SIZE_WORD.

Colocar o cursor no início do ficheiro (rewind(f)) e ler novamente cada palavra, desta vez copiando o conteúdo de temp (fgets(temp,SIZE_WORD,f)) para um vetor de chars de índice i do vetor de vetores de chars, v.

Fig.2 – Esquematização do funcionamento da função de leitura de ficheiro de palavras-chave

O conteúdo do ficheiro das palavras-chave será armazenado num vetor de vetores de chars, e a sua obtenção é feita pelo descrito na figura acima. A estrutura do ficheiro das palavras permitiu o uso de fgets para, primeiramente saber quantas são, e posteriormente armazená-las no vetor de vetores de chars (cada palavra estava separa por \n). Como não foi especificado no enunciado do trabalho o tamanho das palavras a procurar foi definido um tamanho máximo de palavras de 30 caracteres.

Importante referir que a leitura do ficheiro de palavras-chave, no trabalho, ocorre após a criação de "dicionários", tendo sido aqui apresentado por ordem diferente para explicar primeiramente o tratamento dos ficheiros e posteriormente a análise destes. Relativamente ao "dicionário" este será uma matriz de inteiros (vetor de vetores de int) que conterá no índice 0, um vetor com todas as posições da letra 'a', no índice 1, um vetor com todas as posições da letra 'b', e assim sucessivamente. O tamanho do vetor de vetores é conhecido à partida, pois corresponde ao número de letras do alfabeto português (26). O tamanho do vetor de cada índice, corresponde ao número de posições de cada letra, não é sabido. Portanto ao criar um dicionário teremos que, simultaneamente, criar um vetor de inteiros, de tamanho 26 (número de letras), que conterá em cada índice o número de vezes que a letra de igual índice aparece na mensagem. O esquema seguinte ilustra o procedimento implementado:

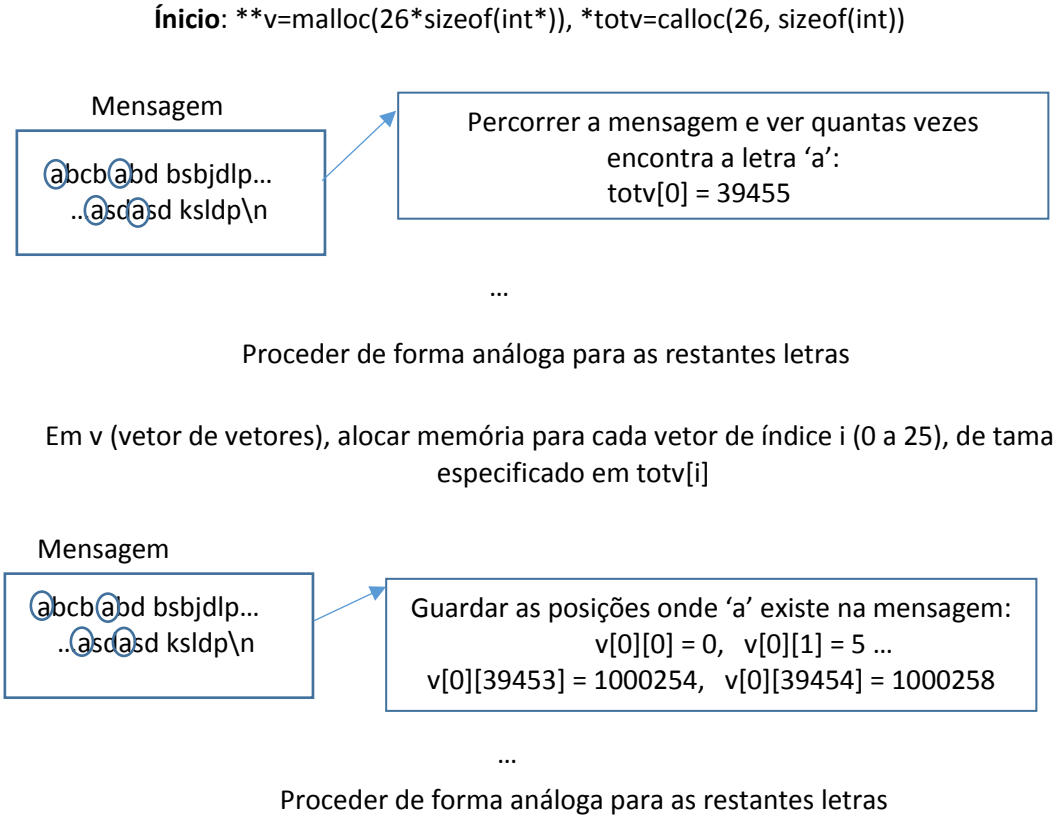


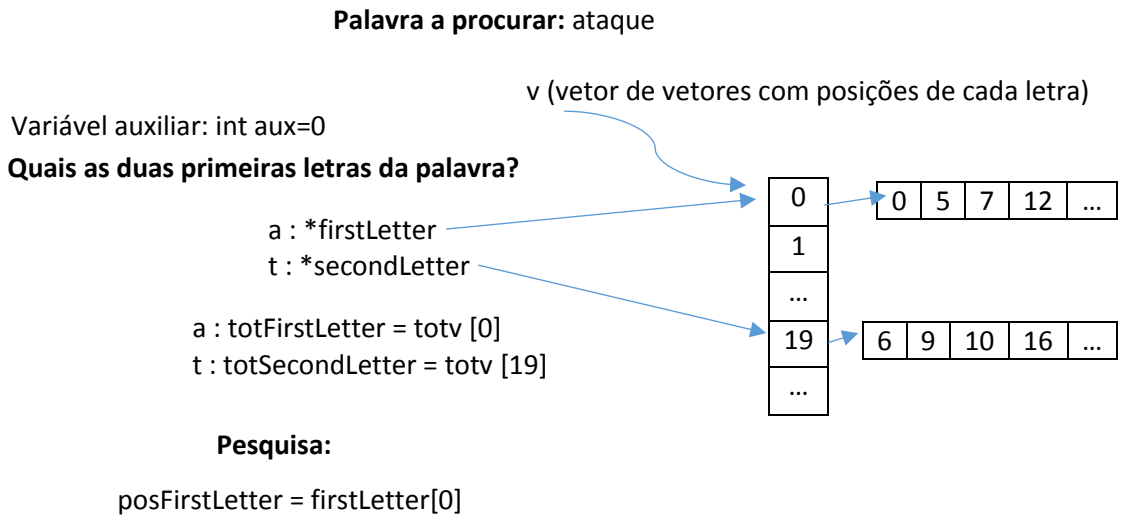
Fig.3 – Esquematização do funcionamento da função de criação de "dicionário" para cada mensagem

A criação de "dicionário" é relativo a cada mensagem, ou seja esta informação terá que ficar armazenada na estrutura "Mensagem". Os nodos criados na função da figura 1, para além das variáveis da posição no ficheiro e tamanho da mensagem, terão outras duas que guardarão a informação do número de vezes que cada letra aparece

na mensagem (int* size_each_char), bem como as posições de cada letra na mensagem (int **dicc). De referir que a procura das posições de cada letra requer que seja conhecido o tamanho da mensagem, informação guardada em “Mensagem”. Este foi outro fator que motivou o não armazenamento integral da mensagem mas sim de informação relativa a onde a encontrar no ficheiro; a variável de tamanho de mensagem era sempre necessária para a criação de “dicionário”.

A criação deste “dicionário” requer a presença da mensagem para poder identificar as posições em que ocorrem cada letra. Como referido anteriormente esta obtenção foi feita com auxílio de variáveis da estrutura “Mensagem” e por recurso à função *le_msg_ficheiro* (detalhes da funções especificados no parágrafo abaixo da figura 1). Após analisar uma mensagem são libertados os recursos, e voltamos a invocar a função para ler do ficheiro a próxima mensagem.

Todo este armazenamento de informação e criação de “dicionários” tem como finalidade facilitar a procura de palavras-chave. Será ilustrado o esquema da função de pesquisa, sendo posteriormente explicado detalhadamente:



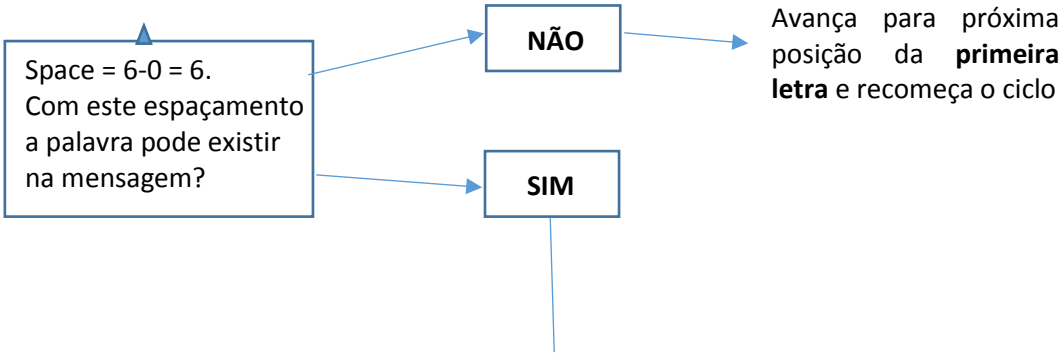
Pesquisa:

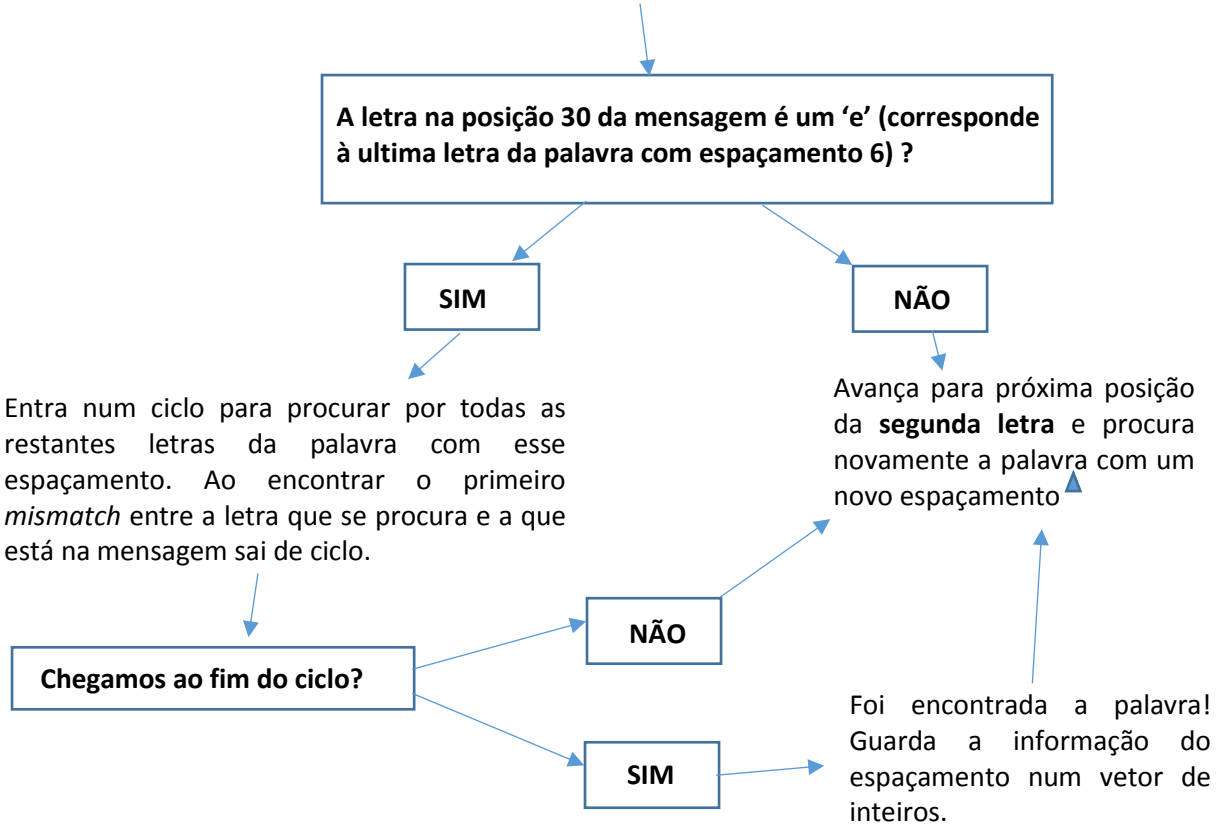
posFirstLetter = firstLetter[0]

Vamos procurar uma posição da segunda letra superior a posFirstLetter:

Neste caso a primeira posição de ‘t’ é superior à posição da primeira letra ‘a’ (6>0)
Vamos guardar a posição da primeira letra de ‘t’ superior à posição da letra ‘a’ em análise, na variável aux: aux = 0;

Começando em aux, vamos percorrer todas as posições possíveis da segunda letra e procurar a palavra ataque:





Proceder de forma análoga até percorrer todas as posições da primeira letra

Fig.4 – Esquematização do funcionamento da função de pesquisa de palavras em mensagem

Nesta função podemos ver a utilidade da criação dos "dicionários" para cada mensagem: ao saber quais são as duas primeiras letras da mensagem sei automaticamente quais as posições em que cada uma delas aparece na mensagem; a criação de apontadores para o "dicionário" (*firstLetter e *secondLetter) permiti-me aceder ao vetor das posições de cada letra. Em que medida isto é benéfico? Como referido sei as posições das duas primeiras letras, mas mais importante é quão valiosa é esta informação e quanto "custou" obtê-la. Relativamente ao "custo" ele foi muitíssimo baixo, na medida em que esta informação foi obtida por criação de "dicionário", um processo que demora o seu tempo (~20s), mas como ocorreu antes de analisar quaisquer palavras, não é contabilizável no tempo de pesquisa. O acesso a esta informação é por acesso direto o que reforça ainda mais o quão baixo o "custo" desta informação foi. No que toca ao valor desta, não é difícil perceber que, sabendo logo à partida (aquando da procura de uma palavra) em que posições estão as duas primeiras letras, sem ter que procurar na mensagem para ver onde elas estão, é uma vantagem enorme. Sem recurso a "dicionário" teria que se percorrer uma a uma cada letra da mensagem e avaliar se era a primeira letra da palavra em análise, o que tornaria o programa inviável pelo tempo que seria necessário para procurar uma palavra. Outra mais-valia associada a saber a posição das duas primeiras letras prende-se em saber quais os espaçamentos possíveis para a palavra em procura: como visto na figura acima, para uma dada posição da primeira letra, procura-se todas as posições possíveis da segunda letra, e a diferença entre estas marca o espaçamento para o qual se irá procurar as restantes letras da palavra.

Pode-se questionar qual a razão pela qual apenas são vistas as posições das primeiras duas e não das restantes, uma vez que temos o "dicionário" para todas as letras. As duas primeiras são fundamentais para encontrar espaçamentos possíveis para a existência da palavra na mensagem. Encontrado um espaçamento precisamos de saber se existem as restantes letras na mensagem nas posições correspondentes ao espaçamento em causa, associada à posição da letra na palavra (3ª letra será posição da primeira letra mais duas vezes o espaçamento). No exemplo da figura acima, para procurar pela 3ª letra da palavra teríamos que procurar no vetor de índice 0 (correspondente à letra 'a') se existia o número 12 (0 mais 2 vezes espaçamento 6). No "dicionário" temos essa informação mas ter que percorrer o vetor para procurar pela posição em questão demora algum tempo e fazer isto milhões de vezes, traduz-me num programa mais lento. A alternativa foi ter a mensagem presente nesta função e ir diretamente à posição 12 e verificar se a letra presente era um 'a', para o exemplo em questão. O acesso direto desta abordagem e evitar percorrer vetores à procura de um valor justificam a abordagem tomada.

Outro pormenor na função esquematizada na figura 4 é a variável 'aux'. Ela guarda a informação da primeira posição da segunda letra que é superior à posição da primeira letra do momento. Esta variável é muito importante na redução do número de iterações que é preciso fazer para procurar uma posição da segunda letra superior à primeira cada vez que esta primeira avança de posição. Imaginemos o caso em que a posição da primeira letra é 987, e corresponde à posição 101000 na mensagem, e a primeira posição da segunda letra superior a esta é 745 e corresponde à posição 101011 da mensagem (guardamos o valor 745 em 'aux'). Calculamos o espaçamento, procuramos pela palavra, e avançamos as posições da segunda letra até que a diferença entre a posição da segunda letra e a posição da primeira seja demasiado grande para que a palavra posso existir dentro da mensagem. Como ilustrado na figura, neste caso avançamos uma posição da primeira letra. Digamos que a posição seguinte da primeira letra (988) corresponde à posição 101111 da mensagem. Ora para que o espaçamento tenha sido tão grande que a palavra não podia existir na mensagem, teríamos a posição da segunda letra à volta dos 400000 (depende do tamanho da palavra, mas a título de exemplo vamos considerar este valor). Qual é a primeira posição da segunda letra que é superior à nova posição da primeira letra (101111)? Sem 'aux', teríamos que percorrer todo o vetor das posições da segunda letra desde o início até à desejada. Com 'aux', começamos a procura na posição guardada em si (745) e avançamos até encontrar uma que tenha valor superior 101111. É facilmente percetível que o número de iterações neste caso é substancialmente menor, pelo que o uso de 'aux' é fortemente justificável.

Na figura a pergunta "com este espaçamento a palavra pode existir na mensagem?", refere-se à situação em que, se a posição da primeira letra mais o espaçamento vezes o tamanho da palavra menos um resultar numa posição que excede o tamanho da mensagem, significa que não faz sentido procurar esse espaçamento e qualquer um superior a esse, razão pela qual se avança a posição da primeira letra e se reajusta a posição da segunda letra, recomeçando o ciclo de pesquisa.

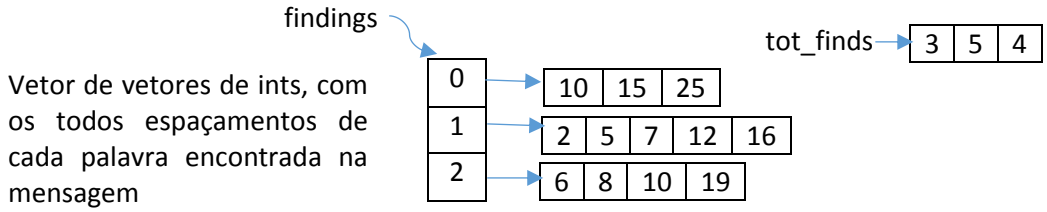
Definido um espaçamento, a primeira pergunta que se faz é se este espaçamento permite que a palavra exista na mensagem, já explicado. A segunda pergunta é se, dado esse espaçamento, a última letra da palavra se encontra na mensagem na posição correspondente face ao espaçamento obtido (No exemplo, se na posição 30 existia um 'e', pois a palavra em procura era 'ataque', o espaçamento 6 e a posição da primeira letra era 0). Esta abordagem por si não reduz substancialmente o

número de iterações do ciclo seguinte (verifica se as restantes letras para além das 1ª, 2ª e última estão na mensagem nas posições correspondentes ao espaçamento), mas foi implementada ainda assim. Dado um espaçamento e sabendo que a última letra existe com esse espaçamento, criamos uma "janela" na mensagem onde a palavra pode existir. A razão pela qual não reduz consideravelmente o número de iterações do ciclo seguinte prende-se em que o ciclo em questão é quebrado caso a posição da mensagem não tenha a letra em procura; basta uma posição não se verificar para ser interrompido (não há possibilidade de a palavra existir nestas condições, espaçamento e posição da primeira letra, na mensagem). Ou seja, ao analisar se a última letra da palavra existe na posição da mensagem (de acordo com o espaçamento), estamos implicitamente a realizar uma iteração do ciclo que vem a seguir.

Percorrendo o último ciclo que avalia as restantes letras da palavra (excetuando a 1ª, 2ª e última), caso consigamos chegar ao fim (pois o ciclo é interrompido em caso de *mismatch* entre letra a procurar e letra da mensagem na posição em análise), encontrou-se a palavra. Nesse caso guarda-se a informação do espaçamento com que a palavra foi encontrada um vetor de inteiros, que será realocado (e aumentado de tamanho) cada vez que se encontrar uma palavra.

Após analisar todas as palavras numa mensagem terá que se organizar a informação e associar um número de palavras encontradas à mensagem analisada. Para tal usou-se a função *compara_pesquisas*, ilustrada abaixo:

Após análise da mensagem 1



Averiguar se existem espaçamentos em comum entre as palavras:

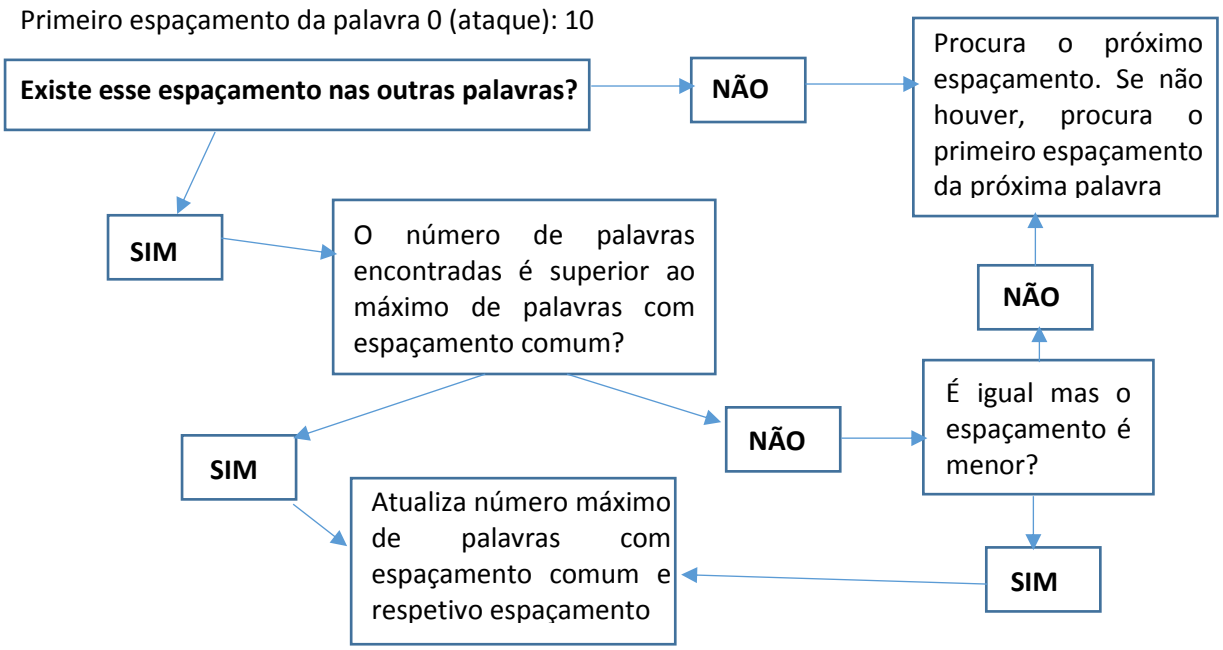


Fig.5 – Esquematização do funcionamento da função de comparação de espaçamentos entre palavras

Já havido sido referido que a função de pesquisa (fig.4) retorna um vetor de inteiros correspondente aos espaçamentos com que a palavra foi encontrada na mensagem. Não foi dito mas a figura 5 ilustra, é que este vetor ficou associado a um índice num outro vetor (findings); ou seja, no caso da primeira palavra (ataque, no exemplo da fig.4), todos os espaçamentos com que a palavra foi encontrada ficaram associadas ao índice 0 do vetor findings. Como o número de espaçamentos é variável, à semelhança na criação do "dicionário", foi criado um vetor de inteiros cuja finalidade é fornecer informação de quantos espaçamentos cada palavra teve (tot_finds). Não ilustrado especificamente na figura 5, mas antes de comparar espaçamentos entre palavras procedeu-se à ordenação dos espaçamentos de cada palavra; este passo antes da comparação permite reduzir o número de iterações na comparação propriamente dita, pois se procuramos por um espaçamento 500, e ao percorrer um vetor de espaçamento de uma palavra encontrarmos espaçamento 545, podemos interromper a pesquisa pois não existe espaçamento 500 para essa palavra; caso não tivesse sido organizado teríamos que, obrigatoriamente, percorrer todo o vetor de espaçamentos para poder obter uma resposta.

Ao encontrar palavras com espaçamento comum, comparamos com o máximo até então (número de palavras com mesmo espaçamento) e caso seja maior atualiza-se a informação. Teve-se como critério que, em caso de igualdade de número de palavras com mesmo espaçamento, fica registado aquelas que tiverem menor espaçamento. Por exemplo, se forem encontradas 'ataque' e 'explosao' com espaçamento 51 e encontradas 'comboio' e 'covilha' com espaçamento 25, ficará registado que a mensagem em questão tem duas palavras-chave com espaçamento 25. Não é óbvio que por ter menor espaçamento entre palavras a mensagem seja mais perigosa, mas teve que ser escolhido um critério.

Finalizado a comparação entre espaçamentos das palavras, será criado um vetor de inteiros, correspondente às palavras que partilham o espaçamento comum; essa informação bem como o tamanho do vetor (número de palavras com espaçamento comum) e o espaçamento com que foram encontradas serão devolvidas por esta função e armazenadas na estrutura "Mensagem" (retornado o vetor, as restantes devolvidas por parâmetro). No exemplo da figura 5, seria retornado um vetor de tamanho dois, com o valor 0 no índice 0 e valor 2 no índice 1, e o espaçamento seria 10. Os inteiros 0 e 2 (correspondentes às palavras-chave de espaçamento comum), aquando da mostragem das mensagens mais perigosas (ranking das mensagens), será feita correspondência com as palavras-chaves (0 seria a primeira palavra-chave, ataque neste exemplo, e 2 seria a terceira palavra-chave).

Após realizar a pesquisa e a comparação de espaçamentos para cada mensagem terá que se proceder à ordenação da perigosidade das mensagens. Esta ordenação foi possível por recurso à função *ordena_ranks,* que cria um vetor de estruturas "Rank", que contém dois inteiros (um para guardar o número da mensagem e outro para registar o número de palavras encontrada com espaçamento comum), que percorre a lista de mensagens e armazena o valor do número de palavras-chave encontradas com espaçamento comum em cada mensagem, na sua variável "total", (informação presente na variável total_palavras de "Mensagem"). Ao ordenar este vetor de "Ranks", via insertSort, ficaremos com uma "lista" das mensagens mais perigosas, ordenadas por ordem decrescente do seu número de palavras-chave. Para mostrar a

informação ao utilizador sobre as mensagens mais perigosas basta percorrer o vetor de "Ranks" e mostrar as informações relativamente à perigosidade da mensagem. Importante referir que, ao percorrer este vetor, estamos a mostrar as mensagens das mais perigosas para as menos, mas o vetor em si só "sabe" qual o número da mensagem e quantas palavras tem. No entanto, ao mostrar ao utilizador é importante mostrar, para além destas duas informações, quais as palavras e qual o espaçamento. Para tal necessitamos de aceder à mensagem em questão, e aceder às variáveis respetivas para tal (*palavras_encontradas e spacing de "Mensagem"). As mensagens não têm associadas a si qual o seu número, mas como a sua introdução foi feita de forma sequencial pelo seu aparecimento no ficheiro, simplesmente fez-se uma função auxiliar que percorre a lista de mensagens e retorna o apontador para a mensagem correspondente ao número em procura.

4. Resultados experimentais

Os seguintes testes foram feitos em Windows 8.1, com um Samsung de processador Intel® Core™ i7- 3630QM CPU 2.40GHZ, com 8GB RAM. O programa em C foi criado e compilado por recurso a CodeBlocks. O registo do tempo de pesquisa foi feito pelo programa, por recurso a biblioteca <time.h>.

Palavras-chave	Tempo de pesquisa (s)	Tempo médio por palavra (s)	Tamanho médio da palavra	Resultados (Top 5)
explosao ataque covilha comboio carruagem	286.516	57.3	7.4	1: 10, ataque, covilha (50935) 2: 46, ataque (6) 3: 82, ataque (13) 4: 29, ataque (15) 5: 5, ataque (20)
ataque bomba comboio carruagem covilha	340.828	68.17	6.8	1: 17, ataque, bomba (102) 2: 25, ataque, bomba (155) 3: 91, ataque, bomba (231) 4: 33, ataque, bomba (289) 5: 42, ataque, bomba (301)
ataque nove bomba prego comboio dois carruagem um meio dia hoje	174.394	34.89	12	Nenhuma palavra encontrada
ataque na covilha explosao hoje ataque as nove estejam atentos bomba de pregos	136.633	27.33	15	Nenhuma palavra encontrada
ataque explosao policia bombista covilha comboio carruagem madrugada cuidado vinganca	580.734	58.07	7.6	1: 10, ataque, covilha (50935) 2: 83, ataque, cuidado (72246) 3: 52, ataque, policia (149957) 4: 46, ataque (6) 5: 82, ataque (13)

Uma rápida análise à tabela de resultados mostra que o tamanho das palavras influencia o tempo de pesquisa desta. Algo que seria de esperar, pois uma palavra pequena tem mais espaçamentos possíveis, e portanto o número de iterações que a função de pesquisa terá que fazer para analisar todas as possibilidades de existência da palavra na mensagem será maior. A última linha mostra uma pesquisa de 10 palavras, ao invés de 5, e verifica-se que o tempo médio despendido por palavra nas 100 mensagens é aproximadamente igual ao observado para 5 palavras (em ambos os casos o tamanho médio das palavras é aproximadamente igual). O resultado não é surpreendente pois a análise de mais palavras por mensagem não influencia o tempo de análise despendido por palavra.

Algo que também é observável na tabela, é que existem muito poucos espaçamentos em comum entre as palavras. Embora todas sejam invariavelmente encontradas nas mensagens, poucos são os espaçamentos partilhados por várias palavras. De facto, as pesquisas que mostram mais resultados em comum, são de palavras pequenas, o que seria de esperar pois são as que têm mais espaçamento encontrados e portanto as que têm maior probabilidade de ter espaçamento comum com outras palavras.

O caso particular da segunda linha mostra um tempo médio por palavra superior à primeira linha (simplesmente a troca de uma palavra levou a um ganho de 10s na pesquisa). Tal deveu-se à introdução da palavra ‘bomba’, uma palavra que tem um número de espaçamentos encontrados médio por mensagem na ordem dos 8000-9000. É a palavra mais pequena analisada ao longo das pesquisas registadas, e o elevado número de espaçamentos possíveis leva a um maior tempo de pesquisa, mas como seria de esperar, no top 5 das mensagens mais perigosas, a palavra bomba aparece sempre.

À semelhança do que ocorre na função *compara_pesquisas,* na ordenação das mensagens esta é feita, primeiramente pelo número total de palavras-chave na mensagem e, em caso igualdade, será ordenada de forma crescente do espaçamento da palavra encontrada. Isso é verificável em todas as colunas da tabela, onde se verifica que, em caso de igualdade do número de palavras, a mensagem mais perigosa é a que tem menor espaçamento comum entre palavras. Tal como referido anteriormente, ter menor espaçamento não é implicativo de maior perigosidade, mas teve que ser tomado um critério de ordenação de perigosidade das mensagens e este pareceu ser o mais justo.

5. Principais dificuldades

As principais dificuldades deste trabalho foram principalmente ao nível da pesquisa. Inicialmente, na criação do “dicionário” optou-se por fazer via listas e listas de listas, ao invés da matriz implementada no trabalho final. Esta opção revelou-se pouco eficiente pois o acesso por listas é mais demorado e o número de posições de cada letra não era elevado o suficiente para se justificar um armazenamento por lista. A implementação da matriz melhorou significativamente a rapidez do programa, pois o acesso a informações desta é feito por acesso direto.

Relativamente à pesquisa propriamente dita, tentou-se reduzir ao máximo o número de iterações possível. O uso da variável ‘aux’ (explicada em melhor detalhe no texto abaixo da figura 4), melhorou significativamente a rapidez do programa, pois

evitava que a procura de uma posição da segunda letra superior à primeira fizesse iterações desnecessárias. A declaração de várias outras variáveis tiverem um intuito de também diminuir o tempo despendido durante a pesquisa. A título de exemplo, declarar posFirstLetter=firstLetter[i] ou lastChar = word[sizeWord-1], têm a finalidade de minimizar o tempo perdido, na medida em que em vez de aceder a uma posição do vetor para obter um valor, associou-se este a uma variável e usou-se esta nos ciclos seguintes. Todos os passos tomados foram com o intuito de reduzir o máximo de tempo durante a pesquisa, pois esse foi o maior obstáculo no trabalho.

6. Melhorias ao programa

A principal melhoria que se poderia implementar ao programa seria uma diminuição do tempo de pesquisa. Embora tenha tentado minimizar ao máximo o tempo perdido durante a pesquisa, é de crer que obter um tempo de 58s para encontrar uma palavra de cerca de 6-7 letras ao longo das 100 mensagens, é passível de ser diminuído.

Aquando da pesquisa e comparação de espaçamentos entre palavras é tomado o critério de, em caso de igualdade no número de palavras (quer na espaçamento da pesquisa, quer nos espaçamentos entre palavras na função *compara_pesquisas*) o menor espaçamento é o registado. Seguindo o mesmo exemplo do texto abaixo da figura 5 (‘ataque’ e ‘explosao’ com espaçamento 51, ‘comboio’ e ‘covilha’ com espaçamento 25), ficaria associado à mensagem, ‘comboio’ e ‘covilha’, como as palavras-chave encontradas e o espaçamento 25. Uma possível melhoria seria guardar todas as ocorrências em que o número de palavras com espaçamento comum fosse superior a 1, e apresentá-las ao utilizador, ficando na sua decisão de escolher qual o par de palavras que achava mais pertinente para a perigosidade da mensagem. Se tivéssemos à procura de um possível ataque com uma bomba, mas que sabíamos que iria ocorrer em Lisboa, o segundo par de palavras-chave, embora tenha menor espaçamento, é menos direcionado para o que se procurava.

7. Conclusão

O programa reuniu todos os requisitos impostos para o trabalho, lendo o ficheiro binário e analisando as mensagens deste na procura de palavras-chave (fornecidas num ficheiro de texto), analisando todos os espaçamentos possíveis, e registando aqueles em que a palavra existe; findo a pesquisa de todas as palavras numa mensagem, analisou-se se existia um espaçamento comum entre estas, tendo sido registado o espaçamento comum menor ao maior número de palavras. As estratégias usadas para armazenar informação e auxiliar à pesquisa (criação de “dicionário”) revelaram-se bastante valiosas na redução do tempo despendido nesta; ainda assim, o tempo médio obtido por palavra, na análise das 100 mensagens, ficou aquém do esperado. Seria necessário repensar uma nova abordagem a este problema, visando diminuir ainda mais esse tempo. A ordenação do ranking das mensagens mais perigosas foi feito com recurso a uma “lista” (vetor de “Ranks”) que continha a informação necessário para ordenar as mensagens por perigosidade: número da mensagem e número de palavras-chave encontradas com espaçamento comum. Ao apresentar o ranking ao utilizador, para além desta duas informações, mostrou-se quais as palavras e qual o espaçamento comum destas, informações essas guardadas em variáveis na lista de “Mensagem” (*palavras_encontradas e spacing, respetivamente).