**I. Descrição do problema**

Nos foi dado um problema em que deviamos, a partir de um texto, devolver os N pares mais frequentes de palavras vizinhas dentro do mesmo. O programa que tivemos que escrever recebe como entrada um ficheiro contendo não só o texto, mas também dois números no início, que representam o número (N) de pares mais frequentes que devemos imprimir, e o número de palavras diferentes dentro desse texto.

Nesse texto, os pares poderiam estar separados por espaços, *new lines*, pontuações, ou qualquer outro elemento que não fosse um número ou um caracter, sendo assim, os pares poderiam ser do modo (palavra, palavra), (palavra, número), (número, palavra), ou (número, número).

A nossa solução, ou seja, os N pares mais frequentes palavras vizinhas, deveriam ser impressos em um ficheiro de saída, no qual o nome é o mesmo que o ficheiro de entrada, mas com a extensão “.sol” em vez do “.txt”. O formato de saida de cada par foi “X – Y palavra1 palavra2”, sendo X o ranking do par (por exemplo: 1 se for o mais frequente), e Y a frequência de aparição.

Diferentes resultados poderiam ter sido obtidos na saída de um mesmo texto, isso porque podem ocorrer empates nas frequências dos pares, e desta forma o resultado depende de como o programa manipulou os dados dentre dele.

**II. Resolução da sequência de palavras**

Para obtermos o resultado desejado, percorremos o texto todo uma vez, desta forma a obter todos os pares de palavras. Sempre que obtido um par, o mesmo é adicionado em uma *hash table* previamente criada. Após termos a *hash table* completa com todos os pares, percorremo-la toda a com o objetivo de identificar os N elementos mais frequentes, adicionando-os em uma tabela auxiliar, da qual é ordenada utilizando o algoritmo *bubble sort*.

**III. Arquitectura do programa**

O programa foi estruturado basicamente em 3 partes: a obtenção dos dados do ficheiro (nome, número de pares a serem impressos, e número de palavras diferentes), a criação da estrutura de dados principal de armazenamento dos pares (juntamente com a leitura do texto), e por final a análise dessa estrutura, que inclui a organização desses dados e a impressão no ficheiro de saida.

**Obtenção dos dados**

Nesta parte foram criadas funções específicas para cada dado a ser obtido. Para o nome, temos a função *get\_name*, que recebe como argumento o nome do programa, retira o “.txt.”, e adiciona “.sol”. Para os valores númericos (número de pares, e número de palavras) foi criada a função *get\_arguments*, em que lê as duas primeiras linhas do ficheiro dado.

**Criação da estrutura**

Para criarmos a *hash table*, fizemos a função *insertion*, que insere os pares na *hash table*. A função busca palavra a palavra, ignorando os caracteres especiais, e após obter duas, chama a função *insert­­\_in\_hash*, que cria a chave (soma do valor ASCII dos caracteres das palavras) para a inserção na tabela. Esta última função, após criar a chave, chama a função *insert\_block\_hash*, que cria e insere o bloco na *hash table*, ou, caso o bloco já exista, incrementa 1 na variável que armazena a frequência de aparição no texto.

**Análise dos dados:**

Com a *hash table* totalmente criada, passamos para a função *top­\_n\_function*, que é onde se desenvolve a resolução do problema. Nela, criamos uma estrutura auxiliar semelhante à nossa tabela principal, mas que terá apenas os N elementos mais frequentes, e percorremos a *hash table* toda analisando e inserindo os elementos nessa outra estrutura. O algoritmo utilizado para ordenar essa estrutura é o *bubblesort*, e é executado toda vez que um elemento é inserido nela. Após percorrida toda a *hash table* e a estrutura auxiliar completa, chamamos a função *writes\_in\_file* que imprime a estrutura auxiliar no ficheiro de saída.

A seguir temos um fluxograma que demonstra o funcionamento geral do programa:

A close up of a map

Description automatically generated

**IV. Estruturas de dados**

## Hash table

A estrutura de dados utilizada é uma *hash table*, a chave da *hash table* é a soma do código ASCII das duas palavras.

O que é inserido na estrutura de dados são números e letras minúsculas, ou seja, do código ‘0’ – 48, ao código ‘z’ – 122. Ao não tratar do código que obtemos, alocaremos mais memoria do que necessitamos. Para resolver este problema, o que se fez foi subtrair a cada caracter que se lê o código de 0, ou seja 48 porque o 0 vai ser o numero com menor valor na tabela ASCII que se vai usar, assim poupa-se alguma memória.

Para saber o tamanho da tabela que se vai utilizar considera-se a palavra com a maior soma do código ASCII, e isto vai ser o número com maior peso, ou seja o ‘z’ (subtrai-se pelo ‘0’), multiplica-se pela dimensão das strings auxiliares que tem um valor pré definido, e multiplica-se por dois, no nosso código está indicado como:

Isto porque poderíamos ter duas strings totalmente preenchidas de z’s, e esse par tem o maior código ASCII.

A *hash table* apresenta várias vantagens na inserção, e como neste projeto a inserção faz parte de maior parte do programa, decidiu-se que era boa ideia utilizar uma *hash table*.

## Variável struct

A nossa variável de estrutura contém dois apontadores para o mesmo tipo de estrutura, anterior e seguinte, dois apontadores para a sequência de palavras, palavra um e palavra dois, e um contador.

Cada vez que se sabe que não existe uma sequência de palavras que queremos inserir na nossa estrutura, alocamos memoria com as duas palavras, inserimos na lista (caso haja colisão), e iguala-se o contador a 1.

Se se encontrar a mesma sequência de palavras que queremos inserir em um destes blocos, simplesmente incrementamos o contador.

## Colisões

Na nossa *hash table* existem vários casos em que pode haver colisões, pares de palavras em que a soma de todos os caracteres seja igual. A nossa solução para este problema foi declarar a *hash table* como um vetor de apontadores e utilizar uma lista duplamente ligada em cada uma das posições, a lista só era criada no momento em que uma sequência não tinha colisão. A *hash table* era toda inicializada a NULL.

Foi escolhida esta solução porque poderia haver um maior número de palavras do que a própria dimensão da *hash table*, ou então direcionar o par de palavras para outra posição poderia resultar em maus resultados, em termos de tempo, no caso de existirem várias colisões.

## Melhorias que podiam ter sido feitas:

**Distribuição**

A nossa *hash table* não tem muito boa distribuição, por exemplo, a ultima posição da *hash table* só e usada para uma sequência de palavras (duas strings exclusivamente com z’s e de dimensão máxima), mas tem-se noção que existem menos sequências que se localizam no início e no final da *hash table*, do que no meio. Também se sabe que no início da *hash table* existem mais combinações do que no final. Por exemplo a última posição só pode incluir as duas strings totais de z’s, enquanto a primeira posição aceita as sequencias (0,0), (00,0), (0…,0), (0…,0…). Poderia se ter feito uma função de redistribuição que pega nas chaves mais aproximadas do meio da *hash table* e incrementa ou decrementa para que a inserção seja mais feita nos extremos. Esta operação iria demorar tempo, mas acaba por salvar tempo na inserção.

**Insertion Sort**

Como não foi decidido fazer uma função de distribuição, pôs-se em causa uma função que cada vez que se incrementa o contador de um dos blocos, realiza-se um insertion sort para que a palavra mais frequente ficasse no inicio da lista, o que ajudaria ligeiramente na inserção, e quando queremos saber as top N sequências mais frequentes, essa organização compensaria o custo dessa mesma, pois a nossa *hash table* deve ter algumas colisões.

**V. Algoritmos usados**

Para descobrirmos as top n sequências que nos aparecem no ficheiro, primeiro alocámos um vetor de apontadores de dimensão n, e iniciamos todas as posições a NULL. De seguida percorremos a *hash table* e cada uma das listas da *hash table*. Sempre que encontramos um bloco, comparamos com o elemento da posição zero do vetor, se fosse igual a NULL, ou se o contador da estrutura fosse menos do que o que nos encontramos, substituíamos e realizava-se *bubble sort*. As posições que estavam a NULL eram equivalentes a ter um bloco com um contador a 0, seriam sempre trocadas para o fim o vetor a partir do bubble sort, e seriam completamente substituídas se o número de pares fosse maior ou igual aos top n pares pedidos.

As desvantagens que temos deriva do facto que nós percorremos todas as listas de todas as posições da *hash table*, e isso custa-nos algum tempo.

## Melhorias:

Se tivéssemos implementados o insertion sort nas listas, não teríamos que precorrer toda a estrutura. Assim que um elemento da lista tivesse menos frequência do que o que tem menos frequência nos top n, poderia se desprezar o resto da lista porque os outros elementos teriam frequência menor ou igual ao que não trocamos.

**VI. Análise dos Requisitos Computacionais**

A *hash table* é uma estrutura que ocupa muita memória, visto que alloca endereços independentemente da quantidade de dados recebida, para depois allocar a memória específica para cada bloco. Pensamos em utilizar uma lista de estruturas duplamente ligada, por ser a melhor opção em questão de memória, visto que alloca apenas a quantidade necessária para guardar os pares, porém o tempo de execução do programa seria muito maior, mais precisamente na parte da inserção na lista, considerando que para cada par, teriamos que verificar nela toda se já existia o par de palavras.

Em questão de tempo de execução, neste problema a parte que tem mais influência é a parte da inserção, visto que não é apenas uma inserção, mas há também uma busca em toda a estrutura por um bloco igual que já exista, ou seja, quanto maior os dados de entrada, mais demorará a ser executado. No caso da lista duplamente ligada (que ocuparia menos memória), para cada par, o programa iria percorrer a estrutura inteira, sendo assim a ordem seria de O(N²) (para N pares, percorre N vezes). Já para as *hash tables*, o programa só busca na lista de elementos em que há conlfito, ou seja, cuja chave é a mesma. Como os conflitos não são muito frequentes por conta do grande tamanho da estrutura, desse modo temos algo próximo de O(N) (para N pares, percorre 1 vez), o que é muito mais rápido.

**VII. Exemplo de aplicação**

Para medirmos a memória utilizada pelo programa, usamos o *valgrind*, e para medir o tempo de execução, usamos o comando *time* no terminal do sistema operativo Linux.

Para a realização do teste, foi utilizado um site gerador de textos aleatórios [https://randomtextgenerator.com/], em que foram gerados não só um, mas vários textos, e todos eles colados em ficheiro. Para termos o número de palavras diferentes nessa coletânea de textos, foi utilizado um site que realiza essa função [https://planetcalc.com/3205/].

Foi obtida a seguinte saída:

A screenshot of a computer

Description automatically generated

E os seguintes recursos computacionais:

|  |  |  |  |
| --- | --- | --- | --- |
| **Número de pares** | **Número de palavras** | **Tempo (s)** | **Memória (B)** |
| 10 | 1245 | 0.012 | 380899 |

Para os textos fornecidos pelo professor como exemplo, obtivemos os seguintes resultados:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Texto** | **Número de pares** | **Número de palavras** | **Tempo (s)** | **Memória (B)** |
| 00 | 6 | 85 | 0.003 | 15781 |
| 01 | 2 | 9 | 0.003 | 9829 |
| 02 | 10 | 173 | 0.004 | 24156 |
| 03 | 15 | 170 | 0.004 | 42217 |
| 04 | 20 | 949 | 0.005 | 85340 |
| 05 | 25 | 766 | 0.006 | 107882 |