Terceira Fase do Trabalho de Recuperação de Informação — v.30/11/2020

Caros alunos,

Na terceira fase do trabalho vocês devem estender o buscador do *My Information Retrieval* construído na segunda fase de forma a introduzir um critério de ordenação dos resultados e mostrar um pequeno trecho dos arquivos que contêm os termos da consulta.

Até a segunda fase de nosso trabalho, os resultados da conjunção dos termos da consulta têm sido listados segundo a ordem oferecida pelos identificadores dos arquivos, a chave de ordenação das listas de incidências. Este é o critério `0` de ordenação.

O capítulo 6º exibe dois critérios de ordenação dos resultados e o 8º discute o uso de sumários dinâmicos para exibi-los. O mais conhecido destes métodos de ordenação é o TF-IDF, um dos métodos a ser implementado, o critério `1`. Este método requer o cálculo da frequência de documentos DF, obtida em tempo constante no caso de um índice único mas pode requerer um tempo proporcional ao comprimento da lista de incidências, no caso de uma implementação com índice auxiliar, por conta da verificação dos arquivos caducados. Ignorando-se o fato de que um arquivo do índice principal possa ter caducado ou não, podemos definir o conceito de *quase-DF* como sendo a soma das frequências de documentos DF dos índices principal e auxiliar, e por conseguinte definir o conceito de *quase-TF-IDF* como sendo o valor do TF-IDF computado com a quase-DF no lugar da DF. Assim, pede-se também a implementação de uma variante do critério de ordenação por TF-IDF que empregue o índice *quase-TF-IDF* como critério de ordenação dos resultados, o critério `2`.

> Ademais, o conceito de *quase-DF* é também útil para postergar e reduzir as junções das listas de incidência na produção de listas de termos do vocabulário ordenadas (ou quase) por frequência de documentos. Assim, as opções `-t`, `-r` e `-R` deverão selecionar uma certa quantidade de termos do topo de listas de documentos ordenadas por *quase-DF* ao invés de *DF*.

Ademais, propomos o desenvolvimento e a implementação de um método de ordenação de resultados que não consta no livro, e que usa o índice posicional já desenhado na segunda etapa,[1], ao mesmo tempo em que se ocupa em preparar a exibição de um sumário dinâmico bastante sintético. A ideia é listar primeiramente os arquivos que possuem locuções mais curtas a conter todos os termos da consulta; a chave de ordenação da listagem dos arquivos será o comprimento do intervalo mais curto de posições cujo segmento de tokens consecutivos contém todos os termos procurados. Este é o critério `3`, e também implementaremos uma variante simplificadora, a considerar apenas os dois termos de menor DF, o critério `4`.

Observe que um tal segmento sempre existe, já que o segmento formado por todos os tokens de um arquivo considerado contém todos os termos da consulta por conta do critério de escolha do arquivo. No caso da aplicação da variante simplificadora relativa a tokens `t` e `u` e um documento `d` que contém ambos, podemos tomar também qualquer intervalo cujas

[1] Disponível em http:www.ime.usp.br/∼alair/mac0333/MeuPrimeiroRecInfo20201118.pdf

extremidades são: um elemento da lista de posições de `t` em `d`; um elemento da lista de posições de `u` em `d`. Cada um destes intervalos é quase certamente mais curto que o das posições todas do arquivo e o mais curto deles pode ser facilmente computado em tempo quadrático com um algoritmo tipo força-bruta.

Chamamos de *distância entre* `t` *e* `u` *em* `d` ao comprimento deste menor intervalo de posições cujo segmento de tokens associado contém `t` e `u`.

Ademais, há que se notar a existência de uma solução linear na soma dos comprimentos das listas de posições que pode ser facilmente implementada para computar a distância entre `t` e `u` em `d`. Para computar a intersecção das listas de incidências relativas aos diversos termos da consulta realizadas na primeira fase deste trabalho, foi sugerido o processo que o capítulo 1 descreve como caso particular de algoritmo mais geral chamado *merge algorithm*, que inclui a etapa de combinação de listas ordenadas no algoritmo do *merge-sort*. Promove-se uma forma de varredura simultânea nas duas listas de incidências. Da mesma maneira, podemos facilmente escrever um novo tipo de *merge algorithm* que varre numa posição `p` a lista de posições de `t` enquanto que avança a varredura na lista de posições de `u` de forma a identificar a posição `q` mais próxima da posição `p`. Varrendo `p` até o fim da lista de posições de `t` e calculando sempre a menor diferença `|p-q|` entre a posição `q` da lista de posições de `u` e a posição `p`, este processo computa a distância entre `t` e `u` em `d`, o que permite a ordenação segundo o critério `4`. Não é difícil generalizar esta varredura para uma simultânea que faça progredir paulatinamente cada um dos possíveis mínimos de um intervalo de posições minimal cujo segmento de tokens associado contenha todos os termos da consulta, o que possibilita uma ordenação pelo critério 3.

A opção de linha de comando dada por `-o n` para o utilitário `mirs` seleciona o critério de ordenação `n`. Para o critério `n` diferente de 0, ela também imprime, entre o identificador e o nome relativo do arquivo, uma coluna com o valor associado ao critério.

A opção `-a n` especifica o número máximo `n` de arquivos a serem listados segundo o critério de ordenação especificado (o critério *default* é o `0`). A opção `-v` complementa a listagem dos arquivos e acrescenta um resumo dinâmico de uma região de menor tamanho contendo os tokens que participam na avaliação do critério de ordenação 3 ou 4. São acrescidos três tokens de contexto em cada extremidade.

A página https://www.ime.usp.br/ alair/mac0333/shell3.html contém uns exemplos de execução de nossa solução para a terceira etapa do sistema MIR. São realizadas diversas formas de busca das palavras `ime`, `prêmio`, `internacional`. Observe que esta implementação posterga a leitura dos índices posicionais, como se pode notar nas mensagens em cinza existentes apenas nos critérios de ordenação `3` e `4`. Não se cobrará isto nesta disciplina. Ainda que postergada ao máximo, nossa implementação carrega integralmente a lista geral de posições, o que pode ser evitado com um esquema de mapeamento de memória como o provido por `np.memmap`.