Findpat2

Relatório de Desenvolvimento

Autores: Alberto Trindade Tavares (att) e Bruna Carolina Baudel de Santana (bcbs)

1. Introdução

Findpat2 é uma ferramenta desenvolvida em C++ com o objetivo de realizar buscas offline de padrões em arquivos de texto, contrastando com o nosso projeto anterior, o findpat, onde a busca executada era sempre online, não necessitando de indexação e armazenamento do texto.

O programa desenvolvido neste projeto suporta dois modos de execução: indexação e busca. No modo de indexação, a ferramenta indexa um arquivo de texto especificado pelo usuário, armazenando o índice gerado de forma comprimida. No modo de busca, a ferramenta realiza uma busca de um ou mais padrões em um arquivo de texto indexado no modo de indexação, realizando, primeiramente, uma descompressão do índice. A saída padrão do programa neste modo são as linhas do texto indexado onde ocorrem pelo menos um dos padrões especificados.

2. Descrição de Funcionamento

2.1 Interface e leitura de parâmetros

Findpat2 oferece uma interface em linha de comando, seguindo as diretrizes GNU/POSIX, que é responsável por ler as entradas do usuário (através do uso do comando *./findpat2 mode [options] param1 … paramn*) e exibir a saída correspondente.

Para realizar o processamento dos parâmetros especificados pelo usuário, assim como no projeto anterior, nós utilizamos o Lean Mean C++ Option Parser (http://optionparser.sourceforge.net/), uma pequena biblioteca. Da mesma forma que fizemos com o findpat, nós adicionamos um único arquivo .h do projeto à nossa ferramenta, para poder usar essa biblioteca. Este arquivo está no diretório /libs, destinado a bibliotecas externas.

Para cada modo de execução, há um diferente conjunto de opções e parâmetros a serem passados pelo usuário. O modo de execução é especificado no primeiro parâmetro como ‘index’ ou ‘search’. A seguir, veremos as opções e parâmetros disponíveis para cada modo.

2.1.1 Modo de indexação: parâmetros e opções

No modo de indexação, o comando a ser dado pelo usuário segue este formato: *./findpat2 index [options] textfile*. A única opção disponível nesse modo é o *help* (-h, --help), que exibe uma mensagem que descreve instruções para o uso do programa.

Como nós decidimos implementar um único algoritmo para a construção do índice e um para a compressão do arquivo, não foi necessário incluir opções adicionais para indicar qual seria utilizado para cada tarefa. O único parâmetro que precisa ser especificado pelo usuário é o *textfile*, que é o nome do arquivo de texto a ser indexado.

2.1.2 Modo de busca: parâmetros e opções

No modo de busca, o comando segue este formato: *./findpat2 search [options] pattern indexfile*. As opções disponíveis são: 1) *help* (-h, --help), com a mesma finalidade do modo anterior; 2) *pattern* (-p, --pattern), que especifica um arquivo de texto com os padrões a serem buscados; e 3) *count* (-c, --count), que em vez de imprimir as linhas do texto indexado em *indexfile* em que os padrões ocorrem, exibe a quantidade total de ocorrências.

Os seguintes parâmetros são especificados nesse modo: *pattern* e *indexfile*. O parâmetro *pattern* especifica um único padrão a ser procurado no texto indexado. Esse parâmetro deve ser sempre especificado, exceto quando a opção *pattern* (-p, --pattern) é utilizada, definindo uma lista de padrões, ao invés de um único padrão. O parâmetro *indexfile* é utilizado para especificar o nome do arquivo de extensão .idx, que deve ter sido gerado pelo programa no modo de busca.

2.2 Modo de indexação

Neste modo, nós implementamos dois algoritmos: um para realizar a indexação do texto especificado pelo usuário e outro para comprimir o índice gerado, incluindo o texto. Para fazer o processamento dos parâmetros do modo de indexação e acionar os algoritmos implementados, nós criamos a classe *IndexProcessor*.

2.2.1 Algoritmo de indexação

Para fazer a indexação do texto, nós implementamos um **array de sufixos**. Antes de tomar a decisão de qual estrutura nós iríamos utilizar, entre o array de sufixos e a árvore de sufixos, nós comparamos o desempenho registrado por Manber e Myers dessas estruturas, considerando textos com 100.000 caracteres. Abaixo, segue um quadro comparativo, retirado do artigo desses autores onde o array de sufixos é apresentado (http://webglimpse.net/pubs/suffix.pdf):

	Space (Bytes/text symbol)					**Construction Time**		**Search Time**	
		S.Trees			*S.Arrays*	*S.Trees*	*S.Arrays*	*S.Trees*	*S.Arrays*
	I/N	Link	Tree	Vector					
Random ($\|\Sigma\|=2$)	.99	23.8	27.8	19.8	5.0	2.6	7.1	6.0	5.8
Random ($\|\Sigma\|=4$)	.62	17.9	20.4	18.9	5.0	3.1	11.7	5.2	5.6
Random ($\|\Sigma\|=8$)	.45	15.2	17.0	20.8	5.0	4.6	11.4	5.8	6.6
Random ($\|\Sigma\|=16$)	.37	13.9	15.4	30.6	5.0	6.9	11.6	9.2	6.8
Random ($\|\Sigma\|=32$)	.31	13.0	14.2	46.2	5.0	10.9	11.7	10.2	7.0
Text ($\|\Sigma\|=96$)	.54	16.6	18.8	220.0	5.0	5.3	28.3	22.4	9.5
Code ($\|\Sigma\|=96$)	.63	18.1	20.6	255.0	5.0	4.2	35.9	29.3	9.0
DNA ($\|\Sigma\|=4$)	.72	19.5	22.4	25.2	5.0	2.9	18.7	14.6	9.2

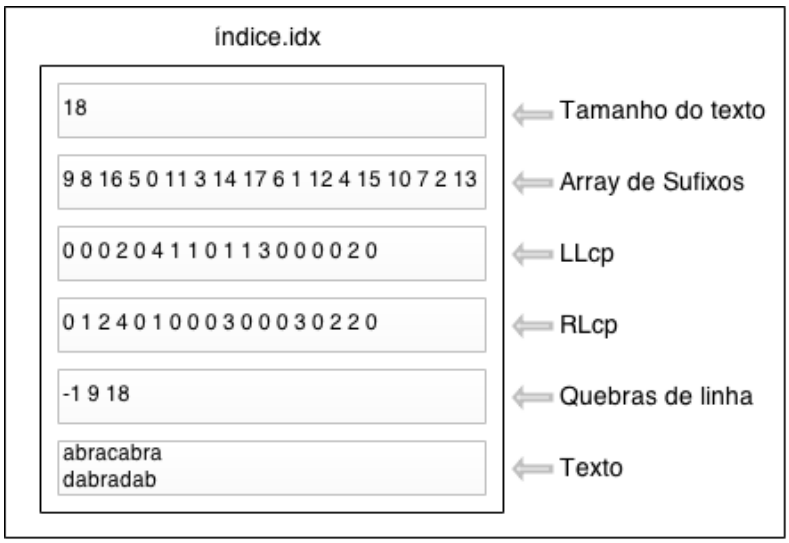
Como podemos ver nessa tabela, o tempo para a construção de uma árvore de sufixos é consideravalmente menor que o da construção do array, porém, na busca em textos, o array de sufixos obtém uma melhor perfomance.

Como a indexação de um texto é algo que deve ser feito esporadicamente e a busca de padrões no índice desse mesmo texto deve ser algo mais frequente, nós decidimos priorizar o tempo de busca, escolhendo, assim, implementar array de sufixos. No entanto, nós tínhamos a intenção de implementar, também, a árvore de sufixos, permitindo ao usuário escolher, através de uma opção na interface de linha de comando, a estrutura desejada, porém, infelizmente, não houve tempo hábil para isso.

No algoritmo de construção do array de sufixos, há uma etapa onde temos a criação da matriz *p*, que representa o ranking de cada substring do texto de tamanho 2^i, onde i é o índice dessa matriz. Ao longo dessa criação, nós temos que realizar a ordenação de triplas, que representam o valor do ranking de um par dessas substrings. Como o intervalo de valores é algo definido, indo de 1 ao tamanho do texto (maior valor possível para o ranking), nós decidimos implementar o **bucket sort** para realizar a ordenação dessas triplas para otimizar essa etapa.

Esse algoritmo não faz comparação entre valores, inserindo as triplas em um array (que representa os “baldes”), obtendo, dessa forma, um desempenho superior a um algoritmo de ordenação por comparação. Enquanto um algoritmo desse tipo, como o *quicksort*, tem um desempenho médio de O(n log n), onde n é o tamanho do texto, o *bucket sort*, nesse contexto, tem uma complexidade de O(n).

Com o objetivo de pré-computar a maior quantidade de informações possível para a busca, além do array de sufixos, armazenado na segunda linha, o nosso arquivo de índice inclui outras informações. A estrutura do índice é apresentada na imagem ao lado.



Na primeira linha, nós temos o tamanho do texto, facilitando o *parsing* das linhas seguintes no momento que o índice é carregado na memória, no modo de busca. Nas terceiras e quarta linhas, nós temos os arrays **LLcp** e **RLcp**, que são usados no algoritmo de casamento de padrão. Eles são computados após a construção do array de sufixos, utilizando a matriz *p* para computar o **lcp** (*longest common prefix*) entre todas as possíveis posições *middle* da busca binária sobre o array e as respectivas posições *left* e *right*.

Outra informação que nós decidimos incluir no índice são os índices de quebra de linha do texto, preenchidos na quinta linha do arquivo do índice. Essa informação é utilizada no modo de busca do programa para, dados os índices de ocorrência, imprir as linhas do texto indexado onde há ocorrências. As linhas restantes do arquivo são dedicadas ao armazenamento do texto.

2.2.2 Algoritmo de compressão

O arquivo do índice gerado pelo programa segue a estrutura ilustrada na imagem acima, no entanto, o seu conteúdo é comprimido, a fim de reduzir o espaço necessário para o armazenamento. Como algoritmo de compressão para realizar essa tarefa, nós escolhemos o **LZ78**. De acordo com vários artigos e fóruns, esse algoritmo é mais eficiente, em termos de tempo de execução, do que o *LZ77*, por esse motivo preferimos implementá-lo. Porém, o *LZ77* tem uma taxa de compressão menor, mas não tão menor quanto a do *LZ78*, o qual em vários testes ficou com taxa de compressão em média de 75%.

Esse algoritmo se baseia na construção de um dicionário dos fatores recorrentes do texto. Na compressão, esse dicionário é representado na nossa implementação como uma árvore, onde cada nó é um *DictionaryTreeNode*, *struct* criada por nós. Cada nó inclui o id de entrada do dicionário e os filhos, representados por um *unordered_map*, que mapeia cada possível letra do texto com o respectivo nó filho.

```
struct DictionaryTreeNode {
  unsigned short id;
  unordered_map<char, DictionaryTreeNode*> children;

  DictionaryTreeNode(unsigned short index):id(index) {}
};
```

Tanto na compressão, quanto na descompressão, o par (id do dicionário, caracter de *mismatch*) é representado pela *struct Codeword*, ilustrada ao lado. Para simplificar e agilizar o processo de descompressão, nós fizemos uma importante modificação no algoritmo do *LZ78*: tornamos os *codewords* de **tamanho fixo**, 3 bytes (2 bytes para o id e 1 byte para o caracter de *mismatch*). Para tornar isso possível, nós tivemos que **limitar o tamanho do dicionário** em 65535, o maior valor para um *unsigned short* , que requer 2 bytes para armazenamento. O resultado da compressão é um array de *CodeWord*, codificando o conteúdo do arquivo de índice descrito acima, que é salvo no arquivo de saída, de extensão .idx, de forma **binária**.

```
struct CodeWord {
  unsigned short id;
  char mismatch;
};
```

2.3 Modo de busca

No modo de busca, nós implementamos mais dois algoritmos: um para a descompressão do arquivo de índice especificado pelo usuário e outro para o casamento de padrão. Para realizar o processamento dos parâmetros nesse modo e executar os algoritmos de descompressão e busca, nós utilizamos a classe *SearchProcessor*.

2.3.1 Algoritmo de descompressão

Na descompressão do conteúdo do arquivo .idx, é utilizado o algoritmo de descompressão do *LZ78*. Na nossa implementação, o dicionário, dessa vez, é representado como um *unordered_map*, que mapeia índices do dicionário aos respectivos fatores do texto.

Nós decidimos utilizar um *map* aqui ao invés da árvore implementada no processo de compressão devido ao *look-up* invertido. Enquanto na compressão, nós estamos buscando o índice do dicionário para uma dada *string*, onde a travessia pela árvore permite um *look-up* imediato, na descompressão, nós buscamos fatores de textos através dos respectivos ids de entrada. Nesse contexto, o uso de um *map* permite uma busca mais rápida.

Como o dicionário reconstruído na descompressão deve ser o mesmo construído na compressão, nós utilizamos aqui o mesmo limite de 65535 sobre o tamanho do dicionário, o que permite uma leitura rápida dos binários do arquivo de entrada, convertendo-os em um array de *CodeWord*. Com o uso desse array, o dicionário é reconstruído e o conteúdo original do arquivo do índice é recuperado. Cada uma das informações representadas nesse arquivo é obtida: o tamanho do texto, array de sufixos, LLcp, RLcp, índices de quebra de linha e o texto.

2.3.2 Algoritmo de casamento de padrão

Com as informações recuperadas na etapa anterior, a única tarefa restante é a busca dos padrões no texto indexado, realizando para isso uma busca binária no array de sufixos. Para otimizar essa tarefa, nós utilizamos os arrays pré-computados LLcp e RLcp. O resultado final da busca binária é o índice do Lp, que permite a recuperação de todos os índices do texto onde há uma ocorrência de cada padrão.

A partir dos índices de ocorrência, obtidos da busca binária, e dos índices de quebra de linha do texto, obtidos a partir do arquivo de índice descomprimido, são impressas as linhas do texto onde há ocorrência. Porém, essas impressões acontecem somente se a opção *count* (-c, --count) não estiver setada. Caso contrário, é impressa somente a quantidade de índices de ocorrência, representando a quantidade total de ocorrências.

3. Recursos extras

Nós implementamos a opção *count* (-c, --count), no modo de busca, que não foi exigida na especificação do projeto. Essa *flag* foi utilizada no primeiro projeto e se mostrou bastante útil para testar o resultado da busca. Dessa forma, nós decidimos implementar essa opção também no segundo projeto.

4. Bugs conhecidos e limites de desempenho

Para avaliar a corretude da implementação dos algoritmos, nós realizamos diversos testes, conseguindo, dessa forma, identificar alguns bugs e corrigí-los. Atualmente, a ferramenta possui nenhum bug conhecido.

Nós também realizamos testes para medir desempenho e concluímos que a ferramenta apresenta uma boa performance na descompressão do índice e busca de padrões. No entanto, o tempo para a construção e compressão do índice está acima do esperado, levando cerca de 3 minutos para gerar o arquivo de índice comprimido de um texto de 7MB, por exemplo. No que diz respeito à taxa de compressão do índice, ela chega, em média, a 75% do original, uma taxa bem aceitável.