Findpat

Relatório de Desenvolvimento

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

1. Introdução

Findpat é uma ferramenta desenvolvida em C++ para buscar um ou mais padrões em arquivos de texto. A ferramenta dá suporte tanto para busca exata de padrões, quanto a busca aproximada. O alfabeto considerado para os padrões e textos são os 256 símbolos do ASCII estendido.

2. Descrição de Funcionamento

2.1 Interface e leitura de parâmetros

Findpat 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 *./findpat [options] pattern textfile [textfiles..]*) e exibir a saída correspondente.

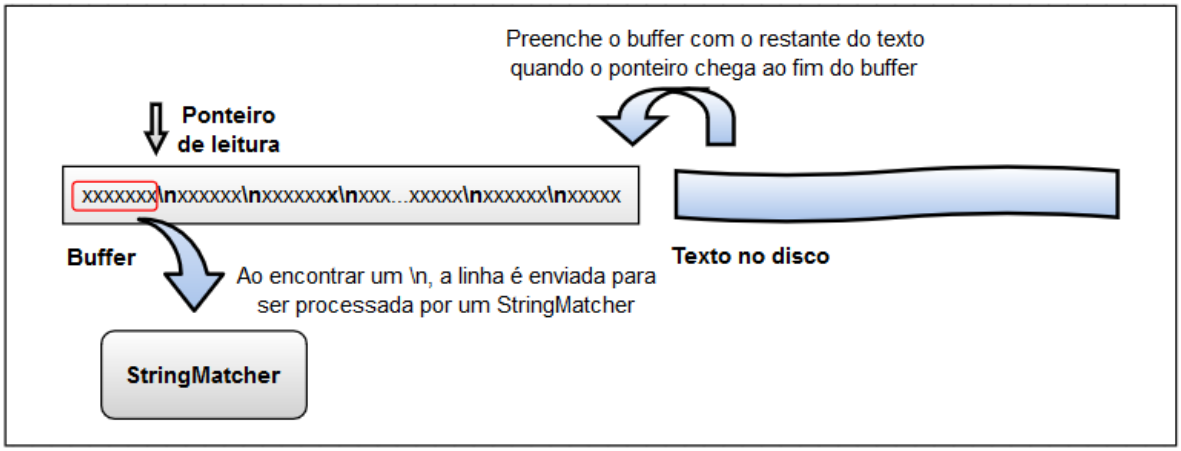
O processamento dos parâmetros passados pelo usuário é feito por intemédio de uma pequena biblioteca externa, o Lean Mean C++ Option Parser (http://optionparser.sourceforge.net/), que se mostrou mais prático que o getopt. Para poder usar essa biblioteca no desenvolvimento do findpat, nós precisamos adicionar um único arquivo .h do projeto à nossa ferramenta. Nós colocamos esse arquivo no diretório /libs, destinado a bibliotecas externas.

Além da leitura dos parâmetros opcionais e obrigatórios, nós implementamos validações para garantir, por exemplo, que a distância de edição máxima setada é um valor positivo.

2.2 Leitura de arquivos de texto

O usuário pode especificar arquivos em dois contextos: 1) para especificar os padrões a serem buscados, através da opção *-p*, *--patternfile*; ou 2) para os arquivos de textos (onde os padrões serão procurados), através do parâmetro *textfile*.

A leitura do arquivo com padrões (*patternfile*) - quando especificado - é feita utilizando o *std::getline*, já que esse arquivo geralmente é pequeno. Por outro lado, os arquivos de textos (*textfiles*) podem ser gigantes, com *gigabytes* de tamanho, podendo superar inclusive a memória RAM disponível. Usar o *std::getline* nesse caso é uma má escolha, pois ele oferece uma taxa de leitura de **~2MB/s**. Dessa forma, nós implementamos uma estratégia de leitura diferenciada para esses arquivos, ilustrada de forma simplificada na imagem abaixo.



Essa estratégia de leitura consiste no uso de um buffer, um array de chars de tamanho parametrizável, atualmente configurado como 50.000. Inicialmente, o buffer é completamente preenchido com os primeiros 50.000 caracteres do texto. Se o texto for menor que o tamanho do buffer, ele já vai ser inteiramente carregado em uma só vez.

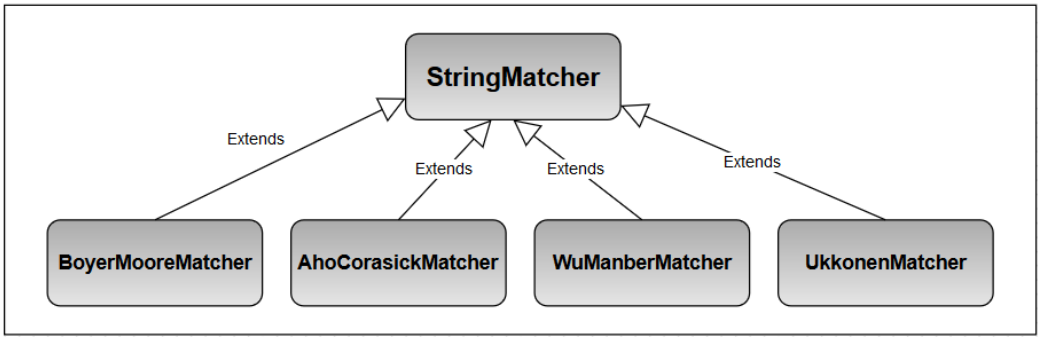
Após esse preenchimento inicial do buffer, é iniciada a leitura de cada letra, usando ponteiros pra indicar o caractere atual do buffer e outras posições relevantes. Ao encontrar um '\n', a linha atual é enviada para um objeto do tipo *StringMatcher*, classe criada para representar os algoritmos responsáveis pelo *string matching*. A estrutura de *StringMatcher* será detalhada a seguir. Esse objeto irá retornar, para cada linha, a quantidade de ocorrências encontradas; caso esse valor seja maior que zero e a opção -c não estiver setada, a linha é impressa na tela.

Quando o ponteiro de leitura do buffer chega ao fim, o buffer é preenchido com os próximos 50.000 caracteres ou o restante do texto. Essa estratégia proporciona um desempenho muito superior ao uso do *std::getline*, com uma taxa de leitura de **~100MB/s**, independente do tamanho do arquivo a ser lido.

2.2 Algoritmos de string matching e heurística para combinação

Para encontrar ocorrências de um ou mais padrões em arquivos de texto especificados pelo usuário, nós implementamos quatro diferentes algoritmos, que foram vistos em sala de aula: 1) **Boyer-Moore**, 2) **Aho-Corasick**, 3) **Wu-Manber** e 4) **Ukkonen**.

Os dois primeiros são usados para a busca exata de padrões, enquanto os dois últimos para a busca aproximada. Para cada tipo de busca, um ou os dois algoritmos de forma combinada são empregados de acordo com alguns critérios que serão descritos na próxima página.



A implementação de cada um desses algoritmos é encapsulada em classes que estendem *StringMatcher*. Acima temos um diagrama que mostra essas classes que herdam de *StringMatcher*.

StringMatcher possui os métodos mostrados ao lado, onde temos três métodos virtuais que são implementados pelas subclasses. Cada uma dessas subclasses possui um ou mais padrões (no caso do *AhoCorasickMatcher*) como atributos e implementam o método *findMatches* de tal forma que ele retorna a quantidade de ocorrências (exatas ou aproximadas) dos padrões no texto que vem como parâmetro.

```
public:
  StringMatcher();
  virtual ~StringMatcher(){};
  virtual int findMatches(string& text)=0;
  virtual void preProcessPatterns()=0;
};
```

O *preProcessPatterns* é chamado no construtor de cada subclasse, sendo responsável por criar estruturas pré-processadas de acordo com o padrão, como as tabelas *badChar* e *goodSuffix* no *BoyerMooreMatcher*.

Dadas as entradas do usuário, um ou mais de um desses tipos de StringMatcher são utilizados para buscar ocorrências exatas ou aproximadas dos padrões especificados nos arquivos de texto. Abaixo temos as situações em que cada um dos algoritmos é aplicado:

1. **Busca exata (opção -e não setada ou definida como zero) com até cem padrões**: quando o usuário especifica uma opção para o parâmetro *pattern* ou um *patternfile* com até cem padrões, o algoritmo

utilizado para procurar ocorrências desses padrões nos textfiles é o BoyerMoore, implementado pelo ***BoyerMooreMatcher***. Um objeto de *BoyerMooreMatcher* é criado para cada padrão.

2. **Busca exata com mais de cem padrões**: quando temos mais de cem padrões, nós utilizamos o Aho-Corasick, que é responsável por procurar ocorrências de um conjunto de padrões, sendo criado, dessa forma, somente um objeto do tipo ***AhoCorasickMatcher*** para todos os padrões. Esse limite de 100 para o uso de BoyerMoore foi definido de forma empírica. Por meio de diversos testes, nós concluímos que para uma quantidade não muito grande de padrões, a complexidade do pré-processamento do Aho-Corasick não compensa o seu uso, sendo mais rápida a aplicação do Boyer-Moore repetidas vezes (uma vez para cada padrão). Por outro lado, para uma quantidade grande de padrões, o Aho-Corasick é mais robusto, sendo muito mais rápido que o Boyer-Moore. Por exemplo, enquanto com 50 padrões, o Boyer-Moore retorna o resultado em 2 segundos, o Aho-Corasick retorna em 3 segundos, mas com 500 padrões, o Boyer-Moore leva 17 segundos, enquanto o Aho-Corasick somente 5.

3. **Busca aproximada (opção -e setada com um valor maior que zero) com um padrão de até 64 caracteres**: entre os padrões especificados pelo usuário, para cada um deles que tem até 64 letras, é usado o Wu-Manber. Esse algoritmo foi escolhido por ele ser o mais eficiente entre todos os ensinados em aula, por fazer uso de paralelismo binário. No entanto, devido a limitações técnicas do C++ e de uma arquitetura de 32 ou 64 bits, nós usamos esse algoritmo somente para palavras com até 64 letras. Um objeto do tipo ***WuManberMatcher*** é criado para cada padrão com no máximo 64 caracteres.

4. **Busca aproximada com um padrão de mais de 64 caracteres**: entre os padrões especificados pelo usuário, para cada um deles que tem mais de 64 letras, é usado o Ukkonen. Esse algoritmo foi usado ao Sellers, por ser mais eficiente e ele também encontra ocorrências aproximadas de padrões de qualquer tamanho. Um objeto do tipo ***UkkonenMatcher*** é criado para cada padrão maior que 64 letras.

2.3 Algoritmos de busca exata

Seguem detalhes de implementação dos algoritmos de busca exata utilizados nesse projeto.

2.3.1 Boyer-Moore

O algoritmo Boyer-Moore utiliza duas tabelas (mau caractere e bom sufixo), que representam saltos seguros de seções do texto. Sua computação é dada pela comparação do conteúdo da janela do padrão da direita para a esquerda, o que resulta em uma menor quantidade de comparações, aumentando sua eficiência, visto que alguns caracteres do texto serão ignorados durante a execução. Esses saltos são pré-processados uma única vez para cada padrão. O padrão acompanha o texto da esquerda para a direita, ou seja, se deslocando sempre para frente. A ferramenta *grep* utiliza esse algoritmo de maneira otimizada. A principal vantagem do algoritmo é não precisar checar cada caractere do texto, podendo ter eficiência sublinear na prática.

As tabelas foram implementadas utilizando arrays de *int*. O array *badChar* guarda a última posição de cada caractere do padrão na ordem dos caracteres do alfabeto. O *goodSuffix* armazena os saltos seguros, valores inteiros, e representa o bom sufixo que também é prefixo do padrão para posição de *mismatch*. Os saltos são calculados a partir dos arrays *border* e *reversedBorder*, que correspondem a borda normal e a borda reversa do padrão, respectivamente.

2.3.2 Aho-Corasick

Aho-Corasick é um algoritmo que encontra ocorrências de um ou mais padrões passando pelo texto uma só vez. O algoritmo se baseia na construção de uma máquina de estados finitas (FSM), em que cada estado representa um ponto da leitura do texto. No pré-processamento dos padrões (executado em *preProcessPattern*), esse FSM é construído uma única vez, e em cada vez que o *findMatches*, o FSM é usado para ir percorrendo de estado para estado enquanto o texto é lido. Em cada estado de aceitação passado, as ocorrências associadas ao estado são computadas.

Para representar esse FSM, nós criamos a *struct AC_FSM*, cuja estrutura é ilustrada ao lado. Esse FSM possui três *maps*: *goTo*, *fail* e *occurrences*. O *goTo* mapeia um número que identifica um estado e uma letra para o próximo estado. Para fazer isso de forma eficiente, o respectivo *map* mapeia um inteiro para um array de 256 inteiros, onde cada posição desse array se refere a uma letra do alfabeto. Então, por exemplo, para saber o próximo estado a partir do estado 3 e a letra 'b', se acessa *fsm->goTo[3][(int) 'b']*.

```
struct AC_FSM {
  map<int, int*> goTo;
  map<int, int> fail;
  map<int, set<int>> occurrences;
};
```

O *map fail* mapeia um estado para outro, que indica onde se deve ir quando não há transições para um próximo estado a partir do estado atual e a letra corrente do texto. Por último, o *map occurrences* mapeia um estado para um conjunto de ocorrências de padrões associados ao estado.

2.4 Algoritmos de busca aproximada

Seguem detalhes de implementação dos algoritmos de busca aproximada utilizados nesse projeto.

2.4.1 Wu-Manber

Sendo chamado também de *Shift-And*, esse algoritmo define máscaras binárias e faz a manipulação de bits de acordo com a distância de edição e a tabela de bits do alfabeto. Esse algoritmo é utilizado na ferramenta *agrep*, porém ocorrências intercaladas são ignoradas pelo mesmo. Um dos motivos para a escolha desse algoritmo, além de ser utilizado por uma ferramenta de referência, é que ele é bastante eficiente para padrões de até 64 caracteres, em relação a outros algoritmos vistos em sala de aula.

Para a implementação das tabelas de bits foi utilizado arrays de *unsigned long long*. Esse é um tipo de inteiro sem sinal que armazena valores em 64 bits. Tentamos utilizar o tipo *bitset*, que emula um array de elementos booleanos, mas como não é permitido criar *bitset* com tamanho dinâmico, descartamos essa possibilidade. O array *charMask* guarda a máscara binária em relação a cada caractere do alfabeto de maneira espelhada. O *bitMask* representa a máscara de bits que contém os prefixos do padrão os quais correspondem a qualquer sufixo do texto com a quantidade de erros menor ou igual a distância de edição. Foi necessário a criação do *unsigned long long maskFilter* para descartar 64 - m primeiros bits do *bitMask*, ao verificar o bit mais significativo.

2.4.2 Ukkonen

O algoritmo Ukkonen se baseia na construção de uma FSM, assim como o Aho-Corasick, para simular a passagem pela tabela de programação dinâmica que computa a distância de edição entre um texto e um padrão. Dessa forma, o algoritmo é capaz de encontrar ocorrências aproximadas de um padrão em um texto ao percorrer a FSM.

Para representar essa FSM, foi criado o *struct Ukk_FSM*, ilustrado ao lado, que possui o *map delta* e o *set acceptStates*. O *delta* é muito semelhante ao *goTo* do *AC_FSM*, mapeando um estado, que é representado por um número, e uma letra para o próximo estado, enquanto *acceptStates* é o conjunto de estados de aceitação.

```
struct Ukk_FSM {
  map<int, int*> delta;
  set<int> acceptStates;
};
```

3. Bugs conhecidos e limites de desempenho

Para avaliar a corretude da implementação dos algoritmos, nós realizamos diversos testes, envolvendo desde pequenos arquivos a até arquivos de 10 GB, além de trocar informações na lista da disciplina. Dessa forma, nós conseguimos identificar alguns bugs e corrigi-los. Atualmente, a ferramenta possui nenhum bug conhecido.

Nós também realizamos testes para medir desempenho e concluímos que a ferramenta apresenta um bom desempenho para a maioria dos casos. Um caso em que o programa dá a saída de forma mais lenta que o esperado é quando o Ukkonen é usado, quando o usuário procura ocorrências aproximadas de padrões com mais de 64 letras.