Universidade Federal de Pernambuco - UFPE
CIn - Centro de Informática

Processamento de Cadeias de Caracteres(if767)
Projeto 1

Antonio José Gadelha de Albuquerque Neto(ajgan)
Gabriel Vinícius Melo Gonçalves da Silva(gvmgs)

Recife, 2018.2

1. Identificação

A equipe é formada pelos alunos Antonio Gadelha(ajgan) e Gabriel Melo(gvmgs). Antonio implementou os algoritmos Brute Force, KMP e Sellers e fez também a parte da leitura das flags da linha de comando. Gabriel implementou os algoritmos Aho-Corasick e o Ukkonen. Ambos contribuíram com a documentação e com a realização dos testes.

2. Implementação

A ferramenta foi toda implementada em c++, assim como havia sido sugerido. O funcionamento do programa obedece a entradas por linha de comando no seguinte formato: “pmt [options] pattern textfile”. Caso não seja especificado, a ferramenta segue uma heurística para escolher qual o melhor algoritmo a ser utilizado. As opções de comando e a heurística de escolha de algoritmo serão explicadas posteriormente nesse relatório.

2.1 Algoritmos Implementados

2.1.1 Algoritmos de Busca Exata

- **Brute Force**

  O algoritmo brute force foi implementado apenas por efeito comparativo, pois a estratégia dele é a mais simples possível e o desempenho acaba não sendo tão bom. Para checar se há um match, o algoritmo percorre todo o comprimento do texto e checa se a posição dá match com o primeiro caractere do padrão. Caso ocorra match, checa se a próxima posição dá match e assim por diante até que todo o padrão dê match. Caso ocorra mismatch em alguma posição do padrão, o algoritmo segue para a próxima posição do texto. O algoritmo pode ser implementado facilmente com dois laços de repetição.

- **Knuth-Morris-Pratt (KMP)**

  O algoritmo de Knuth-Morris-Pratt, o KMP, é uma melhoria em cima do algoritmo brute force. O KMP considera que o avanço de posições no texto pode ser mais efetivo se você considerar os

prefixos e sufixos do padrã. Inicialmente, o algoritmo cria um array do tamanho do padrão para guardar em cada posição o tamanho do maior prefixo que também é sufixo da substring atual do padrão. Esse array otimiza os deslocamentos do padrão no texto, pois entende-se que não há necessidade de checar se há um match em um caractere, se outro caractere igual, mas em outra posição já deu match. Para implementar o algoritmo, basta criar uma função que nos dê o array com os valores de prefixo que também são sufixos na substring do padrão e usar os valores desse array para controlar nossos deslocamentos dentro do laço de repetição.

- **Aho-Corasick**

  O algoritmo Aho-Corasick se baseia na criação de de um autômato finito, para o reconhecimento de variados padrões em um dado texto, de forma que cada estado de aceitação representa a ocorrência de um dos padrões designados. Para tanto o algoritmo constrói 3 funções para o autômato: função de transição (g), função de falha, e função de saída. O autômato é gerado a partir dos padrões informados e por um alfabeto, nesse caso os caracteres ASCI, os 126 primeiros.

  A função de transição indica qual o próximo estado baseado no caractere lido, o função de falha fornece um estado alternativo para o caso de ser lido um caractere inválido para o estado atual e a função de saída indica quando um padrão ocorreu.

  É armazenado em uma lista para cada padrão a posição das suas ocorrências.

2.1.2 Algoritmos de Busca Aproximada

- **Sellers**

  O algoritmo Sellers cria uma matriz de tamanho (m + 1) x (n + 1), sendo m o tamanho do padrão e n o tamanho do texto. Nessa matriz calcula-se para cada posição do texto, a diferença entre o caractere do texto e do padrão. O valor de cada célula da matriz segue uma regra que diz que ele será o valor mínimo entre 3 valores:

  1. O valor da célula da esquerda + 1(ex: se o padrão na sua posição anterior dá match com o texto, então na posição atual o erro vai ser igual a 1)
  2. O valor da célula de cima + 1 (ex: se o padrão dá um match com a posição anterior do texto, o erro na posição atual é de 1)

3. O valor da célula a noroeste + a diferença entre o caractere do padrão com o do texto (ex: se na posição anterior de padrão e texto houve um match, o erro nessa posição será o erro medido da diferença entre as posições atuais)

Com a matriz montada, é verificado para cada linha(que representa cada posição do texto) o valor do erro da sua última coluna(que representa a posição final do padrão). Se no último caractere do padrão o valor do erro na matriz for menor que o erro permitido na entrada, há um match. Caso o erro seja maior que o desejado, há um mismatch. Para implementar esse algoritmo pode-se seguir uma estratégia de se reutilizar dois arrays que representariam a linha atual da matriz e a linha anterior, já que os valores de cada linha dependem exclusivamente da linha anterior e da própria linha. Com essa estratégia, apenas dois arrays de tamanho m+1 são utilizados, ao invés de usar n+1, o que é muito benéfico pro custo de espaço do algoritmo. Com dois laços de repetição dá para implementar o algoritmo sem grandes problemas.

- **Ukkonen**

2.2 Detalhes de Implementação

2.2.1 Estruturas de Dados

Além do uso amplo das estruturas primitivas, foram utilizadas com certa recorrência a estrutura <vector> para melhor manipulação dos dados. De forma menos recorrente (apenas uma vez, para simplificar a escrita do código) foram usadas as estruturas <tuple> e <set> no algoritmo Ukkonen, nele também foi feito uso de um modelo de árvore ternária.

2.2.2 Estratégia de Leitura das Entradas

A leitura de entradas foi dividida em duas partes, na primeira parte da leitura varre-se o argv(todos os argumentos passados para o programa advindos do comando de linha) e procura por strings que sejam iguais às flags esperadas. As flags podem ser:

- *-c ou --count*:

  Flag para exibir contagem de ocorrências do padrão no texto(por default, não se exibe)
- *-e ou --edit emax*:

Flag para explicitar o valor de edit máximo em casos de algoritmos de busca aproximada(por default, é zero)

- *-a ou --algorithm algorithmname*:

  Flag para explicitar qual algoritmo de busca o usuário deseja utilizar(por default, a escolha do algoritmo segue uma heurística que será explicada na próxima seção)

  lista de algoritmos:

  - *brt* (Brute Force)
  - *kmp* (KMP)
  - *aho* (Aho-Corasick)
  - *sel* (Sellers)
  - *ukk* (Ukkonen)

- *-p ou --pattern patternfile*:

  Flag para indicar que os padrões devem vir de um arquivo(por default, o padrão é um só e é digitado pelo usuário)

- *-h ou --help*:

  Flag que exibe informações básicas do programa ao usuário.

- *-t ou --time*:

  Flag que exibe ao final o tempo gasto com a execução do programa(por default, não se exibe)

A flag *-t* foi uma opção que a equipe julgou como interessante, para facilitar a medida do tempo, mas que não estava no escopo das especificações do projeto.

Após o programa identificar as flags dentre os argv, varre-se todos os argvs novamente para se identificar os argumentos que não foram identificados no passo anterior, os dois argumentos achados nesse passo serão o padrão e o arquivo de texto, respectivamente. Caso o usuário tenha fornecido a flag *-p*, apenas um argv terá sobrado para esse segundo passo, e ele será o arquivo de texto. Também é considerada a opção de leitura em múltiplos arquivos, para isso o usuário pode fornecer wildcards(ex: livro*.txt).

Caso haja alguma inconsistência no comando, como uma flag não esperada ou se estiver faltando alguma informação necessária, a flag *--help* é disparada para o usuário.

2.2.3 Heurística de Seleção de Algoritmos

A heurística de escolha de algoritmos em caso do usuário não especificá-lo foi determinada a partir de experimentos e se deu da seguinte forma:

Para busca exata:

- Caso existam mais de 20 padrões para serem pesquisados, é selecionado o Aho-Corasick
- Caso seja apenas um padrão de tamanho menor que 5, se usa o Brute Force
- De resto, utiliza-se o KMP

Para busca aproximada:

- Seleciona o Sellers(tivemos problemas com o Ukkonen)

2.3 Limitações e Bugs Conhecidos

Um ponto que vale a pena ser mencionado, mas que não é um bug, é que o pmt dá respostas diferentes que as ferramentas grep e agrep para o contador de ocorrências. Nossa ferramenta imprime as mesmas linhas que o grep e agrep. Mas a contagem se dá por vezes em que o padrão dá match, enquanto as ferramentas do mercado imprimem um contador de linhas em que ele ocorre, ou seja, linhas com múltiplas ocorrências não são contempladas pelo contador do grep/agrep, mas essa funcionalidade existe na nossa ferramenta. Acreditamos que esse deve ser o comportamento esperado e a diferença de resultado para o grep/agrep foi proposital. Caso não fosse o objetivo, conseguiríamos sem grandes mudanças fornecer o mesmo resultado. Ou seja, a diferença foi proposital e não um erro de implementação.

Até o momento da elaboração deste relatório o algoritmo Ukkonen, para encontrar padrões aproximados, não foi completamente terminado, apesar de estar estruturado seguindo o modelo visto em sala e o artigo referente, o algoritmo entra em um loop infinito, as suspeitas para esse comportamento estão em algum erro na parte de alocação de memória para a estrutura de árvore (node) utilizada no mesmo na montagem da árvore de sufixos, ou na função “tree_find” que também faz uso da árvore de sufixos. O código para o algoritmo Ukkonen se encontra no github (https://github.com/ajgan/pcc1), para a validação de sua implementação, ainda que imprecisa. Devido a falta deste algoritmo em tempo hábil não pudemos o considerar para os testes e a definição das heurísticas.

3. Testes e Resultados

3.1 Dados e Ferramentas de Comparação

Os testes foram realizados com um texto em inglês de 200 MB disponível no Pizza&Chili. Também realizamos testes com o texto de shakespeare de 5.5 MB usado pelo professor durante a cadeira e com o texto em inglês de 1024 MB também do Pizza&Chili. Por motivos de simplicidade, preferimos focar os experimento na base de 200 MB, já que o comportamento era similar nas outras bases, mas o tempo de testes ficaria bem maior para o de 1024 GB e talvez ficasse pouco expressivo para o texto de 5.5 MB.

Comparamos os desempenhos dos nossos algoritmos com a ferramenta grep para casamento exato e com a ferramenta agrep para casamento aproximado.

3.2 Ambiente de Testes

Os testes foram realizados em um MacBook Air com sistema operacional MacOS High Sierra 10.13.6, memória de 4 GB 1600 MHz DDR3 e processador Intel Core i5 1,4 GHz.

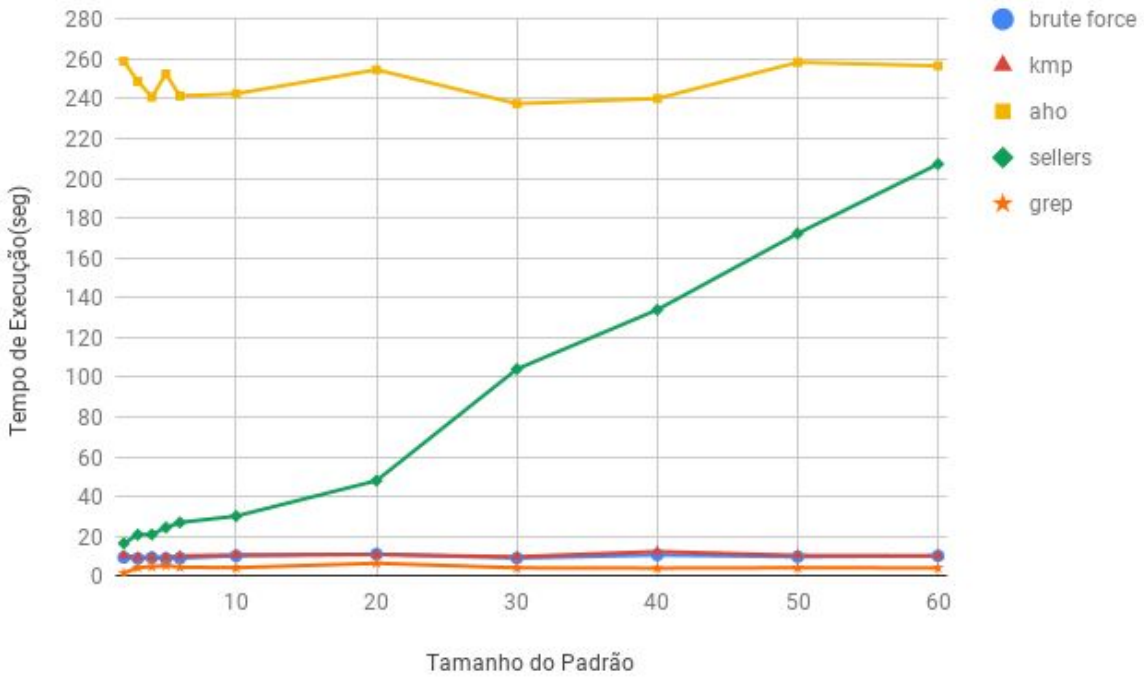
3.3 Experimentos Realizados

Foram feitos dois tipos de teste, um com a entrada de um único padrão(variando tamanho) e outro com um arquivo com várias entradas(variando quantidade). Para os algoritmos de casamento exato, testamos cada um dos algoritmos com os mesmos padrões, com a flag -c ativada para que o tempo de execução não seja atrapalhado pelos prints de linha, e comparamos os desempenhos dos nossos algoritmos com o desempenho da ferramenta grep. A mesma abordagem foi reproduzida para avaliar os algoritmos de casamento aproximado, mas esses foram comparados com o desempenho da ferramenta agrep e também não foram feitos os testes com múltiplos padrões.

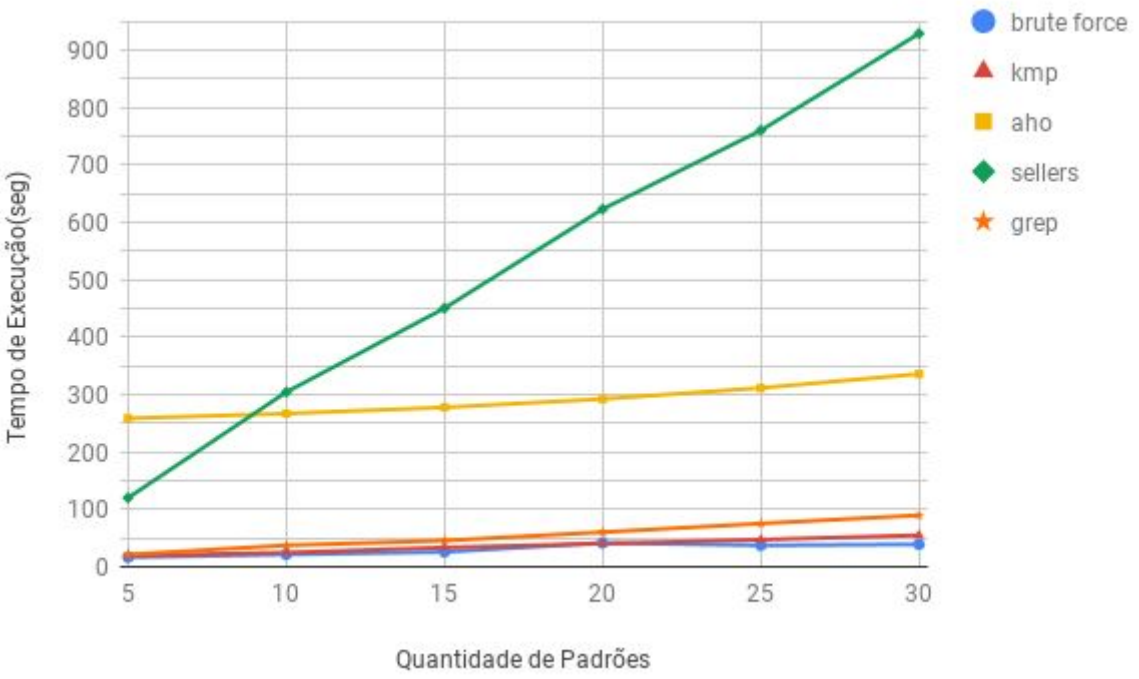
3.4 Resultados Obtidos

Para o primeiro teste foram realizadas buscas de padrões de tamanhos diferentes. O kmp teve os melhores resultados, juntamente

com o brute force. O sellers(com erro zero) se mostrou até razoável para padrões pequenos, mas com o seu crescimento linear, logo percebe-se que não vale a pena utilizá-lo. O aho obteve tempos parecidos para todos os tamanhos, porém muito demorados.



Para o segundo teste usou-se um arquivo de padrões e variou-se a quantidade de padrões inseridos. O sellers(com erro=0) se mostrou inviável. O Aho, apesar de tratar todos os padrões de uma só vez, novamente se mostrou mais lento, mas manteve a mesma média de tempo. Possivelmente com uns 100 padrões, o Aho deve ser melhor. Não tivemos tempo de testar essa possibilidade.



O teste dos algoritmos com busca aproximada foi feito na base de Shakespeare de 5.5 MB pois tivemos problemas com a implementação do segundo algoritmo e ficamos com pouco tempo para testes. De qualquer forma, o Sellers mostrou bons resultados.

3.5 Conclusão

Pudemos ao longo deste projeto aplicar conceitos não apenas, vistos na disciplina de Processamento de Cadeia de Caracteres como também do curso como um todo, o que inclui, desde a pesquisa da melhor forma de implementação dos algoritmos à busca por uma via otimizada que unisse também a compreensão do algoritmo.