Universidade Federal de Pernambuco
Centro de Informática

Relatório Projeto 2
Processamento de cadeias de caracteres

Professor:
Paulo Gustavo Soares Fonseca (paguso)

Sumário

Identificação

Integrantes:
Daniel de Jesus Oliveira (djo)
Rafael Nunes Galdino da Silveira (rngs)
Túlio Paulo Lages da Silva (tpls)

Daniel responsável pela implementação do Array de Sufixos e LZ77
Rafael responsável pela implementação do LZ77 e dos testes
Túlio responsável pela implementação do LZ78
Todos colaboraram em todo projeto seja na forma de implementação ou tentativa de otimização do código. Todos também colaboraram na construção da infraestrutura de leitura, execução, testes e na elaboração relatório.

Execução

Instruções de compilação

Para compilar o projeto, basta executar o comando `make` ou `make ipmt`, e para executá-lo, rodar o executável `impt` que será gerado no diretório `bin`. Para mais informações sobre o uso do programa, a opção `--help` está disponível.

Instruções de execução

Exemplo de uso para indexação:

ipmt index [opções] textfile

Exemplo de uso para busca:

ipmt search [opções] patterns indexfile

Opções:

- -h , --help
  - Mostra uma versão em inglês destas instruções
- -c , --count
  - Faz com que o programa apenas conte as ocorrências de cada padrão, ao invés de imprimir um trecho do texto ao redor de cada uma
- -p , --pattern=<pattern file>
  - Especifica o arquivo de onde serão lidos os padrões a serem buscados (um por linha)
- -x, --compression=<algorithm>
  - Opção extra que configura o tipo de compressão

Se um arquivo com padrões não for especificado, o primeiro argumento após as opções dado ao `impt` será interpretado como o único padrão a ser buscado. Vários arquivos de entrada podem ser especificados.

Implementação

O *ipmt* funciona em dois modos já citados anteriormente, o modo de indexação e o modo de busca. As figuras 1 e 2 abaixo ilustram o fluxo principal de execução da programa.

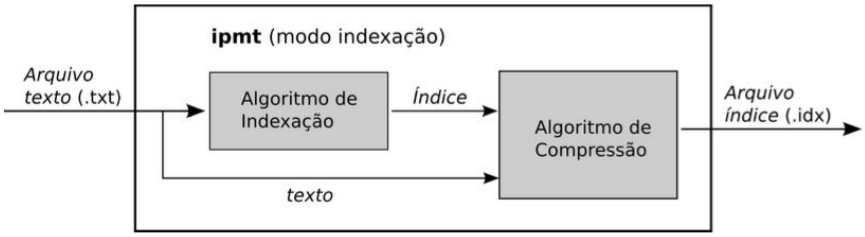


Figura 1: Modo de indexação

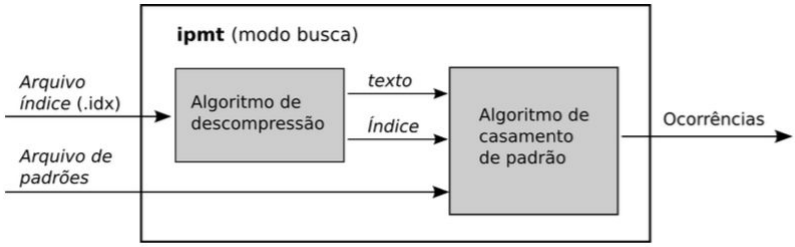


Figura 2: Modo de busca

O algoritmo de indexação escolhido para ser implementado foi o Array de Sufixo, que é array ordenado de todos os sufixos de uma cadeia de caracteres e de uma maneira geral tem um ganho no requisito de espaço quando comparado a Árvore de Sufixo.

Foi implementado para compressão o LZ78, que utiliza um dicionário dinâmico explícito, não uma janela, onde a referência compreende um par composto pelo índice no dicionário e o carácter de mismatch. Isso é uma vantagem com relação ao LZ77, pois a codificação do LZ77 tem um limite dado pelo tamanho da janela que ele utiliza, logo algumas repetições muito espalhadas no código não são reaproveitadas porque podem não se encontrar dentro do tamanho da janela.

Por padrão o algoritmo de compressão utilizado pela programa é o LZ78, mas foi implementado como uma ***opção extra*** o algoritmo de LZ77, acessível por meio da opção de execução *-x=LZ77* ou *--compression=LZ77*.

O algoritmo LZ77 explora as repetições de fatores ao longo do conteúdo a ser comprimido, substituindo tais repetições por referências a ocorrências anteriores de certos trechos de conteúdo. Fazendo uso de uma janela que "desliza" ao longo do conteúdo a ser comprimido, o algoritmo representa o conteúdo comprimido em forma de triplas contendo a posição da repetição na janela, o tamanho da mesma e o conteúdo do byte que a segue no conteúdo descomprimido. O algoritmo é de simples implementação e não requer estruturas de dados complexas, porém pode não oferecer taxas de compressão satisfatórias caso não existam repetições extensas ao longo do texto ou caso elas estejam muito distantes entre si.

Como entrada o programa no modo de indexação deverá receber apenas o texto a ser processado, e ao fim de sua execução criamos o arquivo de índice do arquivo, que contém o texto, o seu array de sufixos e estruturas auxiliares. No modo de busca o programa recebe como entrada o arquivo de padrões (ou o único padrão) a serem buscados e o arquivo de índice (.idx) como argumentos. O programa então imprime, para cada padrão fornecido, uma contagem de ocorrências do padrão no texto, e caso a flag -c não tenha sido fornecida, um pequeno trecho ao redor de cada ocorrência é exibido no console, uma ocorrência por linha.

Decisões de implementação, estruturas utilizadas e limitações

- Devido à forma como a implementação lê e representa os dados lidos, o programa apresenta comportamento indefinido caso as entradas contenham caracteres não representáveis por codificação ASCII;
- Indexação
  - O algoritmo de indexação do texto produz três estruturas: o array de sufixos do texto em questão, e dois arrays de inteiros, Llcp e Rlcp, que contem os prefixos em comum nos índices da busca binária;
  - Durante a construção do array de sufixos, é utilizada uma estrutura auxiliar (um array de triplas representadas por uma struct), e como os índices estão representados por inteiros de 64 bits, a geração do array de sufixos exige um consumo de espaço da ordem de 40n (dos quais 16n fazem parte da saída do algoritmo) bytes, onde n é o tamanho do texto. É também dado como saída um "array de sufixos inverso", usado no próximo passo;
  - Para a construção dos arrays Llcp e Rlcp, É feito o uso de um array com os lcp (largest common prefix, o maior prefixo em comum entre duas strings) entre os sufixos consecutivos na ordem do array de sufixos, gerado no passo anterior. Esse array de lcps é computado em tempo linear, e graças a ele, os arrays Llcp e Rlcp também são. Nesta etapa, nenhuma estrutura auxiliar é alocada, e o consumo de memória é da ordem de 24n;
- LZ77
  - Na nossa implementação reservamos 12 bits para a janela de busca (Ls), que nos dá uma janela de 4095 caracteres e 4 bits para a janela de *lookahead* (Ll), que nos permite detectar repetições de até 16 caracteres;
  - O funcionamento do algoritmo é simples, e a implementação, direta. Nenhuma estrutura de dados especial se fez necessária;
- LZ78
  - Nossa implementação utiliza uma árvore que guarda em cada nó um rótulo e um caractere, com o caminho de um nó para a raiz sendo a palavra representada pelo rótulo. A árvore é construída tanto na codificação quanto na decodificação. Os rótulos e caracteres são transformados em binário, de acordo com um tamanho de rótulo máximo (no nosso caso, 1 << 25 bits) na codificação para seu posterior armazenamento em um arquivo via um buffer. O buffer só é escrito quando fica cheio;

Testes e Resultados

Esta seção apresenta os testes feitos pela equipe. Testamos o desempenho de cada algoritmo implementado, isoladamente, de forma a entender o comportamento deles e poder fazer comparativos. Entretanto também existe a seção onde realizamos testes fim a fim o programa. Os testes foram executados numa máquina com a seguinte configuração:

- MacBook, OS X El Capitan, Intel Core i7 1.7GHz, 4GB 160 MHz DD53.

Para os testes implementamos programas simples para automatizar a execução, eles podem ser encontradas em *ipmt/tests*. E a título de Benchmark, as ferramentas *grep* e *gzip* foram adicionas às execuções em comparativo, tendo a implementação destas vistas como ideais por serem ferramentas já consolidadas, otimizadas e bem testadas. A métrica utilizada para comparação foi o tempo de execução das ferramentas comparadas.

O tempo das execuções foi aferido utilizando as funções *std::chrono::high_resolution_clock* e *durantion* da biblioteca std::chrono:

- http://en.cppreference.com/w/cpp/chrono/high_resolution_clock
- http://en.cppreference.com/w/cpp/chrono/duration

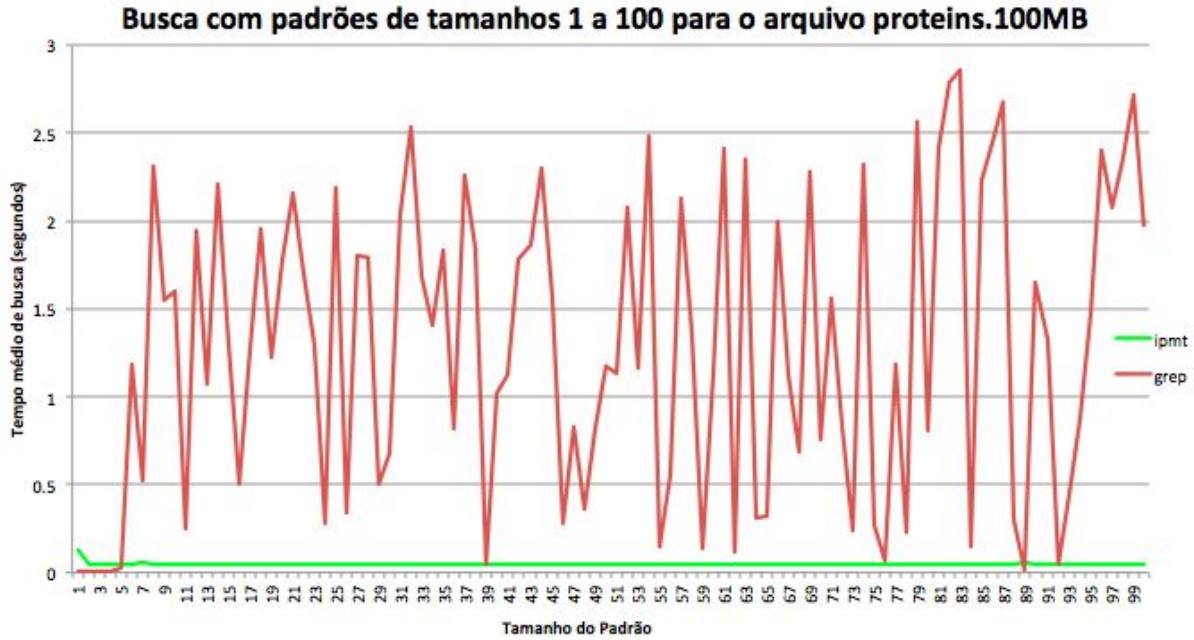
Testes do algoritmo de busca

Os testes então comparam a saída da nossa implementação com a saída esperada (com base no resultado do *grep*) para garantir a corretude e também médio tempo de execução é registrado.

Para reproduzir os testes abaixo ou executar os nossos testes de busca com arquivos e padrões diferentes basta acessar *ipmt/tests*. As instruções de execução encontram-se no arquivo *"searchTests.cpp"*. As bases utilizadas para os testes foram as bases de *proteins* disponíveis em:

- Original: Pizza Chili, Proteins (http://pizzachili.dcc.uchile.cl/texts/protein/)
- Aqui as bases utilizadas nos nossos testes: http://bit.ly/1RqDs8w

Comportamento da busca com padrões de tamanhos 1 a 100 para o arquivo proteins.100MB:



Os padrões (com pelo menos uma ocorrência) a serem buscados foram construídos a partir de substrings do texto e possuem de tamanho 1 até 100. Para isso criou-se um script python (*generatePatterns.py*) que a partir do arquivo de texto e gera os padrões desejados, o script também está disponível em *ipmt/tests*. A discrepância entre os tempos de busca obtidos era esperada, visto que com o índice já construído, a busca passa a ser feita a partir de duas buscas binárias dentro do array de sufixos, apresentando uma complexidade de tempo sublinear.

Comportamento da busca com padrões de tamanhos e ocorrências variados para o arquivo proteins.100MB:

Tamanho(KB)	104857600	**Tempo médio de busca (s)**	
Padrão	***Ocorrências***	*ipmt*	*grep*
GGBABA	0	1.197731	2.353069
CGDED	6	0.00913	0.017441
TLSTK	69	0.002017	0.0033
KAESL	135	0.002786	0.00327
CAVE	316	0.003546	0.00485
GPTG	1964	0.004936	0.006001
SSVV	2914	0.005247	0.004344
PLLA	4611	0.007243	0.005935
FFT	13673	0.004547	0.003264
LLL	142245	0.004544	0.003214
LH	273520	0.009149	0.015476

A tabela acima apresenta a resposta dos programas *ipmt* e *grep* para padrões de tamanhos variados e ocorrências variadas. O teste é para avaliar e tentar entender o comportamento do *ipmt* com padrões que apresentam desde poucas ocorrências à muitas ocorrências e sua evolução (levando em conta que eles possuem tamanhos diferentes, porém não tao distintos). O programa *ipmt* mostrou uma boa performance, principalmente para padrões que possuem poucas ocorrências, ganhando no tempo médio quando comparado ao *grep*. Mostrando mais uma vez que o tempo de busca está satisfatório.

Testes do algoritmo de criação do índice

Para a análise do algoritmo de indexação foram executados testes para medir o tempo médio de criação do do array de sufixo com arquivos de tamanhos variados. Os data sets escolhidos foram os arquivos de tamanhos adaptados para 1MB, 4MB, 9.5MB, 50MB, 100MB. Os arquivos originais e utilizados nos testes estão disponíveis em:

- ❏ Original: Pizza Chili, Proteins (http://pizzachili.dcc.uchile.cl/texts/protein/)
- ❏ As bases dos nossos testes estão em *data*: http://bit.ly/1RqDs8w

Arquivo		Média do tempo de indexaçãodo ipmt
Nome	Tamanho	(segundos)
proteins.1MB	1015886	2.388203
bible.txt	4047392	10.581164
arquivo.txt	9490113	30.316879
proteins.50MB	52428800	222.232224
proteins.100M	104857600	458.095398

Por curiosidade adicionamos benchmarks de diversas implementações de construção do array de sufixo, apresentadas em um comparativo do *libdivsufsort* (biblioteca/API em C para a construção de arrays de sufixo). Acrescentamos a resposta do algoritmo, do programa *ipmt,* de criação do array em sua execução sobre o data set chamado *Canterbury Corpus*, embora a comparação não seja inteiramente válida pelas distintas configurações de máquinas.

- ❏ SACA_Benchmarks disponíveis em: (https://code.google.com/p/libdivsufsort/wiki/SACA_Benchmarks)
- ❏ As bases desses teste estão em *data/cantrbry*: http://bit.ly/1RqDs8w

Canterbury Corpus

Files	Size	ipmt	Archon4r0	BPR	DC	DS	divsufsort1	divsufsort2	KA	KS	MSufSort3	qsufsort	sais
alice29.txt	152089	0.155	0.038	0.044	0.076	0.046	0.038	0.04	0.062	0.058	0.04	0.05	0.046
asyoulik.txt	125179	0.089	0.036	0.044	0.072	0.04	0.04	0.038	0.048	0.056	0.048	0.044	0.04
fields.c	11150	0.01	0.034	0.036	0.042	0.034	0.036	0.038	0.038	0.034	0.038	0.036	0.034
grammar.lsp	3721	0.002	0.038	0.036	0.034	0.034	0.036	0.032	0.036	0.036	0.04	0.036	0.034
lcet10.txt	426754	0.86	0.048	0.07	0.174	0.066	0.06	0.06	0.112	0.122	0.07	0.098	0.074
plrabn12.txt	481861	0.993	0.066	0.082	0.186	0.074	0.056	0.064	0.126	0.144	0.064	0.104	0.078
xargs.1	4227	0.003	0.036	0.038	0.038	0.036	0.034	0.032	0.036	0.036	0.04	0.034	0.036

O ipmt apresentou bons tempos em sua maioria condizentes e equiparáveis com as diferentes implementações mencionadas. Sendo mais rápido por vezes que a própria *libdivsufsort*.

Para reproduzir os testes acima ou executar os nossos testes de indexação com arquivos diferentes basta acessar *ipmt/tests*. As instruções de execução encontram-se no arquivo *"sarrayTests.cpp"*.

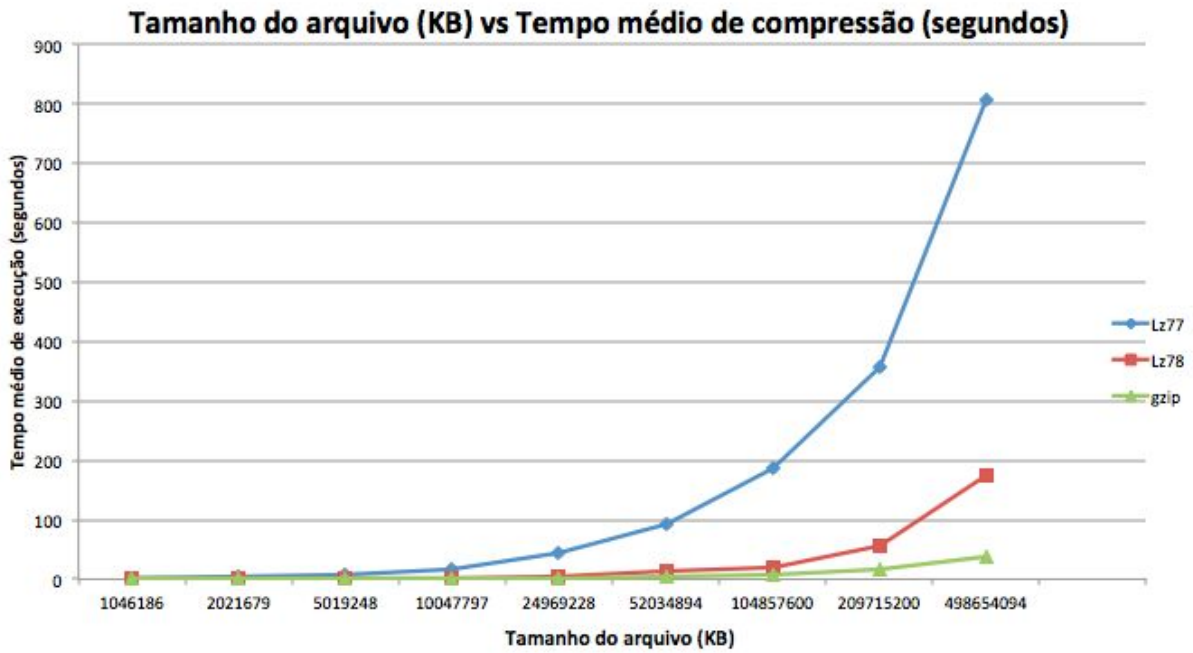
Testes dos algoritmos de compressão

Para analisar o algoritmo de compressão foram utilizadas duas métricas, a primeira verificando a taxa de compressão/descompressão do algoritmo e a segunda métrica utilizada o tempo médio de execução para cada tamanho de base no teste. Executou-se a compressão com as bases *english texts* de tamanhos adaptados para 1MB, 2MB, 5MB, 10MB, 25MB, 50MB, 100MB, 200MB, 500MB, comparando os resultados com o *gzip*. A entrada do algoritmo são arquivos de texto a serem codificados e a saída o código.

Para reproduzir os testes abaixo ou executar testes de compressão com arquivos diferentes basta acessar *ipmt/tests*. As instruções de execução encontram-se no arquivo *"compressionTests.cpp"*. As bases utilizadas para os testes foram as bases de *english texts* disponíveis em:

- Original: Pizza Chili, English texts (http://pizzachili.dcc.uchile.cl/texts/nlang/)
- Aqui as bases utilizadas nos nossos testes: http://bit.ly/1RqDs8w

Teste do tempo médio de compressão



		Tempo médio de execução (segundos)		
Arquivo	**Tamanho original (KB)**	**Lz77**	**Lz78**	**gzip**
english.1MB	1046186	1.672168	0.083681	0.074358
english.2MB	2021679	3.322345	0.199381	0.1496
english.5MB	5019248	8.355696	0.723407	0.363696
english.10MB	10047797	16.836866	1.805937	0.776991
english.25MB	24969228	44.300453	5.575063	1.901204
english.50MB	52034894	90.976456	12.514235	4.248525
english.100MB	104857600	187.589966	20.644581	8.629453
english.200MB	209715200	357.889404	55.554287	15.810278
english.500MB	498654094	805.190613	175.896851	39.15081

O desempenho do Lz77 mostrou-se pouco competitivo, pois desde o início, concluímos que é necessário reavaliar sua implementação e pensar em um preprocessamento e alguma heurística relacionada, pois o algoritmo puro não é suficientemente competitivo. O Lz78 comportou-se bem no entanto, mostrando tempos bons e competitivos comparando com o *gzip* , seu desempenho apenas começa a perder competitividade com arquivos de tamanho próximo a 500MB.

Teste da taxa de compressão (tamanho do arquivo após a compressão)

		Tamanho relativo ao tamanho orignal		
Arquivo	**Tamanho original (KB)**	**Lz77**	**Lz78**	**gzip**
english.1MB	1046186	57.19%	67.32%	67.32%
english.2MB	2021679	58.55%	65.41%	37.85%
english.5MB	5019248	59.16%	59.74%	38.13%
english.10MB	10047797	59.15%	55.04%	37.99%
english.25MB	24969228	58.90%	50.77%	37.70%
english.50MB	52034894	59.22%	48.11%	37.80%
english.100MB	104857600	59.27%	45.85%	45.85%
english.200MB	209715200	59.28%	43.39%	37.87%
english.500MB	498654094	59.56%	40.76%	37.98%

O tamanho relativo apresentado significa que o arquivo ficou com a dada porcentagem do tamanho do arquivo original após a compressão. Tendo isso, o Lz77 mostrou-se bastante consistência, apresentando uma compressão sempre muito próxima, resultando em um arquivo de tamanho em média de 58,92% do tamanho do arquivo original. Já o Lz78 apresentou teve um desempenho bom e com tendência a melhorar a cada arquivo de tamanho maior, tal fato deve-se ao uso do dicionário dinâmico que reaproveita palavras codificadas mesmo que distantes (vantagem com relação ao Lz77 que possui o limite dado pela janela deslizante) .

Teste de tempo médio de descompressão

		Tempo médio de descompressão (segundos)		
Arquivo	**Tamanho original (KB)**	**Lz77**	**Lz78**	**gzip**
english.1MB	1046186	0.004773	0.085952	0.008929
english.2MB	2021679	0.010089	0.20971	0.015661
english.5MB	5019248	0.017139	0.589746	0.04545
english.10MB	10047797	0.029844	1.346426	0.089052
english.25MB	24969228	0.071658	3.737625	0.186638
english.50MB	52034894	0.37604	8.383623	0.434993
english.100MB	104857600	0.891746	16.232986	0.708241
english.200MB	209715200	1.65344	37.663101	1.771319
english.500MB	498654094	5.955645	100.350189	5.83149

O Lz77 mostrou-se na descompressão bastante rápido e competitivo, com tempos equiparáveis ao *gzip* . Já o Lz78 se comporta bem, mas deixa de ser competitivo a partir de arquivos com tamanhos de 200mb.

Com bases nas nossos testes e análises pudemos então escolher o Lz78 como algoritmo padrão para o programa *ipmt*. A principal motivação é explorar sua rapidez na compressão e crescente desempenho no tamanho do arquivo gerado, nestes casos ele mostra-se mais competitivo ao *gzip*. Mas assumindo o Lz78 como padrão aceitamos o tradeoff do desempenho da descompressão que é mais lenta quando comparado ao Lz77.

Testes fim a fim

A título de análise fizemos testes da performance do programa de maneira fim a fim, de forma a analisar o procedimento por completo. Embora não tenhamos parâmetros para fazer uma comparação válida os testes servem de base para uma previsão e entendimento do comportamento do programa. Novamente a métrica utilizada foi o tempo médio de execução e os data sets utilizados disponíveis em:

- http://bit.ly/1RqDs8w

Execução do ipmt no modo de indexação.

Indexação		
Arquivo	*bible.txt*	*arquivo.txt*
Tamanho	4MB	9.5MB
Tempo médio	75.80s	196.77s
Tamanho (.idx)	28MB	70MB

Execução do ipmt no modo de busca.

Busca		
Arquivo	*bible.txt*	*arquivo.txt*
Tamanho	4MB	9.5MB
Tempo médio	50.57s	113.73s
Padrão	God	LLL
Ocorrências	4040	27552