IPMT - Indexed Pattern Matching Tool

Projeto da disciplina de *Processamento de Cadeias de Caracteres* (IF767) do Centro de Informática - CIn, UFPE, ministrada pelo professor Paulo Fonseca.

Alunos

George Oliveira (ghao@cin.ufpe.br)
Horácio Filho (hjcf@cin.ufpe.br)
Lucas Souza (lns2@cin.ufpe.br)

Implementação

A ferramenta foi desenvolvida usando a linguagem C++, em seu padrão C++11. As contribuções de cada aluno ao projeto podem ser visualizadas aqui.

Indexação

As estruturas de dados utilizadas para a construção dos índices são:

- Array de Sufixos (***Suffix Array***), por padrão;
- ou Árvore de Sufixos (***Suffix Tree***), através de uma flag (veja *Utilização e Funcionamento Básico*).

Compressão

Para a compressão e descompressão, utilizamos os algoritmos:

- **LZW** (Lempel-Ziv-Welch), uma modificação do LZ78, por padrão;
- ou **LZ77** através de uma flag (veja *Utilização e Funcionamento Básico*).

Estruturas de Dados

As seguintes estruturas de dados foram implementadas para este projeto:

- **SuffixArray:** utilizada por padrão porque, em geral, utiliza menos memória.

- **SuffixTree:** colocada como opcional porque nossa implementação utiliza muita memória e mostrou-se menos eficiente que a SuffixArray.

Ambas implementações possuem os métodos:

- `getRepr()`: responsável por gerar uma string que represente a estrutura de dados e que será utilizada para construir o arquivo .idx , de indexação do arquivo de entrada.

- `loadRepr(string &rep)`: recupera a estrutura de dados original a partir de sua representação.

Estratégia de I/O

A leitura dos arquivos de entrada, tanto arquivos de texto quando os de índice (.idx), são feitas linha por linha. O resultado dessa leitura, i.e., o arquivo de texto inteiro, é passado para as estruturas de dados correspondentes para que estas sejam inicializadas.

Para a compressão, cada linha do arquivo .idx é escrita logo após esta ser gerada, evitando o uso de grandes quantidades de memória.

Limitações

Para arquivos maiores que 50 MB, a execução do programa é muito lenta, tendo em vista a grande utilização de memória e implementações ingênuas. Além disso arquivos com tamanho maior que 30MB não podem ser utilizados como entrada para Suffix Tree: devido a grande utilização de ponteiros, o erro **bad_alloc** é lançado.

Utilização e funcionamento básico

Compilação

A partir da raiz do projeto, a compilação pode ser realizada da seguinte maneira para cada plataforma suportada:

- Linux: `make all`, executando o comando via terminal.
- Windows: `compile.bat`, executando o programa.

Após a compilação, o programa estará disponível no diretório `/bin` gerado.

Utilização

- **Indexação**:

```
$ ipmt index [options] textfile
```

Opções:

- ❏ `-a` : utiliza um Array de Sufixos para construir os índices **(ativado por padrão)**.
- ❏ `-t` : utiliza uma Árvore de Sufixos para construir os índices.
- ❏ `-compress=lz77` : utiliza o algoritmo LZ77 para comprimir o arquivo de índices.
- ❏ `-compress=lzw` : utiliza o algoritmo LZW para comprimir o arquivo de índices **(ativado por padrão)**.

- **Busca:**

```
$ ipmt search [options] [pattern] indexfile [indexfile...]
```

Opções:

- ❏ `-p file` : realiza a busca utilizando cada linha de `file` como um padrão.
- ❏ `-c` : apenas imprime a quantidade de *matchs* que ocorreram para cada padrão.

Além das opções específicas para cada modo já descritas acima, ainda há a flag `-h` que exibe uma ajuda ao usuário sobre como utilizar o programa.

A saídas para a busca têm o seguinte formato:

- Padrão:

```
$ textfile:pos: pattern
```

Onde:
- `textfile` é o nome do arquivo de texto onde está sendo realizada a busca;
- `pos` é o índice do texto onde ocorre o match (considerando o texto de entrada como uma única string);
- `pattern` é o padrão procurado no texto.

- Usando a flag `-c, --count`:

```
$ textfile: num_occ occurences for pattern
```

  Onde:
  - `num_occ` é o número de ocorrências do padrão no texto.

Testes

Descrição do ambiente de testes

Os testes foram realizados utilizando um computador com as seguintes especificações:

- Sistema Operacional: Debian Linux 64-bit 3.16.0-34-generic
- Processador: Intel Core i7 1.90 GHz
- RAM: 4GB

Os arquivos de texto utilizados foram construídos a partir do arquivo *english.50MB* disponível aqui utilizando um script em Python (disponível aqui) para remover caracteres nulos e coletar prefixos do texto original possuindo os tamanhos (em MB) desejados (e.g., 5MB, 10MB, 20MB, 30MB, 50MB).

A execução dos testes foi controlada a partir deste script em Python, responsável por coletar os tempos de execução e gerar os gráficos utilizando a biblioteca **matplotlib** para avaliação posterior do desempenho do programa.

Cenários de teste

Foram executadas as seguintes situações:

- Tempo para compressão e descompressão, comparando LZ77, LZW e a ferramenta **gzip.**

- Tempo para a criação dos arquivos de índice: utilizando um arquivo combinando as opções de Array de Sufixos e LZ77 e LZW para compressão, num total de 4 cenários (Dadas as limitações da nossa implementação de Suffix Tree, esta estrutura não foi utilizada nestes cenários).

- Tempo para a busca em um arquivo de 5MB comparando SuffixArray, SuffixTree e **grep,** utilizando padrões de vários tamanhos e combinando as opções LZW e LZ77.

- Tempo para a busca em um arquivo de texto com 50MB utilizando Suffix Array para a construção dos índices padrões de vários tamanhos e comparando com a ferramenta **grep.**

Resultados e conclusões

(a) Compressão

Tamanho do arquivo:	LZW	LZ77	gzip
5 mB	0m2.172s	0m19.664s	0m0.449s
10 mB	0m4.802s	0m38.979s	0m0.780s
15 mB	0m8.548s	1m2.210s	0m1.096s
20 mB	0m9.792s	1m15.789s	0m1.471s
30 mB	0m16.164s	1m57.867s	0m2.133s
50 mB	0m31.592s	3m26.350s	0m3.560s

(b) Descompressão

Tamanho do arquivo:	LZW	LZ77	gzip
5 mB	0m0.636s	0m0.926s	0m0.080s
10 mB	0m1.184s	0m2.088s	0m0.113s
15 mB	0m1.701s	0m3.126s	0m0.142s
20 mB	0m2.091s	0m4.134s	0m0.196s
30 mB	0m3.049s	0m6.025s	0m0.339s
50 mB	0m5.327s	0m13.181s	0m0.445s

(c) Indexação (utilizando Suffix Array)

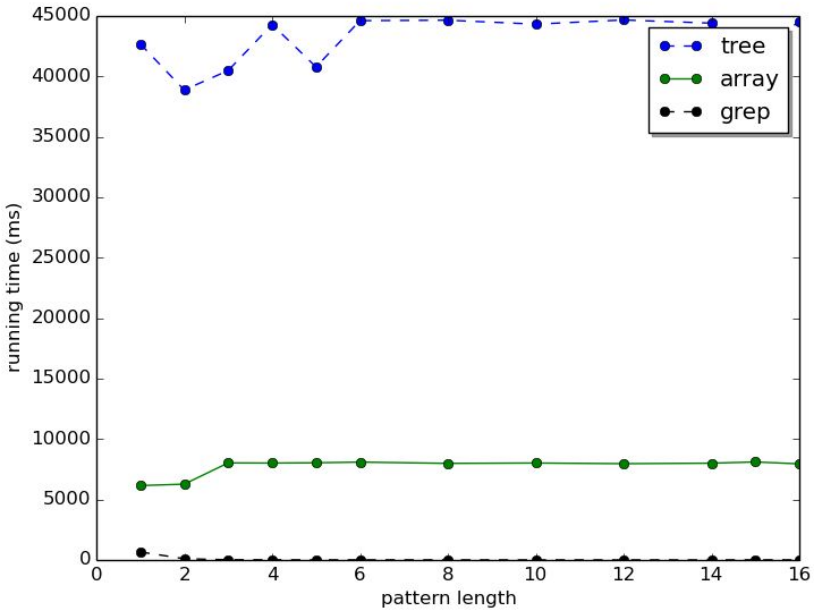
Tamanho do arquivo:	LZW	LZ77
5 mB	0m44.710s	2m19.365s
10 mB	1m36.241s	6m23.222s
15 mB	2m32.514s	7m29.178s
20 mB	3m22.479s	13m42.767s
30 mB	5m23.698s	15m39.701s
50 mB	9m11.241s	28m9.610s

(d) Busca

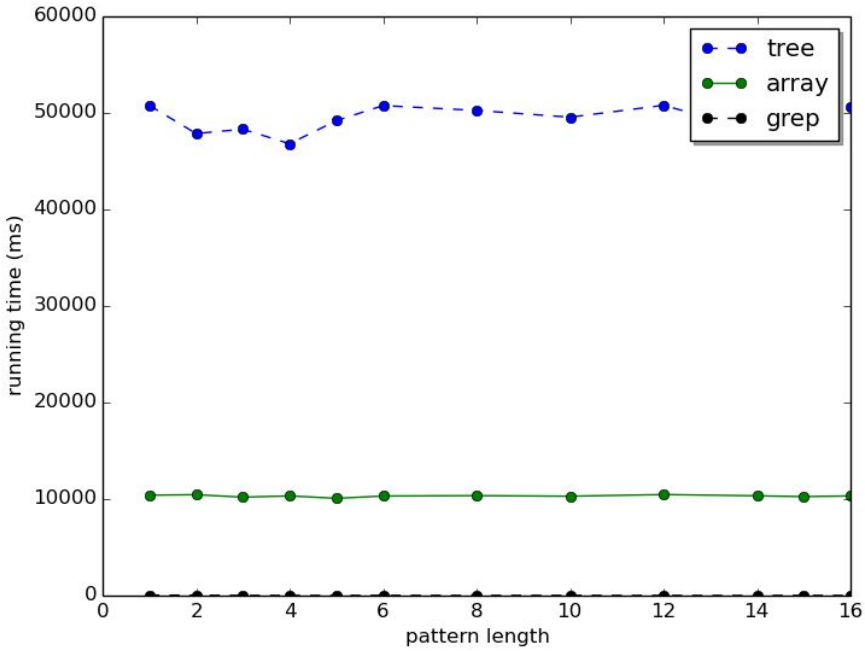
(i) Suffix Tree x Suffix Array x Grep - 5MB

(1) Padrões pequenos (tamanho entre 1 e 16)

a) LZW

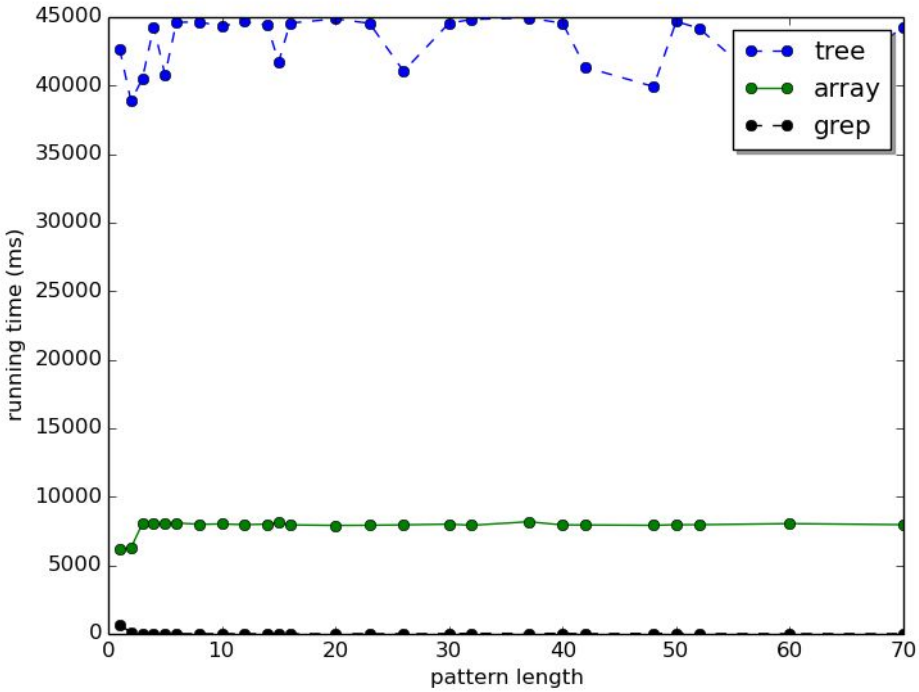


b) LZ77

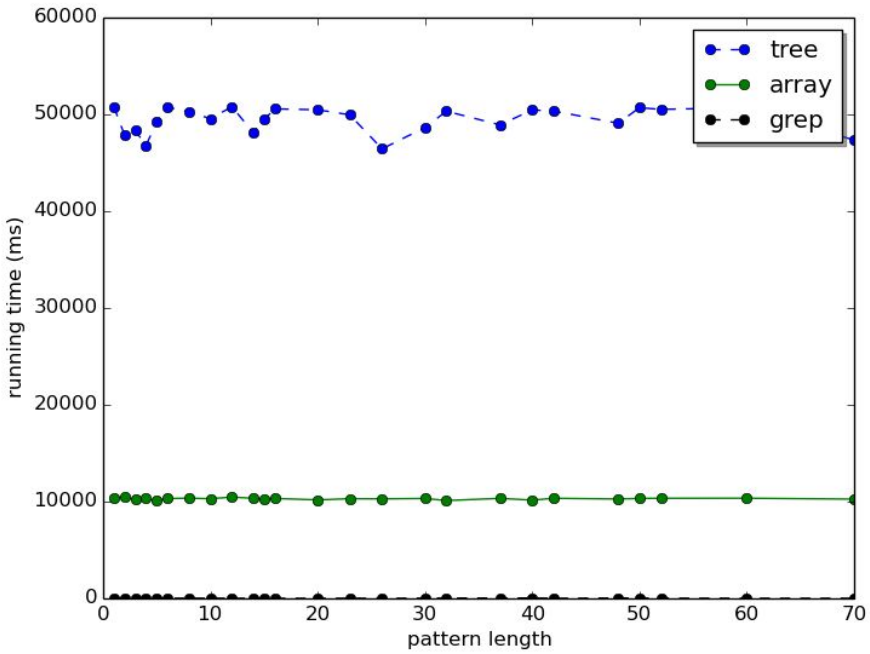


(2) Padrões médios (tamanho entre 20 e 70)

a) LZW

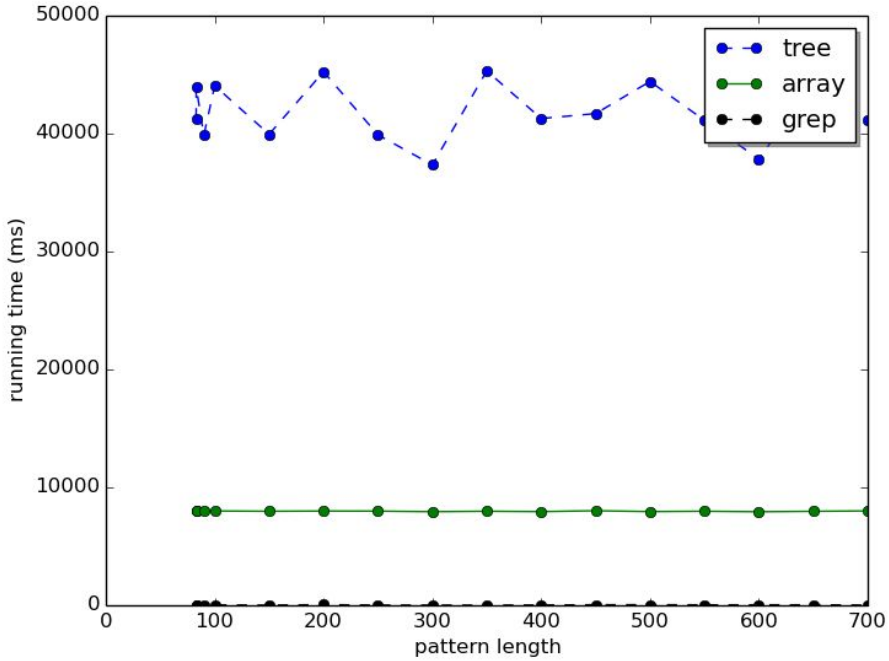


b) LZ77

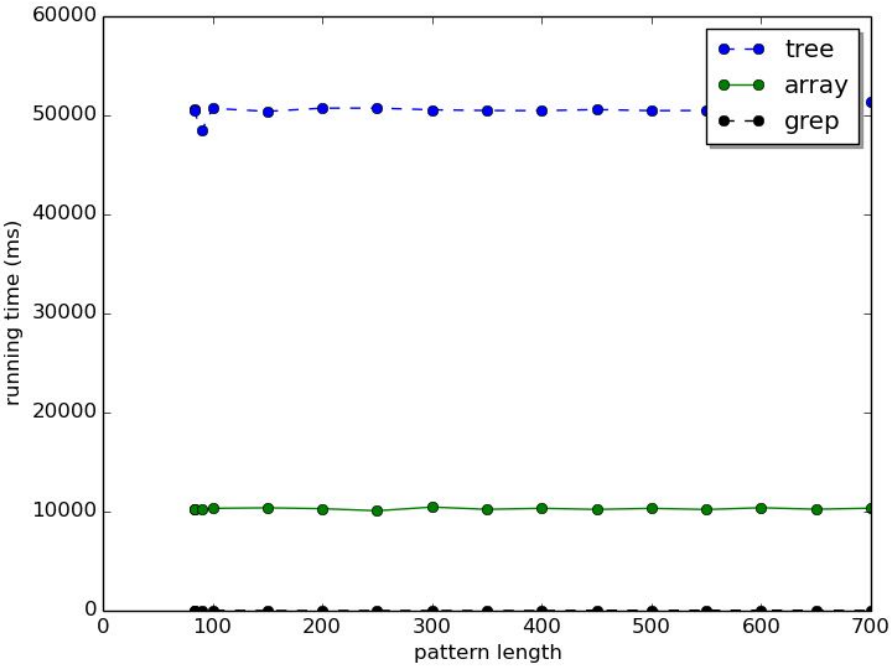


(3) Padrões grandes (tamanho entre 80 e 700)

a) LZW



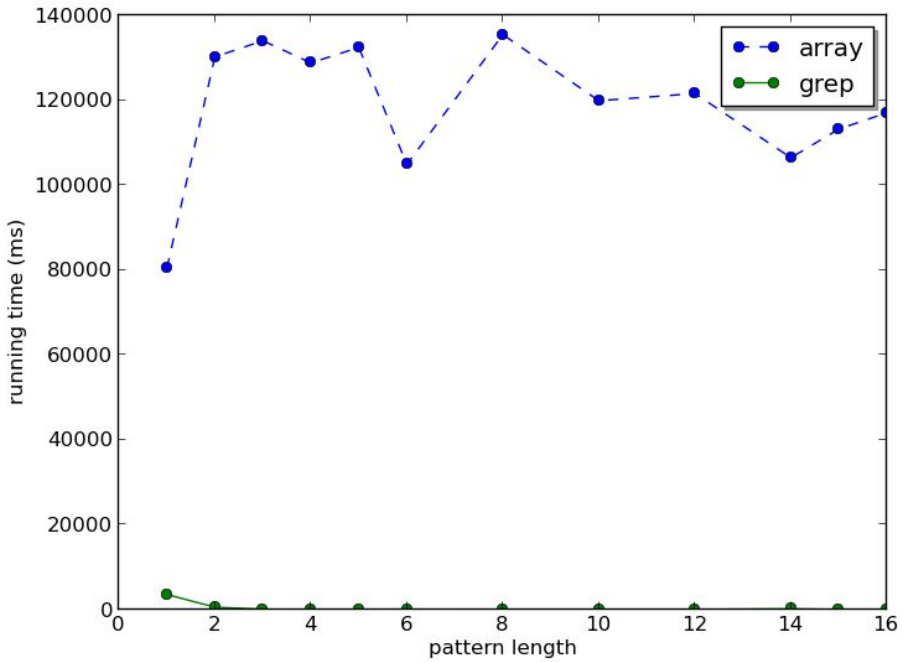
b) LZ77



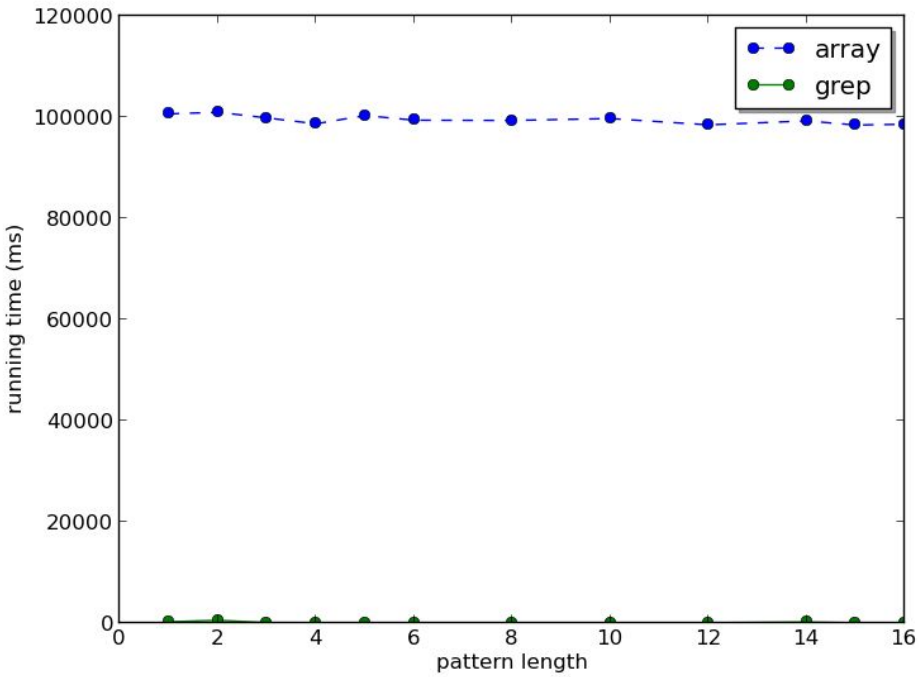
(ii) Suffix Array x Grep - 50MB

(1) Padrões pequenos (tamanho entre 1 e 16)

a) LZW

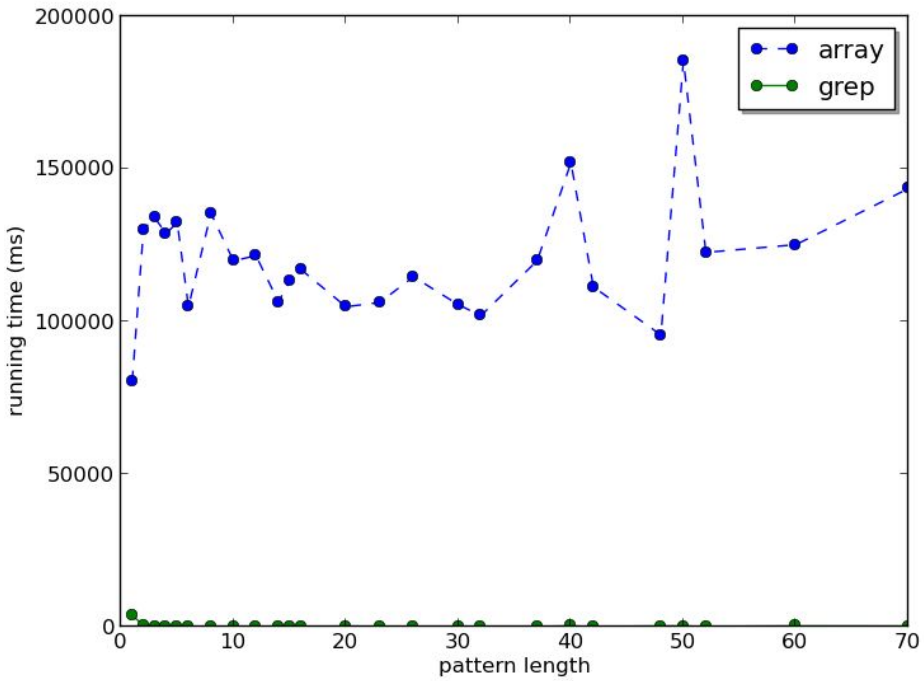


b) LZ77

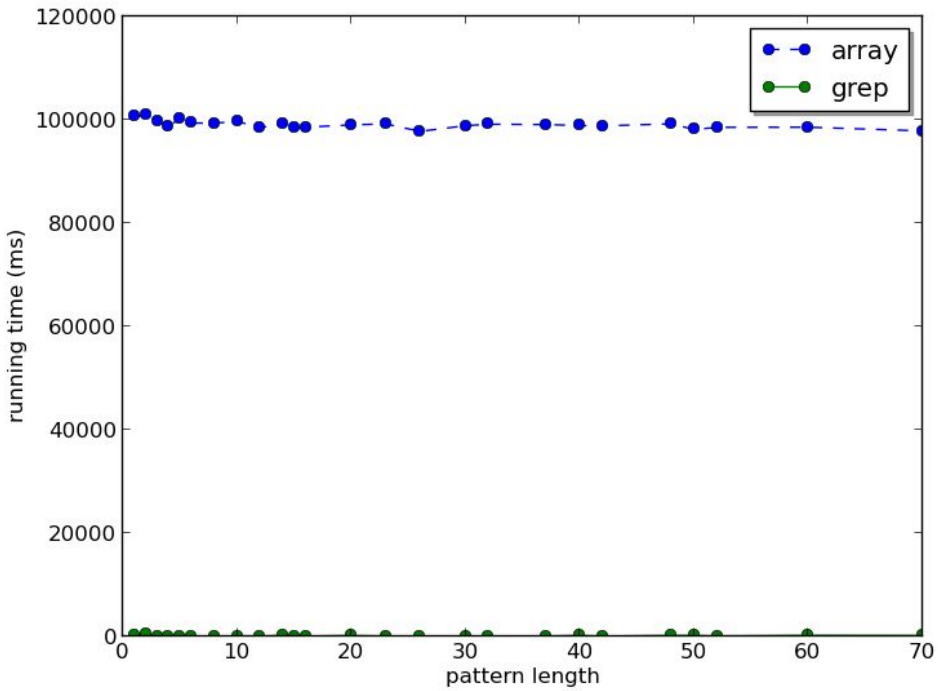


(2) Padrões médios (tamanho entre 20 e 70)

a) LZW

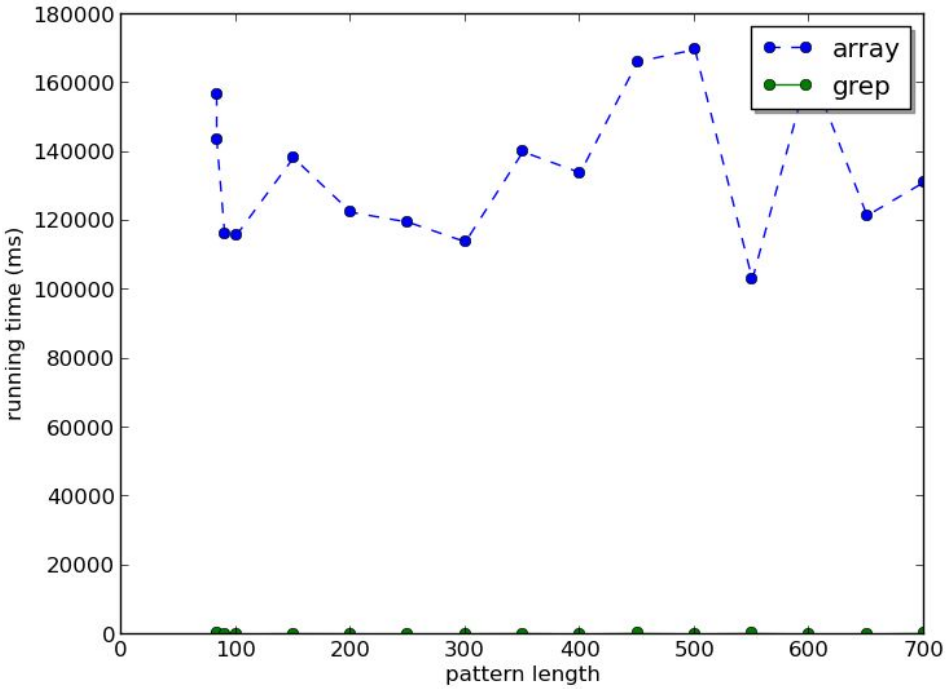


b) LZ77

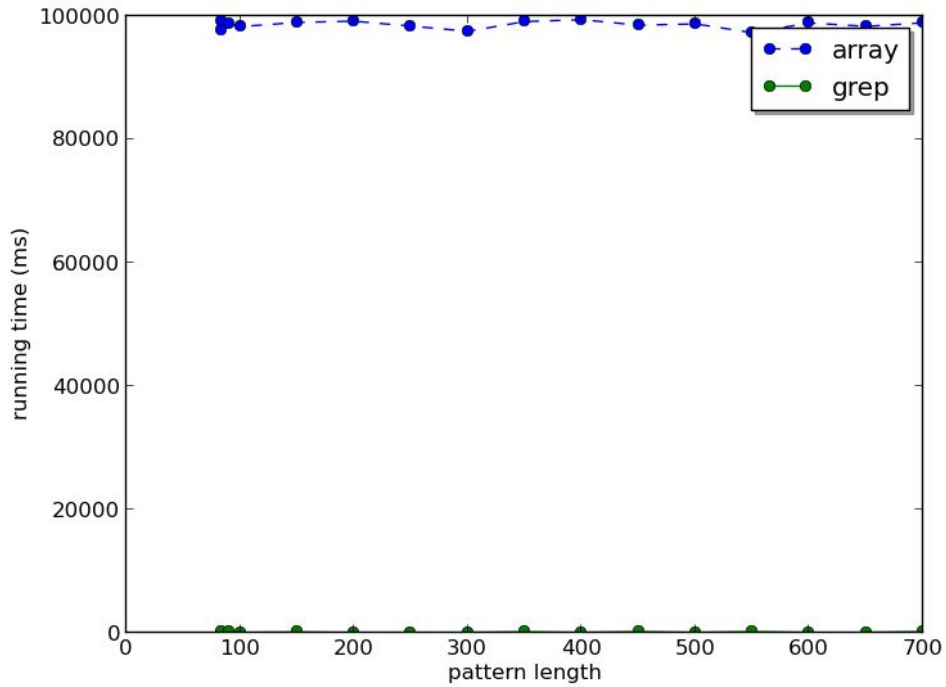


(3) Padrões grandes (tamanho entre 80 e 700)

a) LZW



b) LZ77



- **Conclusões:**

(1) **Compressão:** Percebe-se que LZ77 é aproximadamente 10 vezes mais lento que o LZW que, por sua vez, é aproximadamente 10 vezes mais lento que o gzip. Justificando, dessa forma, a escolha do LZW como o algoritmo de compressão padrão.

(2) **Descompressão:** Neste caso, o LZ77 é aproximadamente 2 vezes mais lento que o LZW, que é quase 10 vezes mais lento que o gzip.

> **OBS:** Nossa implementação do LZW utiliza a estrutura **unordered_map**, de dados padrão do C++, que é, grosso modo, um <map> cujo tempo de lookup é muito baixo. Dessa forma, o LZW possui um desempenho melhor que implementações "ingênuas".

(3) **Indexação:** Verifica-se que a combinação SuffixArray + LZW é mais eficiente, justificando a escolha dessa combinação como padrão.

(4) **Busca:**

(a) **Array x Tree x Grep (5MB):** Verifica-se que o match utilizando Array possui um desempenho mais próximo ao do **grep** e de aproximadamente 5 vezes melhor que o match utilizando Tree, justificando, mais uma vez a escolha dos algoritmos padrão. Além disso, o desempenho utilizando Array é quase constante, enquanto que o utilizando Tree é bastante instável.

(b) **Array x Grep (50MB):** Nesse caso, o desempenho para a busca em arquivos grandes é quase constante utilizando LZ77 para a compressão, enquanto que utilizando LZW é bastante instável. Se realizados mais testes, talvez seja possível verificar que, de fato, para arquivos muito grandes, a combinação SuffixArray + LZ77 seja a mais eficiente, indicando uma possível melhoria e heurística de combinação dos algoritmos.