Ferramenta IPMT

Márcio Barbosa de Oliveira Filho - mbof@cin.ufpe.br

1 Implementação

1.1 Decisões de Implementação Relevantes

1.1.1 Árvore de Sufixos

Uma decisão importante em relação à construção da árvore de sufixo é a maneira pela qual ela será representada. Escolhemos representá-la como uma *árvore binária*. Cada um de seus vértices possui um ponteiro para o primeiro de seus filhos e outro para seu irmão na árvore "original".

A árvore como um todo é armazenada em um vetor de vértices. Dessa forma, podemos nos referir a um vértice através de sua posição neste vetor. Isso facilita bastante a serialização da estrutura no momento de armazená-la em disco.

Para salvar um pouco de memória, compartilhamos, através do uso de um *union*[1], a memória utilizada por dois inteiros de cada nó da árvore: o campo que guarda o *suffix link* (*sl*) e o que armazena a quantidade de folhas em determinada subárvore (*leaves*). Isso é possível porque *sl* só é usado durante a construção da árvore e *leaves* apenas durante a fase de consultas.

A maneira mais direta de serializar a árvore para salvá-la em um arquivo é salvar *todos* os seus campos. No entanto, essa abordagem é ineficiente porque a árvore de sufixo cresce bastante, gerando arquivos muito grandes e aumentando o custo de compressão e descompressão. Para contornar essa dificuldade decidimos sacrificar um pouco de processamento para que não fosse necessário salvar todos os campos da árvore de sufixo. Mais especificamente, salvamos em disco apenas o rótulo da aresta incidente a cada nó. Lembramos que esse rótulo é representado por um par de inteiros. A grande vantagem dessa abordagem é que os inteiros que precisamos salvar no arquivo são todos $\leq n$, em que n é o tamanho do texto. Com isso, cada um deles exige $\log(n)$ *bits* para ser armazenado.

Naturalmente precisamos serializar a árvore de modo a conseguir remontá-la posteriormente de maneira não ambígua. Para isso utilizamos uma estratégia baseada em uma busca em largura. Toda vez que um vértice *entra* na fila de busca ele é salvo, ou seja, o par representando a aresta incidente a ele é armazenado. Toda vez que um vértice *deixa* a fila, seus filhos são adicionados na fila e, portanto, salvos.

[1] UNION

Indicamos o final da lista de adjacência de um vértice com um *bit* 1. Ou seja, toda vez que um vértice deixa a fila de busca, seus filhos são armazenados seguidos de um *bit* 1. Como consequência, quando um vértice entra na fila temos também que precedê-lo de um *bit* 0. Com isso a reconstrução da árvore fica fácil. Toda vez que lermos um 0 no arquivo, sabemos que devemos ler um vértice e colocá-lo na fila. Seu pai na árvore será o vértice na cabeça da fila. Toda vez que lermos um 1 no arquivo sabemos que a lista de adjacência do vértice na cabeça da fila acabou e que, portanto, ele deve ser removido.

Após esse processamento, na reconstrução da árvore, precisamos também calcular o campo *leaves* de cada nó. Esse campo indica quantas folhas existem na subárvore enraizada em um determinado nó. Tudo o que precisamos fazer, nesse caso, é varrer os vértices da árvore começando no último vértice do vetor e indo até o primeiro (a raiz). Isso funciona porque a árvore é armazenada em uma ordem topológica adequada. Ou seja, todos os filhos de um determinado nó estão em uma posição de índice superior no vetor de vértices (ou seja, à sua "direita").

1.2 *Array* de Sufixos

De maneira semelhante da que fizemos com a árvore de sufixos, tentamos separar bem os campos do *array* de sufixos para que instanciássemos apenas aqueles que fossem ser utilizados.

Utilizamos um algoritmo de ordenação linear, o *radix sort*, para alcançar uma implementação de custo $O(n \log(n))$.

O armazenamento do array de sufixos é bem mais simples que o da árvore de sufixos. Precisamos apenas salvar o *array* de sufixos propriamente dito e os vetores auxiliares *lLcp* e *rLcp* definidos como durante o curso. Ressaltamos que todos os inteiros nestes vetores são limitados por n e, portanto, precisam de $\log(n)$ *bits* para serem representados.

O algoritmo visto no curso, e por nós implementado, utiliza um vetor P de tamanho $n \log(n)$. Durante a construção do *array* de sufixos, no entanto, apenas as duas últimas posições interessam. Armazenar apenas essas linhas representa, portanto, uma grande economia de memória. No entanto *todo* o vetor P é exigido para o cálculo dos auxiliares *lLcp* e *rLcp*.

Para evitar manter todo o vetor P utilizamos uma estratégia diferente para a construção dos vetores auxiliares. Primeiramente definimos o vetor *lcp* de forma que $lcp[i]$ é o maior prefixo comum entre os sufixos $sa[i]$ e $sa[i-1]$. Porque ele guarda o *lcp* de sufixos *consecutivos* no *array* de sufixos, conseguimos construir o vetor *lcp* em tempo linear. Note que o maior prefixo comum entre dois sufixos $sa[j]$ e $sa[k]$ quaisquer é o menor valor entre $lcp[j+1]$ e $lcp[k]$, supondo que $j+1 \leq k$. Com essa observação, podemos utilizar uma estratégia do tipo *dividir para conquistar* para preencher os valores de *lLcp* e *rLcp*. Para tanto implementamos uma função recursiva que simula todos os intervalos válidos de uma busca binária. O menor valor de *lcp* contido em determinado intervalo é calculado recursivamente. Essa função gera no máximo $2n$ chamadas recursivas e, portanto, monta os vetores auxiliares em tempo linear.

1.3 Compressão e Descompressão

De maneira geral, evitamos ao máximo trabalhar com *bits* individualmente. Elegemos como alfabeto dos nossos algoritmos de compressão os *bytes*.

Nossos algoritmos, de maneira geral, são *on-line*. Recebemos *bytes*, possivelmente incompletos, na medida em que eles são gerados durante a serialização dos índices. Alguns detalhes relevantes de cada algoritmo serão apresentados nas seções seguintes.

1.4 LZ78

A operação central do LZ78 é manter um dicionário de cadeias que possibilite as operações de busca e adição de forma eficiente. Como vimos durante o curso, podemos representar esse dicionário eficientemente através de uma *trie*. No entanto, a escolha do nosso alfabeto (*bytes* ao invés de *bits*) trouxe a complicação de como representar a lista de adjacência dos nós da *trie*. Uma estratégia semelhante à utilizada na árvore de sufixo acaba por aumentar muito o tempo de execução, pois encarece a operação de busca, a qual é muito utilizada. A utilização de um vetor com 256 posições ou outra estrutura associativa (uma árvore, por exemplo) aumenta bastante o consumo de memória e acaba por afetar também o tempo de execução e diminuir a escalabilidade da ferramenta.

Vimos também que uma tabela *hash* pode servir como dicionário. Um dos problemas desta estrutura, como abordado durante o curso, é que o cálculo do valor *hash* de uma sequência não é, em geral, constante. Isso pode ser contornado se pudéssemos controlar a maneira pela qual o *hash* é gerado como, de fato, podemos. No entanto, não podemos negligenciar outro aspecto importante da tabela *hash*: a resolução de colisões. Caso uma mesma posição da tabela contivesse mais de um elemento, teríamos que comparar as *chaves* de cada um deles para saber qual é o desejado. Ou seja, teríamos que comparar sequências[2].

Na nossa implementação decidimos utilizar uma abordagem mista. Conceitualmente trabalhamos com uma *trie*, mas a implementamos com uma tabela *hash*. Dessa forma, representamos as arestas da *trie* através de pares *(nó, byte)* em que *nó* é um vértice da *trie* e *byte* é um elemento do alfabeto. A vantagem dessa indexação é que o cálculo do *hash* é feito de maneira constante e a resolução de colisões não é encarecida.

Uma desvantagem dessa estratégia é que ela sofre com o crescimento do dicionário do LZ78. Esse crescimento aumenta a ocorrência de colisões na tabela *hash* e acaba encarecendo suas operações. Para lidar com isso limpamos completamente o dicionário quando ele chega a determinado tamanho. Naturalmente que quanto menor for o tamanho máximo do dicionário mais rápido e menos eficaz o algoritmo será. Ou seja, esse compromisso entre rapidez e qualidade de compressão precisa ser observado. Por isso, criamos a opção *level* que influencia o tamanho máximo do dicionário e é fornecida através da linha de comando.

[2] Alternativamente poderíamos adotar uma estratégia de *double* ou *triple hash*, por exemplo. Mas em algum momento teríamos que ou comparar sequências ou admitir que colisões não aconteceriam.

O padrão é limitar o tamanho do dicionário em metade do tamanho da tabela *hash*.

Ressaltamos que o algoritmo de descompressão armazena o dicionário como um vetor, uma vez que não é necessário casar uma sequência nesse caso.

Outra decisão relevante que tomamos foi a de escolher não comprimir uma sequência quando isso não for vantajoso. Ou seja, se a codificação da tupla emitida pelo LZ78 for mais longa que a da codificação da sequência que ela representa, então o algoritmo emite a sequência descomprimida. Para viabilizar esse comportamento precedemos cada tupla com um *bit* 1 e cada sequência descomprimida de tamanho x com x *bits* 0 e um *bit* 1. Verificamos que isso trouxe uma melhora à qualidade de compressão do algoritmo.

Por fim, os inteiros emitidos pelo LZ78 têm tamanho variável e, por isso, precisam ser codificados. A codificação $C(x)$ de um inteiro com representação binária x é a seguinte cadeia: $|t|_1 + 0 + |x| + x$, em que $t = |x|$. Ou seja, precedemos x com o seu tamanho, um *bit* 0 de marcação e o tamanho *do tamanho de* x em unário. Dessa forma $|C(x)| = O(\log\log(|x|) + 1 + \log(|x|) + |x|)$. Essa codificação nos pareceu estar mais alinhada com nossa estratégia geral de evitar lidar com *bits* individualmente.

1.4.1 LZW

O LZW é uma variação do LZ78 na qual o dicionário é inicialmente preenchido com todos os caracteres do alfabeto. Com isso, quando uma sequência Xa não está no dicionário, o algoritmo emite apenas o índice de X ao invés do par (X, a) emitido pelo LZ78.

Ressaltamos que todas as decisões de implementação do LZ78, tanto na compressão quanto na descompressão, se aplicam à nossa implementação do LZW.

1.4.2 LZ77

Uma das operações mais caras do LZ77 é a busca de um casamento de um prefixo da janela de pré-visualização (*lookahead*) na janela de texto já processado (*buffer*). Nossa primeira estratégia foi utilizar o *KMP*, recalculando a função de falha para o *lookahead* e percorrendo o *buffer* linearmente. Ela, no entanto, ainda deixava a operação de busca muito cara.

A estratégia implementada armazena a janela de *buffer* em uma árvore binária de busca. Na árvore guardamos os índices contidos na janela de *buffer* e os comparamos através das cadeias que se iniciam neles. Ou seja, *conceitualmente* cada nó da árvore armazena um sufixo da janela de *buffer* com no máximo o tamanho da janela de *lookahead*.

Para encontrar o maior casamento para um prefixo da janela de *lookahead* nós começamos na raiz da árvore. Em cada nó que visitamos calculamos o maior prefixo comum entre a janela de *lookahead* e a cadeia armazenada no vértice corrente. Esse pode ser a resposta. Para tentar encontrar um prefixo maior nos baseamos em se a cadeia do vértice é maior ou menor (lexicograficamente) que a

janela de *lookahead*. Se ela for maior, então precisamos buscar cadeias menores e, portanto, continuamos com a subárvore à esquerda. Caso contrário vamos para a subárvore à direita.

Interrompemos a busca sempre que o maior prefixo comum for igual à própria janela de *lookahead* ou quando tentarmos visitar uma subárvore vazia. Se wb é o tamanho da janela de *buffer* e wl o da de *lookahead*, o custo deste procedimento é $O(wl \log(wb))$ se a árvore for balanceada. Além disso, quando deslizamos a janela em j posições, temos que fazer j remoções na árvore e mais j inserções, com o custo total de $O(j \cdot wl \cdot \log(wb))$.

A árvore que utilizamos foi a *treap*[3]. Ela é uma árvore balanceada com *alta probabilidade*. Um dos atrativos para seu emprego no nosso projeto é sua facilidade de implementação e verificação de corretude.

2 Testes

2.1 Algoritmos de Compressão

Realizamos testes de compressão entre diferentes configurações dos algoritmos implementados e a ferramenta de compressão *gzip*. As configurações testadas foram as seguintes:

- LZ77 com janela de *buffer* com 128 *bytes* e janela de *lookahead* com 8 *bytes*,
- LZ77 com janela de *buffer* com 1024 *bytes* e janela de *lookahead* com 16 *bytes*,
- LZ77 com janela de *buffer* com 4096 *bytes* e janela de *lookahead* com 32 *bytes*,
- LZ78 com os três níveis de compressão: 0, 1 e 2,
- LZW também com os três níveis de compressão: 0, 1 e 2.

O nível de compressão 0 permite que o dicionário do LZ78/W tenha no máximo 2^{16} elementos. O nível 1 permite que ele tenha 2^{19} elementos e o nível 2 permite que ele tenha capacidade virtualmente infinita (2^{31}).

No primeiro cenário de testes apresentamos aos algoritmos arquivos de texto em inglês separados por tamanho. Cada tamanho foi representado por 3 arquivos diferentes e cada um deles foi apresentado 3 vezes aos algoritmos. O tempo de execução para cada tamanho foi calculado como a média dessas 9 execuções. Já o tamanho do arquivo comprimido foi calculado como a média dos tamanhos dos arquivos gerados em cada execução. Registramos os resultados nos gráficos das figuras 1, 2 e 3.

Separamos os tempos de execução do teste 1 nas figuras 1 e 2 apenas para facilitar a visualização. Note que os tempos do algoritmo LZ77 foram bem

[3] https://en.wikipedia.org/wiki/Treap

Tempo de Execução do Teste 1 - Sem o LZ77

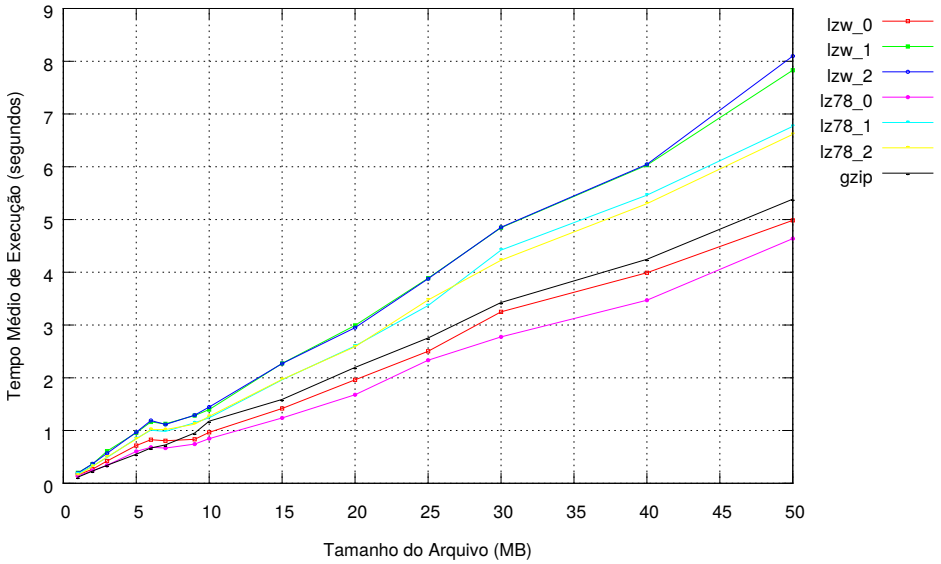

Figura 1

Tempo de Execução do Teste 1 - LZ77

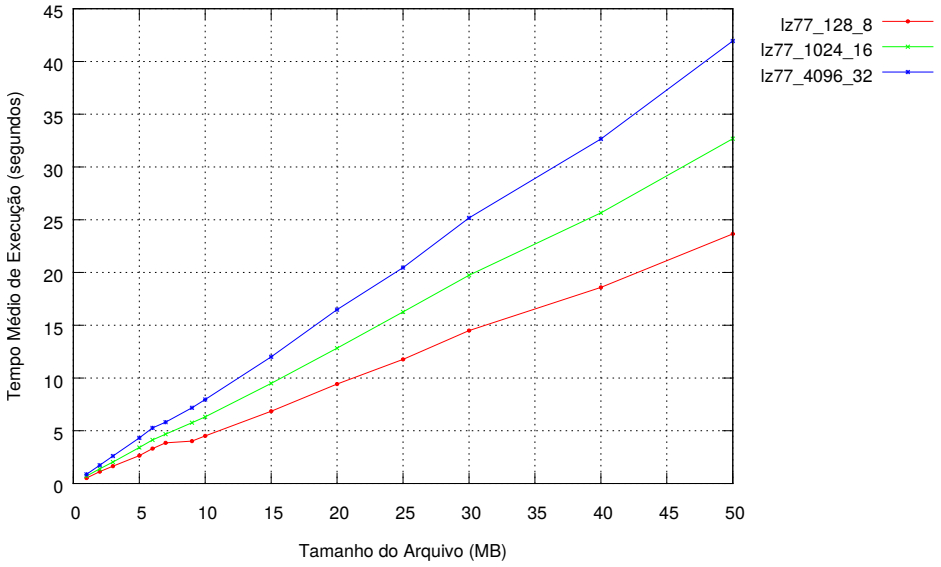

Figura 2

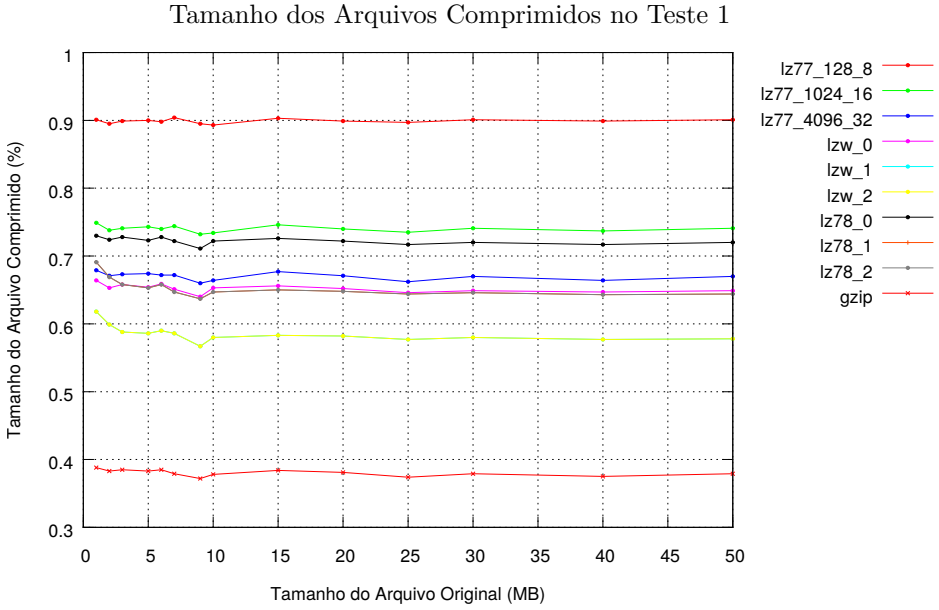

Figura 3

superiores aos demais em todas as suas configurações testadas. Os níveis 1 e 2 de compressão dos algoritmos LZ78 e LZW não causaram grande diferença nos tempos de execução. Todos os algoritmos e também a ferramenta *gzip* apresentaram um crescimento linear do tempo de execução.

Como mostra a figura 3, todos os algoritmos apresentaram um comportamento aproximadamente constante em relação ao tamanho do arquivo comprimido. Note que os níveis 1 e 2 de compressão, tanto do LZ78 quanto do LZW, tiveram resultados indistinguíveis neste teste. A ferramenta *gzip* se mostrou bastante superior em relação à taxa de compressão neste primeiro cenário.

Ainda no mesmo cenário de teste, medimos o tempo médio necessário para descomprimir os arquivos gerados. Como mencionamos, cada tamanho é representado por 3 arquivos distintos. Por isso, o tempo de descompressão é calculado como a média para descomprimir os 3 arquivos gerados na etapa anterior (de compressão). Apresentamos os resultados no gráfico da figura 4.

Em relação à descompressão, os algoritmos apresentaram um crescimento do tempo de execução aproximadamente linear. Algumas anomalias podem ser notadas na figura 4, provavelmente decorrentes de alguma característica particular dos arquivos utilizados. Os tempos de execução de todos os algoritmos são compatíveis, com exceção do *gzip*, que cresce mais devagar. Em particular, o LZ77 consegue bons tempos de execução.

Embora os arquivos de texto nos ajudem a ter uma ideia do comportamento dos algoritmos, os tipos de arquivos que iremos comprimir são binários. Não podemos dizer, a princípio, o quão diferentes eles são dos arquivos de texto,

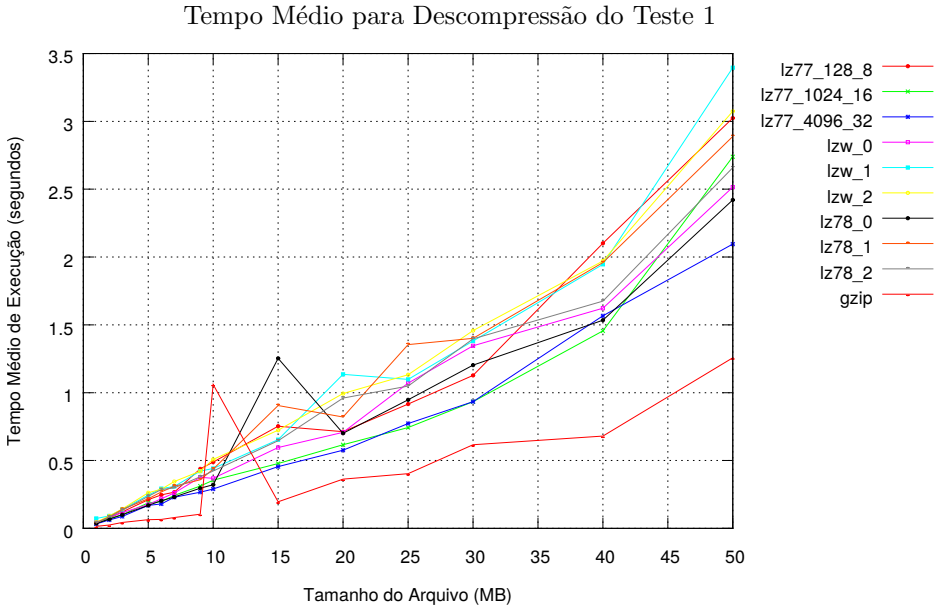

Figura 4

principalmente porque não sabemos o quão redundante eles são e nem como as possíveis repetições presentes neles estão distribuídas. Por essa razão criamos um segundo cenário de testes no qual os arquivos apresentados aos algoritmos são pedaços de arquivos de índices gerados por nossas implementações do *array* de sufixo e da árvore de sufixo.

Utilizamos a mesma estratégia para a geração dos arquivos de teste: separamos os arquivos por tamanho de modo que cada tamanho fosse representado por 3 arquivos. Nossa escolha dos tamanhos, no entanto, foi um pouco diferente. Notamos que os arquivos de índice gerados, os que de fato serão comprimidos, têm, em geral, tamanho bem superior ao dos arquivos de texto original. Por isso, neste cenário de arquivos binários, utilizamos tamanhos maiores. Reportamos os resultados nos gráficos das figuras 5, 6 e 7.

Os tempos médios de execução reportados nas figuras 5 e 6 não chegam a trazer grandes novidades em relação aos testes anteriores. Constatamos mais uma vez que esses tempos crescem linearmente com o tamanho dos arquivos.

A novidade fica por conta do tamanho dos arquivos comprimidos, mostradas no gráfico da figura 7. Alguns arquivos foram "expandidos" por alguns algoritmos. Em particular, o LZ77 com 4096 *bytes* de *buffer* e 32 de *lookahead* chegou a aumentar um arquivo de 50MB em 30%. É interessante também notar que a taxa de compressão não se comportou de maneira constante, como no teste anterior. Em parte isso se deve pela natureza dos arquivos de índice. Isso porque eles contém tanto o texto original quanto a estrutura de dados serializada. Por essa razão, alguns dos arquivos de testes, que foram extraídos aleatoriamente

Tempo de Execução do Teste 2 - Sem o LZ77

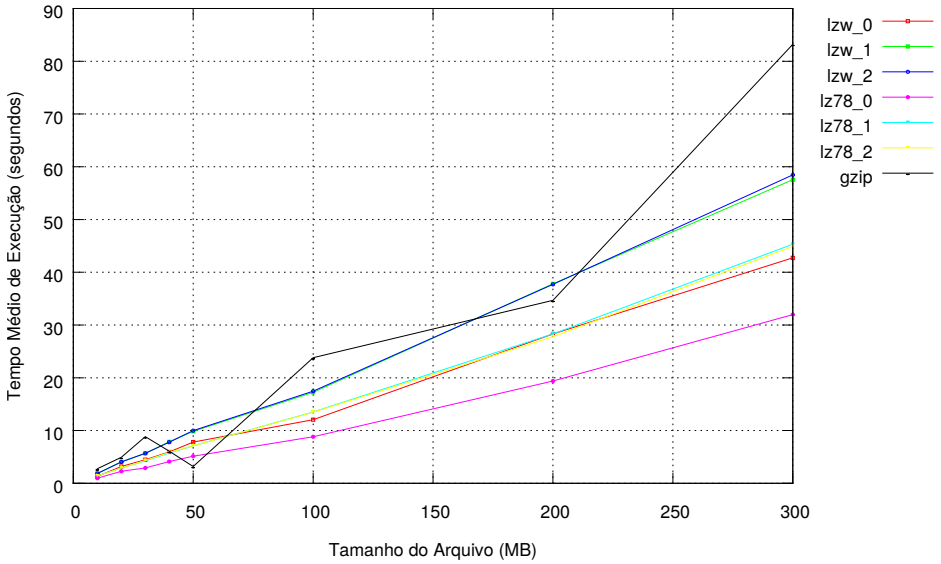

Figura 5

Tempo de Execução do Teste 2 - LZ77

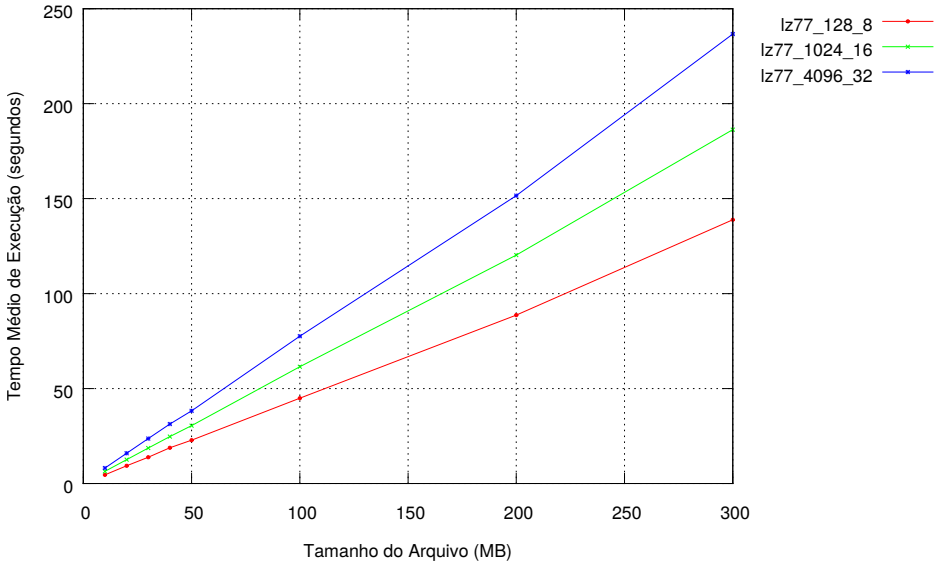

Figura 6

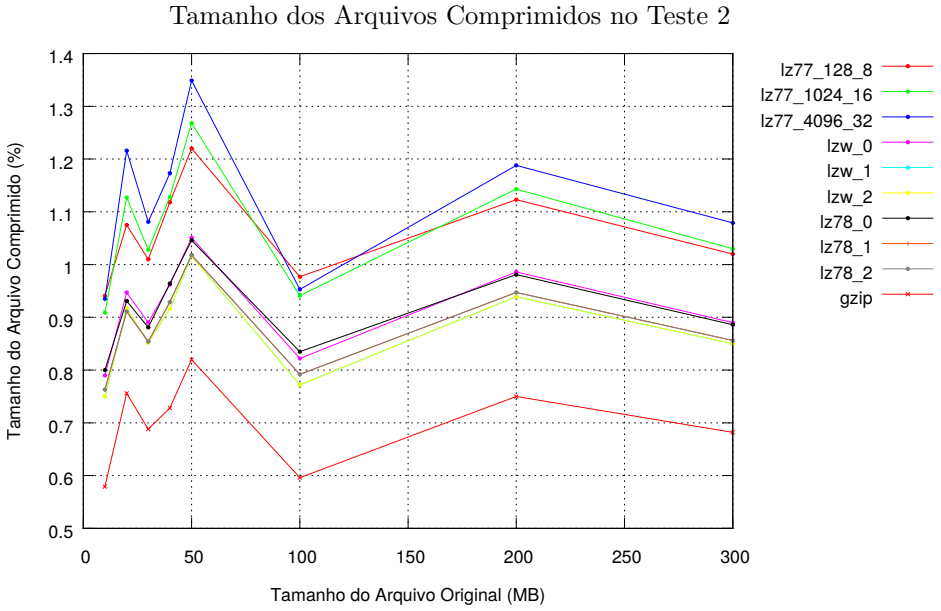

Figura 7

de arquivos de índice, tiveram maior quantidade de texto que outros.

Ressaltamos que também o *gzip* experimentou uma diminuição na sua taxa de compressão quando apresentado aos arquivos binários. Enquanto que no cenário anterior ele reduzia os arquivos a 40% de seu tamanho original, agora ele gera arquivos com cerca de 70% de seu tamanho original. Esse fato atesta que os arquivos binários apresentados são mais "desafiadores" para os algoritmos de compressão.

Por fim reportamos na figura 8 os tempos de descompressão dos arquivos binários comprimidos. Podemos perceber que o *gzip* possui um tempo de execução bastante inferior aos das nossas implementações.

Tempo Médio para Descompressão do Teste 2

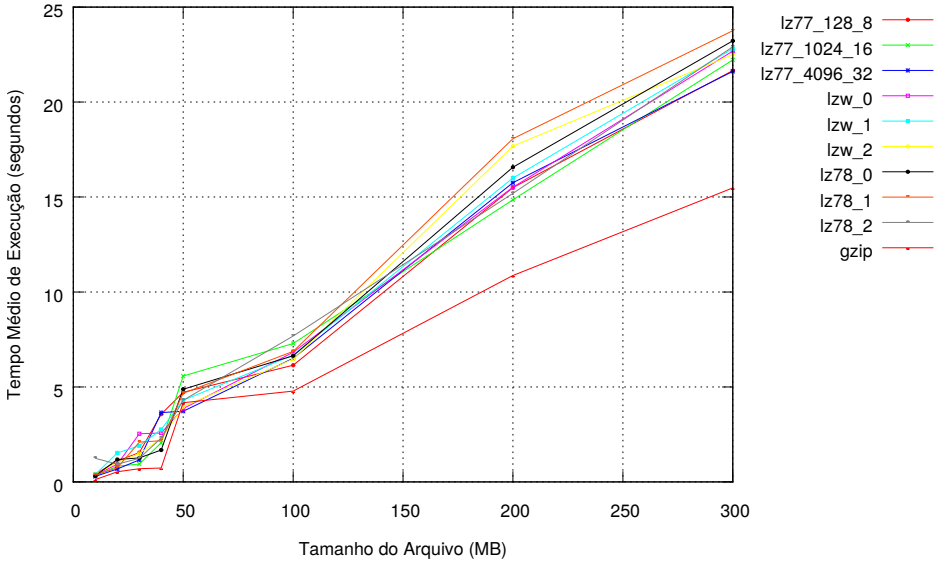

Figura 8