Trabalho Pratico II - Compressor de Arquivos

Joao Paulo Mendes de Sa

1 Introdução

Para quantidades enormes de dados os métodos tradicionais de ordenação não são suficientemente adequados. Ordenação em memoria externa é requerida quando todos os dados a serem ordenados não cabem dentro da memoria principal disponível da maquina. Em vez eles tem que permanecer na memoria externa, o hard drive. Para realizar a ordenação desses dados é necessária uma estrategia de ordenar/intercalar onde na primeira etapa os dados são quebrados em pedaços e cada pedaço e ordenado independente dos outros. Em seguida, na segunda etapa, os pedaços ordenados são juntados para formar um único arquivo ordenado. De modo geral o algorítimo é:

1. Quebra o arquivo em subarquivos que são capazes de caber dentro da memoria disponível.
2. Enquanto não atingir o fim do arquivo:
   (a) Carrega na memoria o numero de items que couber.
   (b) Ordena o conteúdo da memoria.
   (c) Escreva o conteúdo ordenado para um subarquivo arquivo.
   (d) Armazene o arquivo ordenado.
3. Reabra todos os arquivos gerados.
4. Carrega na memoria o primeiro item de cada subarquivo.
5. Crie uma fila de prioridades esses items.
6. Enquanto a fila não ficar vazia:
   (a) Retire e escreva no arquivo de saída o primeiro item da fila.
   (b) Adicione a fila o próximo item do subarquivo onde o ultimo retirado originou.
7. Delete todos os subarquivos intermediários gerados.

2 Implementação

2.1 Estrutura de Dados

Foi criado um tipo abstrato de dados para processar as rodadas gerados, cada rodada composta por um TAD de items que armazenavam cada url e pageviews, e outro para manter a fila de prioridades composta de items.

Para implementar a fila de prioridade foi usado um heap. O TAD usado criava e manipulava um heap para facilitar as operações de remoção e inserção.

2.2 Funções e Procedimentos

```
void entry_swap(entry_t *a, entry_t *b);
```

Troca o conteúdo de dois items.

```
int round_split(FILE *input, int entryMax);
```

Quebra o arquivo em múltiplos pedaços. Ordena eles atraves de um quicksort e finalmente escreve o resultado para um arquivo.

```
void round_write_file(round_t *round, int roundCur);
```

Recebe a rodada ordenada e escreve para um arquivo intermediário.

```
void round_merge(FILE *output, int roundNum);
```

Reabre os subarquivos gerados e intercale eles para gerar as saída final ordenada.

```
void sort_quick(entry_t* entry, int start, int end);
```

Função principal do quicksort. Para casos pequenos usa o Shell sort.

```
int sort_quick_partition(entry_t* entry, int pivot, int start, int end);
```

Função de partição do quick sort para dividir e conquistar.

```
void queue_fix(entry_t *heap, int heapSize, int father);
```

Mantem a ordem do heap.

```
void queue_build(entry_t *heap, int heapSize);
```

A partir de um arranjo desorganizado cria um heap.

```
entry_t queue_pop(entry_t *heap, int *heapSize);
```

Retira e retorna o elemento com maior prioridade da fila.

```
void queue_push(entry_t *heap, int *heapSize, entry_t insert);
```

Insere um elemento mantendo a integridade da fila.

2.3 Programa Principal

O programa principal é divido em trés etapa.

1. Particionar o arquivo de entrada.
2. Ordenar os pedaços.
3. Intercalar os pedaços e produzir a saída.

Para quebrar os pedaços e lido um trecho do arquivo que cabe na memoria. Esse trecho é analisado para descobrir o url e pageviews de cada item.

Esses items são ordenados através do quicksort. Finalmente eles são escritos no subarquivo intermediário.

Para intercalar os pedaços, são reabertos todos subarquivos gerados simultaneamente. É lido e analisado o primeiro item de cada subarquivo para construir a fila de prioridades implementada usando um heap. Ate a fila ficar vazio, são escritos para o arquivo final o item com maior prioridade e adicionado a fila o próximo item do subarquivo de onde o item que acabou de sair foi retirado. No fim os subarquivos são removidos.

2.4 Organização do Código, Decisões de Implementação e Detalhes Técnicos

O código esta dividido em 3 arquivos: exsort.h contem as declarações does TADs e funções, exsort.c as implementações dos TADs e main.c com o programa principal. Foi considerado que o tamanho máximo de um url era 100 caracteres para simplificar a implementação e que a ordenação de URLs no caso de empate respeitava a ordem da tabela ASCII. O compilador usado foi GCC 4.6.020110429 no sistema operacional Arch Linux 2.6.39-ARCH. O programa aceita os parametros ¡arquivo de entrada¿ ¡arquivo de saída¿ ¡numero máximo de items que cabe na memoria¿.

3 Analise de Complexidade

A variável n e definida como o numero de items no arquivo de entrada x como o numero de arquivos gerados dependendo da memoria.

```
void entry_swap(entry_t *a, entry_t *b);
```

Simple troca em $O(1)$.

```
int round_split(FILE *input, int entryMax);
```

Para cada x, Le o pedaço do arquivo em $O(n)$, e usa sort_quick() é $O(nlog(n))$, escreve para um subarquivo em $O(n)$. Logo a complexidade é de $O(xnlog(n))$.

```
void round_write_file(round_t *roundUnsrt, int roundCur);
```

Escreve cada item a um arquivo em $O(n)$.

```
void round_merge(FILE *output, int roundNum);
```

Abre os arquivos em $O(x)$, constrói a file em $O(xlog(x))$ por que o numero de items é limitado pelo numero de arquivos gerados. Para cada item n insere $O(log(n))$ e remove $O(log(n))$ que é $O(nlog(n))$. Como o numero de subarquivos nunca e maior que o numero de items totais a função acaba sendo $O(nlog(n))$.

```
void sort_quick(entry_t* entry, int start, int end);
```

Função recursiva por que chara a partição e duas quicksort's para cada metade da entrada com recorrência de $T(n) = 2T(fracn2) + O(n)$. Resolvendo essa relação $O(nlog(n))$.

```
int sort_quick_partition(entry_t* entry, int pivot, int start, int end);
```

Percorre todos elementos do uma vez logo $O(n)$.

```
void queue_fix(entry_t *heap, int heapSize, int father);
```

Navega um vetor como uma arvore que acaba sendo em $(Olog(n)$.

```
void queue_build(entry_t *heap, int heapSize);
```

Chama queue_heapify() para metade de n logo $O(nlog(n))$.

```
entry_t queue_pop(entry_t *heap, int *heapSize);
```

Chama queue_heapify() uma vez então $O(log(n))$.

```
void queue_push(entry_t *heap, int *heapSize, entry_t insert);
```

Como um queue_heapify() navega como uma a arvore porem de baixo para cima.

3.1 Programa principal

A complexidade da implementacao de ordenacao externa acabou sendo:

$$max(O(xnlog(n)), O(nlog(n))) = O(xnlog(n))$$

x depende diretamente do valor da entrada mais vai ser no máximo igual a n no caso onde o programa gera um arquivo por entidade. O que faz a complexidade ser igual a $O(n^2log(n))$

4 Testes

Para testar a saida do programa foi desenvolvido um script para gerar arquivos com url e pageviews aleatórios usado como entrada. Usando o sort (UNIX) com parâmetros -k2nr -k1 test.txt"os testes gerados eram ordenados. Para verificar se a saída do programa estava certa foi usado o diff do vim. Para determinar o tempo de execução para cada arquivo foi feito outro script que rodada 10 vezes um arquivo de mesmo tamanho e pegava a media dos tempos de execução. A curva parece com a de $n^2log(n)$ deslocada por uma constante.

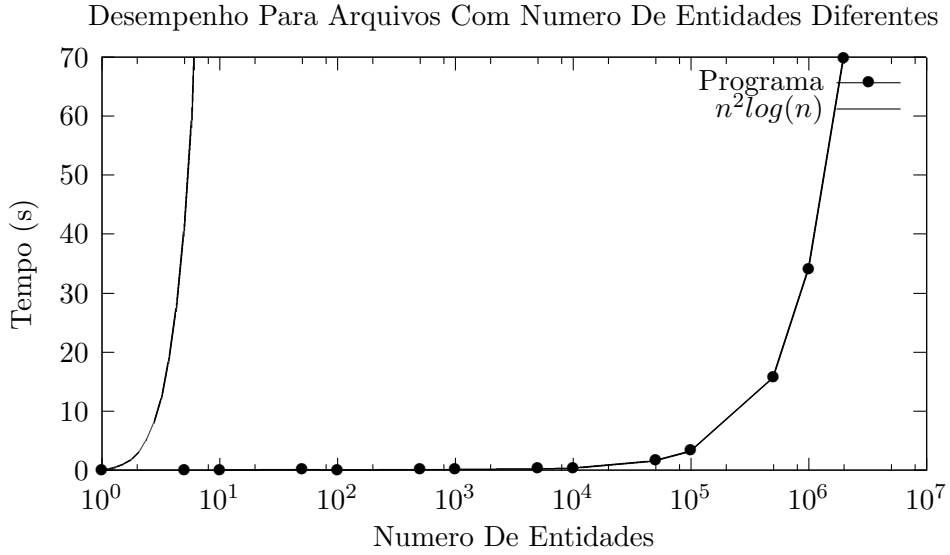

Figura 1: Tempo de execução para vários arquivos

Foram feitos também, testes para determinar o impacto do limite de memoria. O tempo de execução parece estar mais ou menos constate. Minha hipótese por isso e que as operações em disco são as que gastam mais tempo. Como independente do limite de entidades que cabem na memoria terão que ser feitas n leituras ao arquivo o tempo de execução não muda muito.

5 Conclusão

Eu percebi que o maior gargalo na implementação era as operações que envolviam o disco. Eu pensei em algumas maneiras de melhorar essa porcão do código. Pensei em reduzir a quantidade de leitura ao disco no inicio pegando a maior string que coubesse na memoria e processando ele de uma vez ao invés de ler cada item um por um. Como o limite de URL era 100 caracteres assumi que a memoria máxima dada por esse limite vezes o numero de entidades dado na entrada. Uma consequência disso foi que o numero de items em cada subarquivo variaria para preencher o máximo possível. Na hora do quicksort isso ajudaria por que não seria necessário

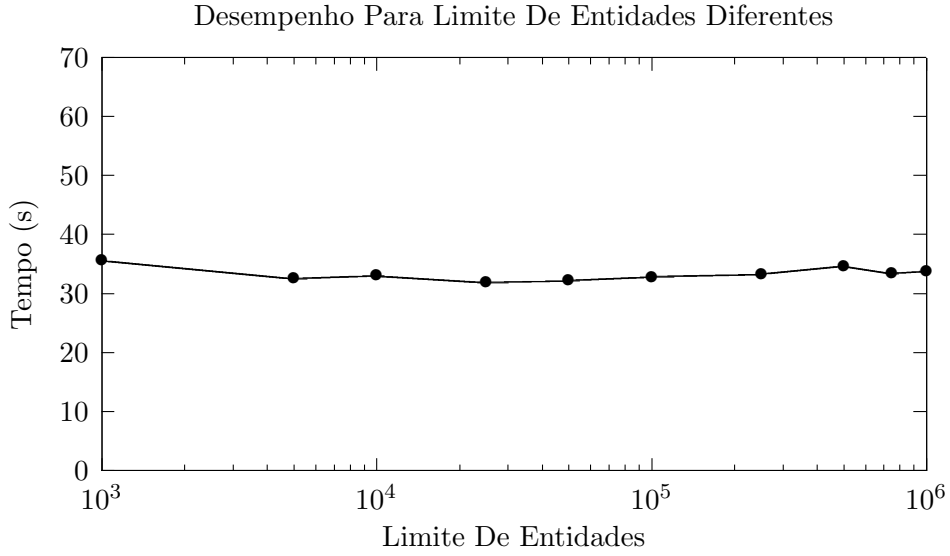

Figura 2: Efeito do limite de entidade

fazer varias leituras de arquivos pequenos e a ordenação poderia acontecer diretamente na memoria reduzindo o acesso ao disco.

Para o quicksort em si, quando ele particiona casos pequenos eu tentei usar o shell sort ao invés de mais chamadas do quicksort que para ter um melhor desempenho em vetores menores, porem em vários testes esse não foi o caso que me levou a retirar esse trecho. Não consegui usar o mesmo método na hora de intercalar pois e necessário ler o topo de cada subarquivo. Neste caso tive que ler cada entrada uma por uma.

6 Referencias

- Cormen T., Leiserson C., Rivest R., Stein C., Introduction to Algorithms 3rd Edition 2009.
- `http://en.wikipedia.org/External_sorting`