Mergesort Externo utilizando Heap Queue

Pedro Probst Minini

ppminini@inf.ufsm.br

1. Introdução

Algoritmos de ordenação interna – quicksort, insertion sort, mergesort, etc. – são úteis quando temos dados que podem ser endereçados diretamente na memória por inteiro. Entretanto, muitos são os casos em que temos de tratar um arquivo muito grande, de tal modo que o conteúdo dele não caiba na memória. Diante disso, temos de utilizar um algoritmo de ordenação externa, que envolve princípios de *dividir e conquistar*. Nestes casos – em especial o método utilizado neste trabalho –, devemos ler o arquivo de entrada em "fatias", armazenando os dados lidos em buffers temporários para que por fim possamos ordená-los e escrevê-los em arquivos de saída. Terminada a leitura do arquivo de entrada, passamos para a etapa de *merge*, que junta os arquivos de saída gerados em apenas um arquivo final com todos os dados tratados.
Neste trabalho, foi aplicado um algoritmo de ordenação externa para ordenar todas as palavras de um arquivo-texto de entrada em ordem alfabética. Como exemplo de entrada, utilizamos o livro *Endymion: A Poetic Romance*, de John Keats. Para testar o programa, insira `python t1.py keats.txt` em algum terminal.

2. Algoritmo

O algoritmo de ordenação externa implementado segue basicamente 3 etapas:

a. Lê o arquivo de entrada em fatias de N bytes e gera os arquivos de saída ordenados;

b. Faz o *merge* dos arquivos de saída num único arquivo ordenado;

c. Deleta os arquivos de saída temporários gerados.

```
# Reúne todos os passos para realizar o mergesort externo
# N: bytes a serem lidos por vez
# nome_arq: nome do arquivo de entrada
def mergesort_externo(N, nome_arq):
    num_arqs = arqs_ordenados(N, nome_arq) # número de arquivos gerados
    merge(num_arqs) # faz o merge dos num_arqs utilizando uma heap queue
    deleta_arquivos("out") # remove os arquivos de saída temporários
    print("Fim!")
```

A função `arqs_ordenados(N, nome_arq)` lê o arquivo de entrada N bytes por vez ("fatia"), inserindo os dados lidos num buffer e ordenando-o com *quicksort*. Com o buffer ordenado, gravamos o seu conteúdo num arquivo de saída *outfile_i.txt*. Esse processo se repete até que o arquivo de entrada seja lido completamente. Para tornar os arquivos de saída mais legíveis, foi optado retirar todos os sinais de pontuação da fatia lida e separar cada palavra por uma nova linha.

```
# Retorna o número de arquivos de saída ordenados gerados pelo arquivo de entrada
def arqs_ordenados(N, nome_arq):
    i = 0 # contador do número de arquivos gerados
    # grava N bytes nos arquivos de saída, continuando até o fim do arquivo de entrada
    with open(nome_arq, 'r') as f_in:
        for fatia in ler_em_fatias(f_in, N):
            with open("outfile_{}.txt".format(i), 'w') as f_out:
                fatia = tira_pontuacao(fatia)
                fatia = fatia.lower().split() # cria lista com as palavras minúsculas
                qcksort(fatia)  # é um tanto lento!
                # fatia.sort() # usa timsort e é mais rápido do que quicksort
                fatia = '\n'.join(string_lst) # lista -> string separada por newline
                f_out.write(fatia) # escreve a string no arquivo de saída
            i += 1

    return i+1 # retorna o número de arquivos de saída gerados
```

Na 2ª etapa, devemos fazer o *merge* dos arquivos de saída ordenados. A função `merge(num_arqs)` primeiramente busca o nome dos arquivos a serem juntados com o auxílio da função `pega_filenames(num_arqs)`. Após isso, os arquivos-objeto são passados como iteráveis para a função `hmerge_manual(*arquivos)`, que junta vários *inputs* ordenados num único *output* (gerador) ordenado. O conteúdo desse iterável ordenado é escrito linha à linha linha ao arquivo *final.txt*, que representa o arquivo final com todas as palavras ordenadas.

```
# Junta os arquivos de saída ordenados em um único arquivo final utilizando
# heap queue
def merge(num_arqs):
    arquivos = pega_filenames(num_arqs)

    with ExitStack() as stack, open('final.txt', 'w') as final:
        arquivos = [stack.enter_context(open(arq)) for arq in arquivos]
        # escreve o conteúdo do iterador linha por linha
        final.writelines(hmerge_manual(*arquivos))
```

A função `hmerge_manual(*arquivos)` é apenas uma versão simplificada da função `heapq.merge(*iterables, key=None, reverse=False)` presente na bilioteca *heapq* (`https://docs.python.org/3.6/library/heapq.html`).

```python
# Versão simplificada de heapq.merge(), que recebe iteráveis ordenados e junta tudo
# num único gerador iterável sobre os valores ordenados
def hmerge_manual(*arquivos):
    h = []

    for itnum, it in enumerate(map(iter, arquivos)):
        # itnum -> número da iteração
        # it -> arquivo
        # next -> chamado por next() para retornar o próximo valor
        # next() -> primeiro elemento de cada arquivo
        try:
            next = it.__next__
            h.append([next(), itnum, next])

        except StopIteration:
            pass

    heapq.heapify(h) # "heapifica" hi

    while True:
        try:
            while True:
                v, itnum, next = s = h[0] # h[0] -> menor valor da heap
                if v[-1] != '\n':
                    yield v + '\n'
                else:
                    yield v # cada yield retorna o menor valor
                s[0] = next() # StopIteration quando exaustada
                heapq.heapreplace(h, s) # restaura a heap com o próximo elem

        except StopIteration:
            heapq.heappop(h) # remove iterador vazio

        except IndexError:
            return
```

A fim de compreendermos o funcionamento de `hmerge_manual(*arquivos)`, primeiro devemos elucidar alguns conceitos. Para juntarmos todos os arquivos temporários em um único arquivo final ordenado, essa função utiliza uma *priority queue* (ou fila de prioridade) elaborada a partir de uma min-heap.

Uma *priority queue* funciona como uma fila normal, mas com um porém: cada valor possui um valor de prioridade relacionado. É esse valor que determina a posição de um elemento da fila ou qual elemento deve ser removido da mesma.

Na escrita da função `hmerge_manual(*arquivos)`, foram utilizadas diversas funções da blibliteca *heapq*:

- **heapq.heappop(heap):** dá "pop"(remove) e retorna o menor item do heap, mantendo o heap invariante. Se

o heap estiver vazio, IndexError será gerado.

- **heapq.heapreplace(heap, item)**: faz um heappop e insere o novo item na heap.
- **heapify(x)**: transforma a lista *x* em uma heap (in-place e em tempo linear).

Primeiramente, fazemos uma lista com o primeiro elemento de cada arquivo, o número da iteração e o método para buscar o próximo elemento. Então, transformamos essa lista *h* em uma min-heap, ou seja, todos os nós pais são maiores do que os nós filhos, e o nó de menor valor é sempre a raiz (*h[0]*). Apos a "heapificação", iteramos sobre a heap adicionando o menor valor dela num gerador iterável, que será o retorno da função (*yield v*). O próximo valor do arquivo então é colocado na heap (mantendo suas propriedades) e *v* é retirado da mesma. O processo se repete até que a heap esteja vazia.
Por fim, usamos o gerador retornado por `hmerge_manual(*arquivos)` e escrevemos cada elemento (linha) no arquivo de saída *final.txt* em `merge(num_arqs)`.

Limitação: apesar do algoritmo descrito acima buscar a eficiência, temos um fator a nos atentar: o número de bytes N que podem ser lidos do arquivo de entrada. Dependendo do tamanho do arquivo de entrada, é necessário procurar por um valor de N que torne suave a execução do programa; caso o arquivo de entrada seja grande demais para um valor de N muito pequeno, corremos o risco de gerar muitos arquivos abertos, de tal modo que a etapa de *merge* acabe sendo prejudicada (*IOError: [Errno 24] Too many open files*). Note que, no caso do arquivo *keats.txt*, caso escolhermos um valor de N < 203, teremos uma quantidade muito grande de arquivos, o que impossibilitará a etapa de *merge*. Ainda, caso escolhermos um valor de N grande demais para um arquivo de entrada também muito grande, é possível que o o hardware não comporte a quantidade de RAM utilizada no processo de ordenação de um buffer temporário.

3. Aplicações

Como já foi mencionado anteriormente, algoritmos de ordenação externa são geralmente utilizados quando possuímos um arquivo de entrada que – por fatores de limitação de hardware – não pode ter seu conteúdo lido por inteiro em apenas uma passada. Atualmente, é comum esse tipo de ordenação ser utilizada para ordenar ou filtrar dados genéticos e grandes arquivos CSV.

Referências

https://en.wikipedia.org/wiki/Merge_algorithm

https://en.wikipedia.org/wiki/External_sorting

https://en.wikipedia.org/wiki/K-way_merge_algorithm

http://code.activestate.com/recipes/491285/

https://stackoverflow.com/questions/36379360/is-there-a-way-to-simplify-this-n-way-me

https://stackoverflow.com/questions/1001569/python-class-to-merge-sorted-files-how-ca

https://docs.python.org/3.6/library/heapq.html
https://hg.python.org/cpython/file/default/Lib/heapq.py#l314
https://www.youtube.com/watch?v=sVGbj1zgvWQ
https://rosettacode.org/wiki/Sorting_algorithms/Quicksort#Python
https://en.wikipedia.org/wiki/Priority_queue