Análise empírica de algoritmos[1][2]

1 Análise empírica

Como discutido em sala, a análise teórica da complexidade de algoritmos pressupõe algumas premissas, como a abstração do modelo de computação e a equivalência de custo computacional de instruções de diferentes níveis de complexidade. A análise empírica da performance de algoritmos visa complementar a análise teórica, respaldando ou refinando as conclusões obtidas através da análise teórica.

Assim como a análise teórica, a análise empírica também apresenta alguns princípios básicos:

1. **Isonomia de implementação** – quando se compara a performance de dois algoritmos, é imprescindível que o contexto da análise seja justo para todos os algoritmos considerados. Assim, é fundamental, por exemplo, que as implementações dos algoritmos tenham sido feitas pelos mesmos desenvolvedores. É imperativo, também, que a execução dos testes seja feita em máquinas de hardware idêntico. Por último, é fundamental evitar a sobrecarga do sistema operacional, como executar mais testes simultaneamente do que a quantidade de núcleos existentes no processador.

2. **Métrica de performance** – o desempenho dos algoritmos pode ser analisado sob diferentes perspectivas, traduzidas analiticamente em diferentes métricas de performance. Tradicionalmente, as métricas mais utilizadas na análise de algoritmos são o tempo de execução ou o consumo de memória. Para identificar o consumo de memória ou o tempo gasto pela execução de um algoritmo, pode-se utilizar ferramentas internas ao código (bibliotecas) ou externas (*profilers*). Quando se opta por bibliotecas, é fundamental mensurar apenas o consumo do algoritmo testado, descartando da análise operações adicionais como entrada e saída de dados, por exemplo.

3. **Benchmark diversificado** – para avaliar o comportamento do algoritmo nos diferentes casos considerados pela análise teórica (melhor, pior e caso médio), é importante considerar diferentes tipos de entrada (*instâncias*). Em geral, cada benchmark de avaliação de desempenho é composto por subconjuntos de instâncias, agrupadas por características em comum consideradas importantes na prática. A análise, então, pode se dar tanto em escala global (o caso médio), como nos casos particulares, enriquecendo o entendimento sobre a performance do algoritmo.

2 A biblioteca `sys/time.h`

Nesta disciplina, usaremos como ferramenta principal de análise a biblioteca `sys/time.h`, oferecida por ambientes Unix para medição de tempo nas linguagens C/C++. Os principais componentes desta biblioteca são[3]:

`struct timeval`, que inclui as variáveis `tv_sec` e `tv_usec`, armazenando respectivamente o tempo em segundos e em microsegundos ($1s = 10^6 \mu s$).

[1] Roteiro de estudo fornecido na disciplina de Estruturas de Dados Básicas 1 (EDB1) da Universidade Federal do Rio Grande do Norte (UFRN).

[2] Autor: Leonardo Bezerra (`leobezerra@imd.ufrn.br`).

[3] `http://pubs.opengroup.org/onlinepubs/7908799/xsh/systime.h.html`

`gettimeofday(struct timeval *, void *)`, que permite registrar o horário atual em uma variável do tipo `struct timeval`.

Para medir o tempo de execução de um procedimento, deve-se registrar o horário antes e depois da execução e calcular a diferença de tempo decorrida:

```
struct timeval inicio, fim;
int duracao;

gettimeofday(&inicio, NULL);
procedimento_a_ser_analisado();
gettimeofday(&fim, NULL);

duracao = (fim.tv_sec - inicio.tv_sec) * 1000000 + (fim.tv_usec - inicio.tv_usec);
```

3 Laboratório

Para aprender como funciona uma análise empírica de um algoritmo, neste laboratório você deverá gerar um gráfico comparativo dos tempos de execução das buscas sequencial e binárias (ambas implementadas de forma iterativa). Para a realização deste laboratório, um material suplementar foi disponibilizado no SIGAA (`lab.zip`).

3.1 Escrevendo o código da aplicação

O arquivo `lab.zip` contém exemplos de código de aplicação para C (`u01.c`) e para C++ (`u01.cpp`). Note que o código está incompleto, uma vez que o vetor v não foi inicializado nem lido, e que a chamada da função está considerando o pior caso para as buscas (elemento não está presente). Sua primeira tarefa é completar o código desta aplicação:

- Como entrada-base para os testes, use o arquivo `entrada.in`. A primeira linha desse arquivo contém a quantidade n de elementos presentes no vetor. As demais n linhas contêm os elementos do vetor.
- O seu código não deverá ler o arquivo `entrada.in`, mas a entrada padrão (`printf` e `scanf` para C, `cin` e `cout` para C++). A razão disso é a flexibilidade de uso, uma vez que ambientes Unix permitem converter um arquivo de entrada em entrada padrão de aplicações usando redirecionamento de fluxo:

  ```
  bin/u01 < entrada.in
  ```

- O código atual registra apenas o tempo para a execução da busca sequencial. Ajuste o código para que ele possa medir, também e de forma independente, o tempo para a execução da busca binária.
- Notem que o resultado da busca está sendo impresso utilizando a saída padrão. O motivo é evitar que flags de otimização removam a chamada da busca, o que aconteceria caso o resultado não fosse utilizado no restante da aplicação.
- A saída do seu código deverá ser escrita na saída de erro (`stderr` do C[4], `cerr` do C++), sendo uma linha contendo três elementos separados por tabulação (`\t`): o tamanho do vetor n, o tempo de duração da busca sequencial e o tempo de duração da busca binária. Lembrem-se de adicionar uma quebra de linha à sua instrução de impressão, uma vez que o sistema operacional só realiza impressões quando o *buffer* de impressão está cheio ou quando encontra uma quebra de linha.

3.2 Efetuando a análise para tamanhos crescentes

O código apresentado acima executa as buscas apenas para o tamanho máximo n. Para realizar a análise empírica, é necessário realizar alguns ajustes:

- Transforme o código da aplicação em um código iterativo. Especificamente, cada iteração deverá medir e imprimir o tempo para a execução das buscas em uma entrada de tamanho i ($i = 1, \ldots, n$). Lembre-se de imprimir o tamanho correto do vetor considerado a cada iteração.

[4] Para usar as variáveis de entrada e saída padrão do C, é possível utilizar as funções `fprintf` e `fscanf`. A diferença dos protótipos dessas funções para os protótipos da funções `printf` e `scanf` é que se torna necessário especificar um ponteiro do tipo `FILE` indicando o dispositivo de entrada/saída. O C/C++ fornecem ponteiros padrão para a entrada (`stdin`), saída (`stdout`) e saída de erro (`stderr`). Por exemplo, para converter a impressão `printf("teste")` na versão que utiliza a saída de erro com a função `fprintf`, a chamada se torna `fprintf(stderr, "teste")`.

- Para gerar o gráfico comparativo, será necessário armazenar a saída da aplicação em um arquivo. Ambientes Unix oferecem a praticidade do redirecionamento de fluxo, sendo que é possível também diferenciar entre a saída padrão (>) e saída de erro (2>). Além disso, é possível descartar uma ou ambas as saídas realizando o redirecionamento para o caminho `/dev/null`.

```
bin/u01 < entrada.in > /dev/null 2> tempo.out
```

- Utilize o *script* plot.R para gerar o gráfico comparativo a partir do arquivo contendo os tempos de execução. Para executar um script R em linha de comando, basta invocar o comando `R CMD BATCH`, como exemplificado abaixo. Note que, para o correto funcionamento do script, você deverá acessar a pasta que o contém, e esta pasta deve conter também o arquivo `tempo.out` gerado pela sua aplicação. Um pdf chamado `buscas.pdf` será criado no caso da execução bem sucedida do script.

```
R CMD BATCH plot.R
```

Uma vez gerado o gráfico escreva uma análise sobre o mesmo. Você será avaliado pela sua capacidade de relacionar as complexidades assintóticas dos algoritmos com seu desempenho prático. Além disso, observações adicionais devem ser feitas.

3.3 Efetuando a análise para o caso médio

Ainda que seja importante entender o desempenho de um algoritmo no pior caso, na prática é fundamental entender seu desempenho no caso médio. A análise teórica deste tipo de caso é bem mais complexa que as análises de melhor e pior caso. No entanto, uma análise empírica bem planejada pode dar indicações importantes sobre o desempenho esperado de um algoritmo.

Para esta análise, transforme o código da aplicação para que o elemento a ser buscado seja escolhido aleatoriamente. Para isto, use as funções de geração de números aleatórios fornecidos pelo C/C++:

`void srand(int),` que define uma semente inicial a partir da qual o algoritmo de geração de números aleatórios será executado. Definir esta semente assegura que execuções repetidas sempre vão obter os mesmo resultados, uma vez que a mesma sequência de números aleatórios será gerada. Neste exercício use `srand(0)` no início do código da sua aplicação.

`int rand(void),` que gera um número inteiro positivo aleatório. Para obter um número no intervalo desejado, é necessário utilizar o operador aritmético módulo: `rand() % (2*n + 2000)`. Note que, para assegurar que números menores e maiores que os elementos presentes no vetor sejam usados, estamos utilizando um intervalo que excede, em cada direção, em 1000 unidades o intervalo dos dados presentes na entrada.

Siga os passos da seção anterior para gerar um novo arquivo `tempo.out` e `buscas.pdf`. Note que o ideal é que você renomeie os arquivos produzidos na seção anterior para que eles não sejam sobrescritos! Uma vez gerado o gráfico, escreva uma análise sobre o mesmo.