UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL

Elton de Souza Vieira
Pedro Arthur Medeiros Fernandes

Análise empírica de algoritmos de busca

Natal/RN, 2016

SUMÁRIO

1. INTRODUÇÃO

Esse relatório foi feito para a disciplina de Estrutura de Dados Básicas 1 para implementar e analisar empíricamente funções de busca em diferentes cenários. Obtendo como resultado o melhor algoritmo para cada situação. Para essa análise, testamos cada uma das funções com 26 massas diferentes de dados, alocando o vetor com tamanho 2^i no qual i varia entre 4 e 30.

Nos testes, foram utilizados três casos diferentes para testar a eficiencia de cada algoritmo sendo eles: o pior caso (onde o elemento não está no vetor), busca pelo elemento a uma distancia de $\frac{3}{4}$ do comprimento do vetor, e a busca da kesima ocorrencia de um elemento (sendo utilizado a 3ª ocorrencia). Por fim, espera-se poder analizar os algoritmos de busca em relação a sua eficiencia e tempo de execução em diversas sitações e compara-los entre si.

2. DESENVOLVIMENTO

2.1. Método

As configurações do notebook utilizado para executar os algoritmos são:

Modelo	Dell Inspiron 5548
Processador	Intel Core i7 – 4510U 2.0GHz
Memória RAM	16Gb DDR3 (1600MHz)
Disco	1Tb HDD
Sistema Operacional	Ubuntu 14.04 LTS x64
Compilador	gcc versão 4.8.4

Lista de Algoritmos Utilizados

Biblioteca do C++:

- std::search
- std::bsearch

Algoritmos desenvolvidos:

- Busca sequencial recursiva

```
template <typename T>
int seq_search_r(T *v, T x, int l, int r) {
    return (l > r) ? -1 :
        *(v+l) == x ? l : seq_search_r(v, x, l+1, r);
}
```

- Busca sequencial interativa

```
template <typename T>
int seq_search_i(T *v, T x, int l, int r) {
    while (l <= r) {
        if (*(v+l) == x)
            return l;
        l++;
    }

    return -1;
}
```

- Busca binária recursiva

```
template <typename T>
int binary_search_r(T *v, T x, int l, int r) {
    int m = (r + l)/2;
    return (l > r) ? -1 :
        *(v+m) > x ? binary_search_r(v, x, l, m-1) :
            *(v+m) < x ? binary_search_r(v, x, m+1, r) : m;
}
```

- Busca binária interativa

```
template <typename T>
int binary_search_i(T *v, T x, int l, int r) {
    while (l <= r) {
        int m = (r + l)/2;
        if (*(v+m) == x)
            return m;
        else if (*(v+m) < x)
            l = m+1;
        else
            r = m-1;
    }
    return -1;
}
```

- Busca ternária recursiva

```
template <typename T>
int ternary_search_r(T *v, T x, int l, int r) {
    int m1 = (r+l+l)/3, m2 = (r+r+l)/3;
    return (l > r) ? -1 :
        x == *(v+m1) ? m1 :
            x == *(v+m2) ? m2 :
                x > *(v+m2) ? ternary_search_r(v, x, m2+1, r) :
                    x < *(v+m1) ? ternary_search_r(v, x, l, m1-1) :
                        ternary_search_r(v, x, m1+1, m2-1);
}
```

- Busca ternária interativa

```
template <typename T>
int ternary_search_i(T *v, T x, int l, int r) {
    while (l <= r) {
        int m1 = (r+l+l)/3, m2 = (r+r+l)/3;
        if (*(v+m1) == x)
            return m1;
        else if (*(v+m2) == x)
            return m2;
        else if (x > *(v+m2))
            l = m2+1;
        else if (x < *(v+m1))
            r = m1-1;
        else
            l = m1+1, r = m2-1;
    }
    return -1;
}
```

Cenários considerados

Foram considerados os cenários em que o vetor está com os elementos fora de ordem (apenas para as funções sequenciais) e quando o vetor está ordenado (utilizando todas as funções). Para cada cenário informado, consideramos também o pior caso (quando o valor pesquisado não está no vetor), o elemento distando em 3/4 do comprimento do vetor (a partir do local de início da busca) e a busca da terceira ocorrência de um valor k no vetor (a partir do início do arranjo).

2.2. Metodologia

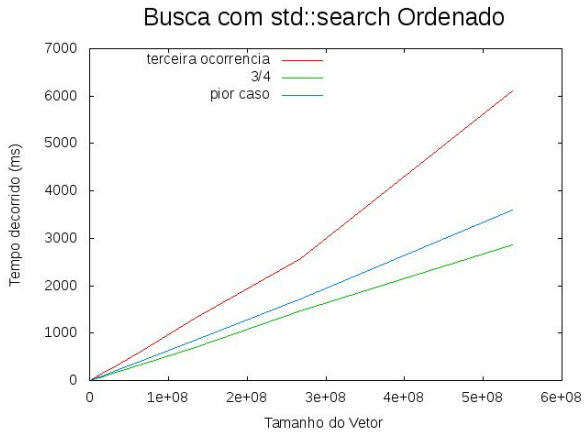
Para os testes, foram utilizados dois programas desenvolvidos em C++, sendo o primeiro composto por todos os algoritmos que podem ser executados com ponteiros alocados dinamicamente e o segundo com a função *std::search*, que utiliza um elemento da classe vector. Em ambos os casos, foi utilizada uma massa de dados (preenchida por números aleatórios entre -9876543210 e 9876543210, representados por um *long int* distribuídos de forma normal (utilizando a função *std::uniform_real_distribution*)). Cada função foi executada utilizando como parâmetro vetores de tamanho 2^i com i variando no intervalo [4, 30], simulando todos os cenários para cada tamanho diferente.

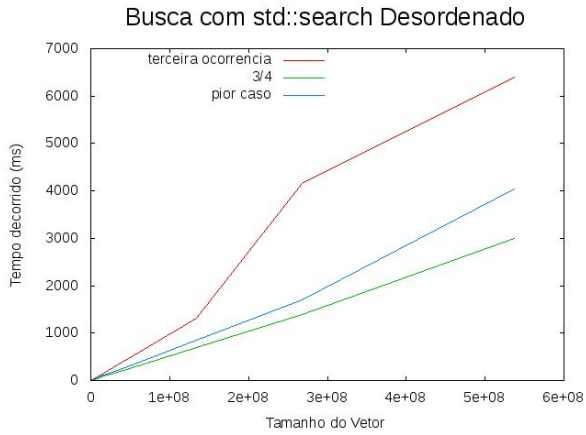
Para cada cenário, foram executadas 100 vezes cada função, medindo o tempo de execução e extraindo a média, utilizando o desvio padrão, com a fórmula $A_k = A_{k-1} + \frac{x_k - A_{k-1}}{k}$ e exportando os resultados para um arquivo de extensão “.dat” que foi lido pelo Gnuplot para gerar os gráficos (com os dados de cada um dos 3 cenários, para efeitos de comparação).

3. RESULTADOS

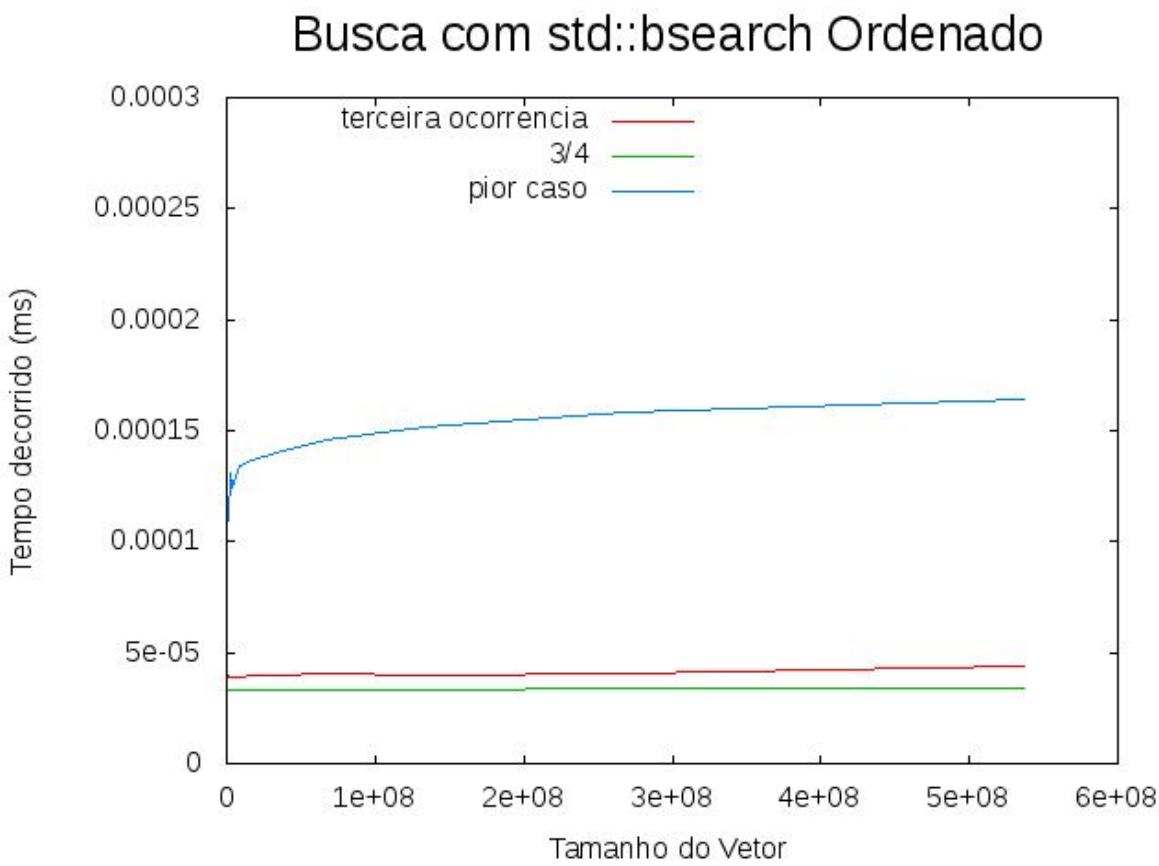
A partir dos dados obtidos pela execução dos algoritmos, foi possivel gerar os seguintes gráficos:

std::search

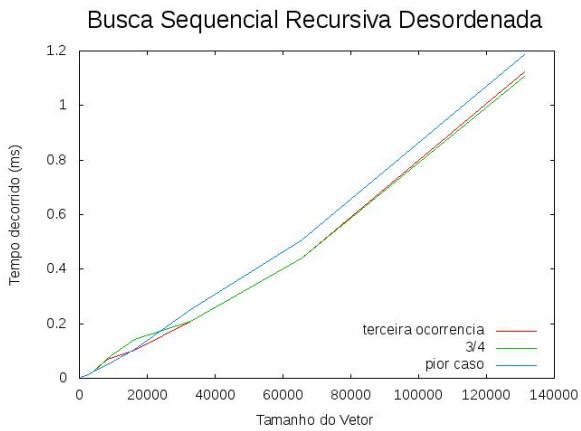


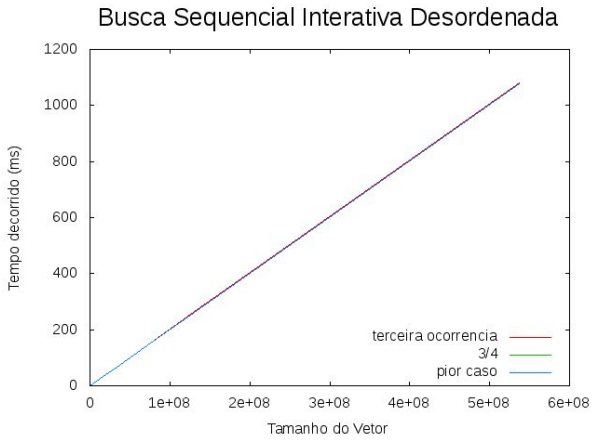


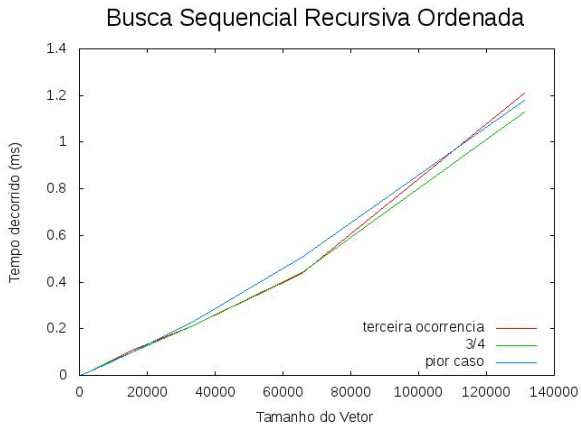
std::bsearch

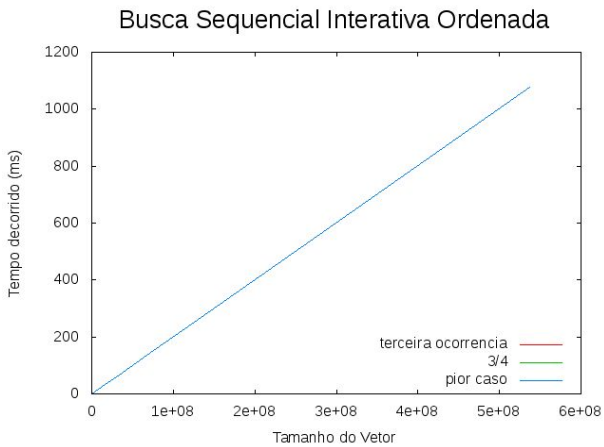


Busca sequencial

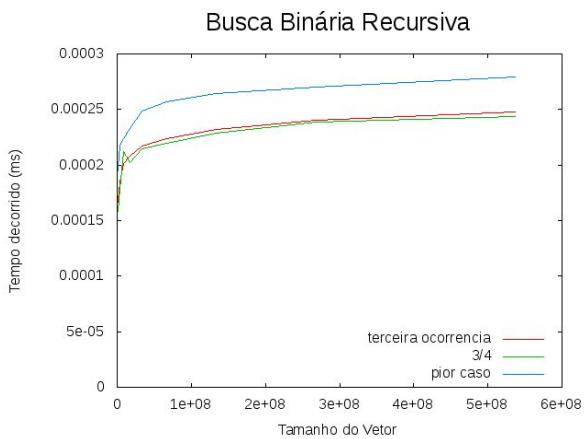


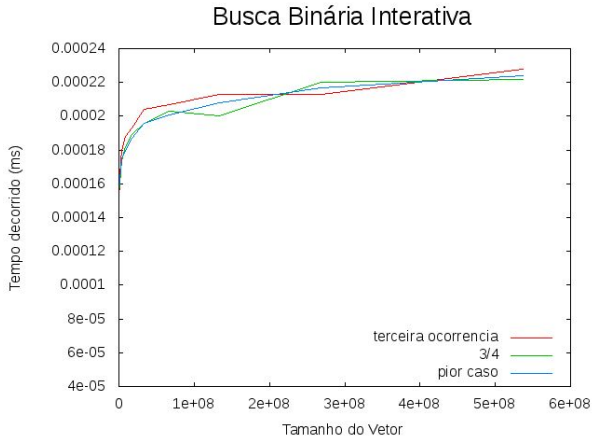




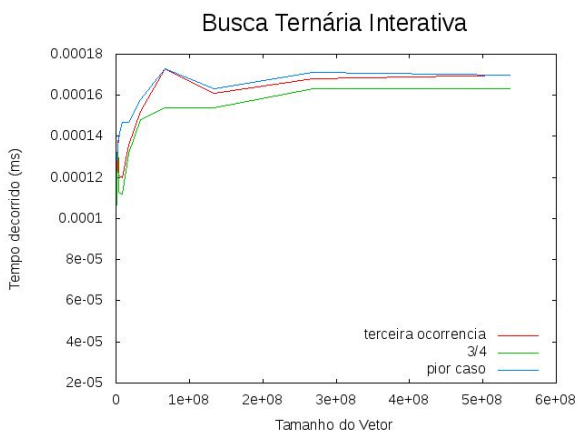


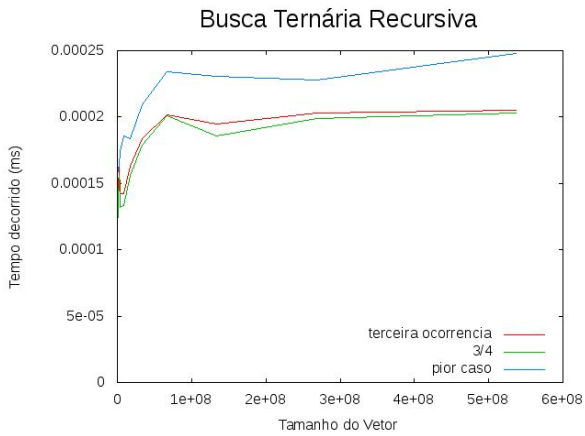
Busca binária





Busca ternária





4. CONCLUSÃO

Com base na análise dos gráficos, pudemos comprovar empiricamente a complexidade de cada função de busca. A saber:

- Busca sequencial: O(n) = n -> linear
- Busca binária: O(n) = $\log_2(n)$ -> logarítimica
- Busca ternária: O(n) = $\log_3(n)$ -> logarítimica

Observando o comportamento de cada algorítmo podemos prever o comportamento dele para qualquer elemento. Portanto, pode-se concluir que o melhor algoritmo a ser utilizado é o da busca binária, pois, apesar de nos gráficos ter perdido em relação ao tempo em comparação a busca ternária, a busca binária é mais eficiente por fazer menos comparações. Isso mostra que dividir o vetor varias vezes para fazer a analise não o torna mais eficiente.

Em relação comparação entre os algoritmos em suas formas interativa e recursiva, percebeu-se melhor desempenho na versão interativa pelo fato de que as verções recursivas empilham as funções para depois desempilhar e assim gerar o resultado, o que demanda mais tempo.