Universidade Federal Rural do Rio de Janeiro

Nicolas Magalhães Silva
Sérgio Felipe Rezende do Nascimento
Yan Carlos de Figueiredo Machado

Relatório Sortlib

AA - Estrutura de Dados I

magalhesnicolas23@gmail.com
sergiorezende2001@gmail.com
yancfm@gmail.com

1 Introdução

O seguinte relatório foi criado para explicar e apresentar os resultados da avaliação da execução dos algoritmos de ordenação de vetores, aplicados na linguagem C. As funções utilizadas nesse projeto foram usadas para criação de uma biblioteca visando a reutilização em outros trabalhos, construindo os códigos para suportar vetores de qualquer tipo, baseado no funcionamento da função 'qsort' da biblioteca 'stdlib.h'.

2 Desenvolvimento

A biblioteca é constituída de 5 funções básicas: quicksort, mergesort, bubblesort, insertionsort e selectionsort, e todos os algoritmos foram desenvolvidos com base no conteúdo apresentado em aula e no livro[1], com pequenas edições. Tais funções são construídas de forma que qualquer tipo de dado possa ser avaliado, necessitando apenas da criação de uma função que seja capaz de comparara-los.
Também existem funções pré-definidas para facilitar o uso dos tipos primitivos com a biblioteca. Sendo elas: compareInt, compareFloat, compareDouble.

2.1 Estrutura das Funções

A estrutura base das funções é descrita da seguinte maneira:

```
1     void foo(void *vector, int n, int size, int (*compare)(void*,  void*))
```

Sendo **vector** o vetor a ser recebido, **n** o número de elementos no vetor, **size** o número de bytes do tipo e **compare** uma função para comparar dois elementos do tipo do vetor.

2.2 Uso da biblioteca

Para utilizar as funções da biblioteca será necessário usar a flag de compilação -lsortlib e incluir 'sortlib.h' no cabeçalho do seu código. Após esse passos, todas as funções estarão disponíveis para uso. É possível ter acesso ao código fonte clicando aqui.
Os algoritmos de ordenação possuem a mesma estrutura de parâmetros, ou seja, todos funcionam da mesma maneira para o usuário. Veja o exemplo abaixo:

```
1  #include <stdlib.h>
2  #include "sortlib.h"
3
4  int main(void) {
5
6      int * vector = calloc(10, sizeof(int)); // Vetor Qualquer
7
8      quicksort(vector, 10, sizeof(vector[0]), compareInt); // Funcao QuickSort
           implementada pelo pacote
9
10     return 0;
11 }
```

3 Testes

Foram realizados testes usando vetores inteiros de tamanho: 100, 1000, 100.000, 1.000.000 e 10.000.000. Além disso foram usado três tipos diferentes de vetores para avaliar com maior precisão o resultado: vetores ordenados, aleatórios e ordenados inversamente.

3.1 Ambiente

Todos os testes foram rodados e testados numa máquina equipada com:
CPU: Ryzen 5 3600
RAM: 16 GB DDR4 3200Mhz
SO: Windows 10 Pro Versão 1909(WSL)
GCC: (Ubuntu 9.3.0-10ubuntu2) 9.3.0

3.2 Desenvolvimento

Os testes foram executados para o tamanho 10.000.000 apenas para o quicksort e o mergesort, os demais foram executados até o tamanho 1.000.000, pois o tempo de execução cresce em uma taxa discrepante dos mais rápidos. Além do mais, cada algoritmo de teste foi executado em paralelo, usando OpenMP, para adiantar o processo.
Os resultados dos testes são exportados para CSV, onde informações úteis como: Método, Tamanho, Tempo e se a ordenação teve sucesso são armazenados.

3.2.1 Avaliação do Tempo de Execução

Para avaliar o tempo de execução de um algoritmo de ordenação foi desenvolvido uma função que consegue avaliar com nanosegundos de precisão, onde os parâmetros necessários são: um número n de instâncias, uma seed para geração dos números, um gerador de números e o método de ordenação.

```
// Funcao para Avaliar o tempo
UnitTest evaluateMethod(int n, int seed, int* (*generator)(int, int), SortMethod
     methodsort) {

    UnitTest unit;
    int * vector;

    struct timespec start, end;

    long double seconds, nanoseconds;
    long double interval;

    {
        // Usa o gerador passado pelo usuario para gerar o vetor
        vector = generator(n, seed);

        clock_gettime(CLOCK_REALTIME, &start); // Pega o tempo inicial

        // Executa a funcao de sort
        methodsort(vector, n, sizeof(vector[0]), compareInt);

        clock_gettime(CLOCK_REALTIME, &end); // Pega o tempo apos execucao

        // Calcula o tempo em segundos e nanosegundos
        seconds = end.tv_sec - start.tv_sec;
        nanoseconds = end.tv_nsec - start.tv_nsec;

        // Calcula o intervalo
        interval = seconds + (nanoseconds * 1e-9);

        // Usando uma struct interna, armazenamos informacoes relevantes
        unit = createUnitTest(vector, 10, sizeof(vector[0]), compareInt,
            is_ordered);

        // Incluindo o tempo ao nosso teste
        evaluateUnitTest(&unit, interval);

        free(vector); // Liberar o vetor alocado
    }

    return unit;  // Retornar o teste unitario
}
```

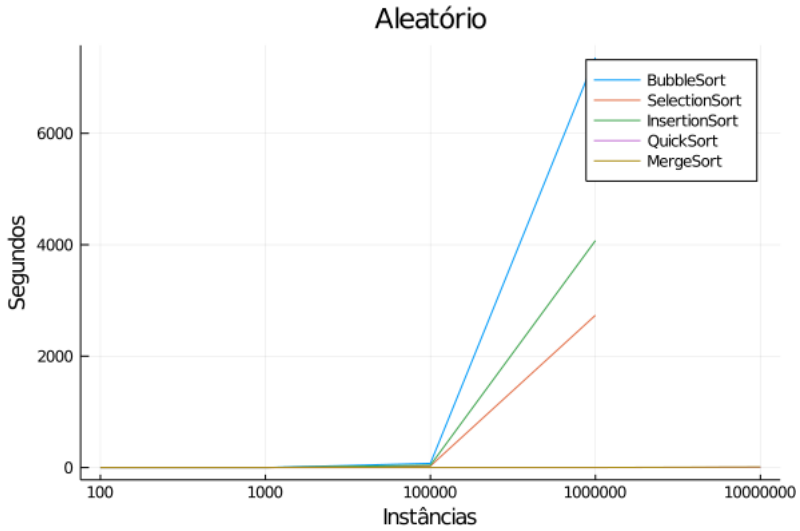
4 Resultados

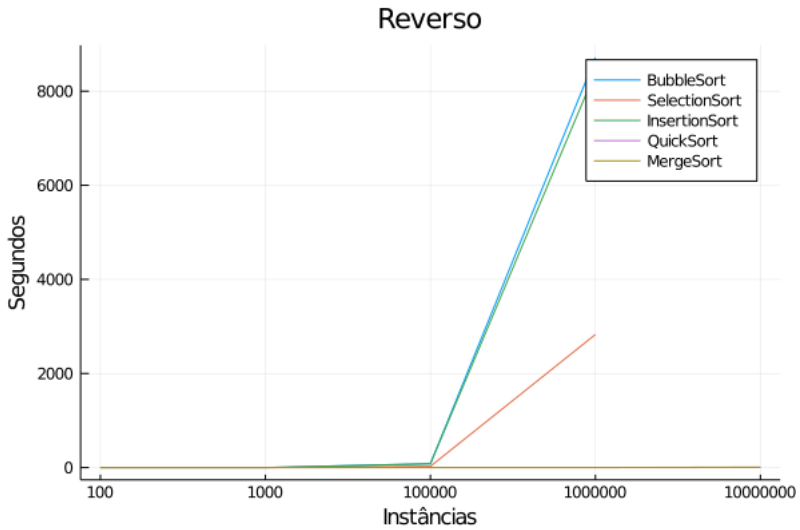
A partir dos resultados dos testes, foi possível gerar tabelas e gráficos mostrando a diferença de rapidez dos algoritmos e a partir disso tirar conclusões que serão mostradas a seguir, sendo as colunas numéricas os números de instâncias.

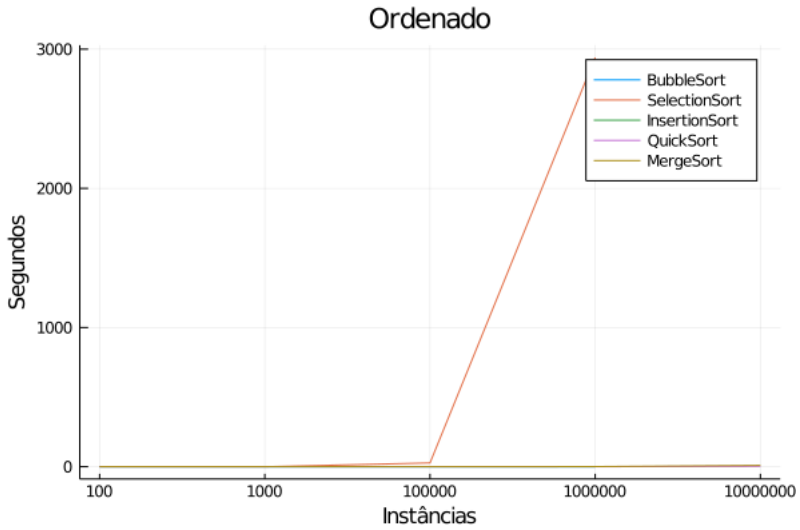
Método	100	1.000	100.000	1.000.000	10.000.000
Ordenado					
mergesort	$2.23e-5s$	$0.00028s$	$0.0584s$	$0.78s$	$9.16s$
bubblesort	$1.73e-6s$	$7.2e-6s$	$0.000669s$	$0.0057s$	
quicksort	$8.13e-6s$	$0.000107s$	$0.0232s$	$0.25s$	$2.74s$
selectionsort	$2.83e-5s$	$0.00277s$	$26.973s$	$2940.37s$	
insertionsort	$1.46e-6s$	$6.633s$	$0.00057s$	$0.0058s$	
Aleatórios					
mergesort	$2.75e-5s$	$0.00036s$	$0.066s$	$0.92s$	$10.38s$
bubblesort	$7.66e-5s$	$0.0073s$	$75.5s$	$7359.66s$	
quicksort	$1.56e-5s$	$0.00017s$	$0.032s$	$0.377s$	$4.57s$
selectionsort	$3.04e-5s$	$0.0026s$	$28.84s$	$2734.27s$	
insertionsort	$5.62e-5s$	$0.0041s$	$42.011s$	$4074.80s$	
Reverso					
mergesort	$2.17e-5s$	$0.00029s$	$0.060s$	$0.70s$	$9.06s$
bubblesort	$8.27e-5s$	$0.0082s$	$82.60s$	$8714.57s$	
quicksort	$1.60e-5s$	$0.00012s$	$0.021s$	$0.26s$	$2.83s$
selectionsort	$2.94e-5s$	$0.0026s$	$26.56s$	$2829.95s$	
insertionsort	$7.83-5s$	$0.0079s$	$80.08s$	$8407.84s$	

(1)

Tabela (1): As células representam o tempo em segundos que o algoritmo apontado pela sua linha levou para ordenar um vetor com o número de instâncias correspondente a sua coluna. Além disso, a tabela é separada entre os 3 tipos de vetores testados, sendo eles, Ordenado, Aleatórios e Reverso.







4.1 Conclusões

A partir da tabela conseguimos perceber a baixa escalabilidade que os algoritmos com complexidade $O(n^2)$ possuem para um número elevado de instâncias, e ao mesmo tempo podemos conferir a eficiência dos algoritmos com O(n log n) com tempos ótimos até para vetores com 10.000.000 de instâncias.

4.1.1 Algoritmos Inviáveis

Olhando mais especificamente para os algoritmos, vemos que o bubblesort tem sempre os piores tempos nos vetores do tipo aleatório e reverso, mas quando se trata do vetor já ordenado faz excelentes tempos, o que não é muito útil em casos reais(o mesmo pode ser dito para o insertionsort, porém com os tempos relativamente melhores que o bubble). O selectionsort por sua vez possui o melhor desempenho em vetores aleatórios e reversos dentre os algoritmos $O(n^2)$ porém possui um péssimo resultado quando o vetor já está ordenado.

4.1.2 Melhores Opções

Como é visível nos dados mostrados, os algoritmos de quicksort e mergesort conseguem um desempenho consideravelmente superior aos demais. O quicksort consegue se manter veloz (4.57 segundos) até com 10.000.000 elementos, e o mergesort não fica muito atrás, tendo tempo máximo de 10.38 segundos.
Vemos que a velocidade do quicksort pode chegar até 18.000x mais veloz para 1.000.000 instâncias quando comparado com o bubblesort e de 2x em relação ao mergesort. Assim, confirmando a enorme eficiência do algoritmo e a sua maior viabilidade de uso em qualquer projeto.

Referências

[1] Renato Cerqueira Waldemar Celes und José Lucas Rangel. *Introdução a Estruturas de Dados*. Editora Campus, 2004.