A ordenação da estratégia de dominação do imperador Estrutura de Dados

Rita Rezende Borges de Lima - 2019021760

[1]Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

ritarezende@ufmg.br

1. Introdução

O problema proposto nesse trabalho consiste em ordenar as civilizações integaláticas a serem conquistadas pelo império por critério de distância da civilização com relação à sede do novo império e o tamanho da população desta. Contudo existe uma surpresa, as funções implementadas para o império devem ser pouco eficientes, de ordem quadrática, e funções eficientes devem ser implementadas para a aliança rebelde!

2. Implementação

O código foi desenvolvido na linguagem *c++* e compilado com *g++ -std=c++11* da GNU Compiler Collection

2.1. Algoritmos Implementados

Foram implementadas quatro algoritmos de ordenação para o trabalho:

- **Bubble Sort**: É o primeiro algoritmo implementado para o império, seu funcionamento é da seguinte forma: comparamos de par em par trocando a posição dos elementos caso o primeiro seja maior que o segundo, caso estes sejam iguais colocaremos antes o que tiver a maior população. É necessário percorrer o vetor diversas vezes para ordena-lo, contudo a cada iteração do loop principal, mais posições do inicio estarão corretas de maneira que percorremos menos posições do vetor reduzindo o critério de parada do loop interno.

- **Selection Sort**: É o segundo algoritmo implementado para o império, seu funcionamento é da seguinte forma: Percorremos o vetor n vezes, sendo n a quantidade de elementos, comparando seus elementos buscando o menor entre eles e trocando este com o da posição n, caso em uma iteração a menor distância seja igual para duas civilizações será colocada na frente do vetor a que tiver a maior população. Ou seja, na primeira iteração percorreremos o vetor inteiro e colocaramos o menor elemento na posição 0, na segunda iteração percorreremos da posição 1 até o final do vetor e pegaremos o menor elemento do conjunto e trocaremos ele de lugar com o elemento da posição 1.

- **Quick Sort**: É o primeiro algoritmo implementado para a aliança rebelde, seu funcionamento é da seguinte forma: Primeiro escolhemos a mediana dos elementos do inicio, meio e fim do vetor para ser o nosso pivô, depois percorreremos nosso vetor em ambos os sentidos com dois iteradores, i e j, comparando esses com o pivô até q j seja menor ou igual a i. Quando encontramos um elemento com index i com distância maior que o pivô ou distância igual e população menor trocamos este de lugar com um elemento de index j que possua distancia menor que a do pivô ou distância igual e população maior. Caso não exista esse i ou j, trocamos com o próprio pivot ou não fazemos nenhuma troca. Depois dividimos esse vetor em dois depois do pivot e escolhemos dois novos pivôs utilizando do menos critério com seus respectivos inicios e fins.

- **Heap Sort**: É o segundo algoritmo implementado para a aliança rebelde, para explicar seu funcionamento primeiro precisamos explicar o que é um heap. Heap é uma estrutura de dados que satisfaz a premissa de que todos os nós filhos tem valor menor que o de seus pais. Em um vetor o nó tem como filhos os elementos que tem a posição do tipo $2n$ e $2n + 1$, sendo n a posição do nó pai. Para transformar um vetor em um heap nosso programa percorre todos os nós filhos e compara estes a seus respectivos pais, caso exista um ou mais filhos de valor maior que o pai este e o maior filho são trocados de lugar, em caso de empate comparamos a população. O heapsort funciona da seguinte maneira: fazemos n iterações, sendo n o tamanho do vetor, onde montamos o heap com os primeiros n elementos do vetor e substituimos a raiz do heap na ultima posição do vetor.

2.2. A classes implementada: Civilização

A classe civilização armazena um vetor de caracteres para seu nome, um inteiro para o tamanho de sua população, e um inteiro para a sua distância a sede do império. Ainda que não explicitamente pedida, é interessante pois além ser uma forma de guardar as informações de maneira mais clara, em oposição a ter por exemplo um vetor para distancias um para populações e um para nomes. Nossa classe também contém os seguintes métodos:

- **O construtor** da classe que cria uma nova instância com os parâmetros distância, população e nome passados.
- **O método *void imprime()*** que imprime na saída padrão o nome a distância e a população no formato:

  Alderaan 55000 89982
  Tatooine 97000 6201
  Mustafar 86100 92516

- **A função *int get distancia()*** que retorna a distância à sede do império.
- **A função *int get populacao()*** que retorna o tamanho da população da civilização
- **A função *char *get nome()*** que retorna o nome da civilização.

2.3. Entrada e Saída do programa

A entrada do programa, *ver tabela 1*, consiste primeiro de um inteiro que significa a quantidade de civilizações que irão ser passadas pelo usuário para o programa, seguido de uma linha para cada civilização com o nome, distância e população nessa ordem. Todos esses dados lidos da entrada padrão (`stdin`)[1].
Após o final da entrada o programa irá retornar essas civilizações ordenadas por distância até o império de forma crescente, *ver tabela 2*, na saída padrão (`stdout`) sendo o critério de desempate qual tem a maior população.

Tabela 1: Exemplo de Entrada

Nome	Distância	População
Corellia	98800	9550000
Mustafar	07150	9380000
Tatooine	66260	9370000
Dagobah	98800	1
Castilon	66260	9200000

Tabela 2: Exemplo de Saída

Nome	Distância	População
Mustafar	07150	9380000
Tatooine	66260	9370000
Castilon	66260	9200000
Corellia	98800	9550000
Dagobah	98800	1

3. Análise de Complexidade

3.1. Analise da Complexidade de Tempo do Bubble Sort

Para compreender a complexidade do algoritmo iremos analisar suas movimentações e comparações. Em seu melhor caso, o vetor já está ordenado logo não ocorrem movimentações. Já em seu pior caso, o programa faz n^2 movimentações. Também ocorrem sempre $\frac{n*(n-1)}{2}$ comparações, logo sua complexidade de tempo é $O(n^2)$.

3.2. Analise da Complexidade de Tempo do Selection Sort

Já no `Selection Sort` sempre acontecem n movimentações, como visto na descrição da sessão 2.1. Como cada elemento é comparado com todos os outros elementos do conjunto, retirando comparações repetidas, $\frac{n*(n-1)}{2}$ comparações ocorrem, logo sua complexidade de tempo é $O(n^2)$.

3.3. Analise da Complexidade de Tempo do Quick Sort

O `Quick Sort` foi a primeira escolha para os algoritmos da aliança rebelde por sua eficiência. Em seu melhor caso e em seu caso médio sua complexidade é da forma de $O(n*log(n))$. O algoritmo possui um pior caso que pode ocasionar em comportamento quadrático, quando o pivô escolhido for o menor ou o maior elemento do vetor. Contudo como a escolha do pivô é feita pela mediana dos elementos do inicio, fim e meio do vetor, a chance é reduzida.

[1] /https://stackoverflow.com/questions/38685724/difference-between-ms-and-s-scanf

3.4. Analise da Complexidade de Tempo do Heap Sort

O `Heap Sort` foi a segunda escolha para os algoritmos da aliança rebelde por sua eficiência. Para o analisarmos, é interessante analisar suas funções. A função *refaz(Civilizacao *civ, int esq, int dir)* tem como pior caso $O(log(n))$, já a função *heapsort(Civilizacao *civ, int numcivilizacoes)* chama a função refaz $n - 1$ vezes. Logo a complexidade total do programa é $O(n * log(n))$.

3.5. Espaço

A complexidade espacial irá depender da quantidade de civilizações, ***n*** , adicionadas pelo usuário. Nenhum dos algoritmos descritos acima utiliza memória auxiliar de maneira que o custo é da forma $O(n)$.

4. Instruções de compilação e execução

Para a compilação de cada programa basta a utilização de seu respectivo *Makefile* presente dentro da pasta *src* de cada algoritmo executando o comando *make* no diretório. Para a execução: *./tp2*.

O algoritmo foi implementado e testado em um ambiente linux Ubuntu com as seguintes características:

Distributor ID: Ubuntu
Description: Ubuntu 18.04.5 LTS
Release: 18.04
Codename: bionic
CPU: Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz

Utilizando o compilador *g++ -std=c++11*.

5. Comparação de tempo de execução dos algoritmos

Para comparamos o tempo necessário para cada um dos algoritmos nos múltiplos testes disponíveis usamos a função `time`[2] em cada um deles.

5.1. Bubble Sort

Index	Tamanho da Entrada	Tempo gasto
0	50	0s
1	100	0s
2	500	0s
3	1000	0s
4	10000	1s
5	100000	98s
6	250000	10min 41s
7	500000	44min 32s
8	1000000	2horas 50min 18s
9	2000000	Não foi possível calcular em tempo hábil

[2]/https://www.geeksforgeeks.org/measure-execution-time-with-high-precision-in-c-c/

5.2. Selection Sort

Index	Tamanho da Entrada	Tempo gasto
0	50	0s
1	100	0s
2	500	0s
3	1000	0s
4	10000	0s
5	100000	66s
6	250000	6min 56s
7	500000	26min 23s
8	1000000	1horas 44min 33s
9	2000000	Não foi possível calcular em tempo hábil

5.3. Heap Sort

Index	Tamanho da Entrada	Tempo gasto
0	50	0s
1	100	0s
2	500	0s
3	1000	0s
4	10000	0s
5	100000	0s
6	250000	1s
7	500000	1s
8	1000000	1s
9	2000000	2s

5.4. Quick Sort

Index	Tamanho da Entrada	Tempo gasto
0	50	0s
1	100	0s
2	500	0s
3	1000	0s
4	10000	0s
5	100000	0s
6	250000	0s
7	500000	0s
8	1000000	0s
9	2000000	1s

Como mostrado pelos dados acima, quando a entrada ultrapassa 100000 elementos a diferença de tempo entre os primeiros dois algoritmos e os dois últimos é perceptível. A partir de 250000 elementos de entrada se torna completamente imprático o uso dos dois primeiros. Logo é possível inferir que O `heap sort` e o `quick sort` são *extremamente* mais eficientes que o `bubble sort` e `selection sort`.

6. Conclusões

Com o desenvolvimento do trabalho foi possível por em prática o conhecimento adquirido na disciplina a respeito de métodos de ordenação e como observado na análise experimental, comprender a diferença de eficiência entre os métodos escolhidos em múltiplos tamanhos de entrada.

7. Bibliografia

Ziviani, N. (2006).Projetos de Algoritmos com Implementações em Java e C++: Capítulo 3: Estruturas de Dados Básicas. Editora Cengage