Trabalho Prático 2

A ordenação da estratégia de dominação do imperador

Fernanda Carolina da Silva Pereira

Universidade Federal de Minas Gerais

Belo Horizonte – MG – Brasil

fernandacsp@ufmg.br

1. Introdução

Dado o sucesso do sistema desenvolvido para gerenciar as naves do imperador Vader para os ataques, ele agora deseja expandir seus domínios, conquistando as civilizações intergalácticas que ainda resistem ao Novo Império.

Frequentemente, Vader recebe informações sobre novas civilizações para conquistar, porém esses dados estão desorganizados. Sendo assim, o objetivo é traçar um plano de dominação obedecendo a dois critérios: distância da civilização com relação ao planeta sede do Novo Império; e o tamanho da população da civilização.

No entanto, descobrimos que somos agentes infiltrados da Aliança Rebelde, e assim teremos a oportunidade de enganar Vader e ajudar a Aliança.

Para realizar esse feito, precisaremos implementar formas mais rápidas de organização dos dados para dominação para a Aliança, enquanto usaremos os mais lentos para o imperador. Sendo assim, a Aliança recebe os planos de dominação primeiro.

2. Implementação

O sistema foi desenvolvido em C++ e compilado com o compilador GNU G++.

3. Estruturas de Dados

Houveram ao todo 4 etapas no sistema, em que duas pertencem a Aliança e duas ao imperador Vader. Em cada etapa foi utilizado uma estrutura de dados que melhor atendesse

à demanda, ou seja, entregasse os dados mais rápido para a Aliança e com atraso para Vader.

Para a Aliança Rebelde foram implementados os métodos QuickSort e MergeSort. E já para Vader, os métodos Seleção e Inserção.

Para armazenamento dos dados sobre as civilizações, foi criado um vector do objeto Civilization. Utilizando essa estrutura não precisamos nos preocupar com alocação estática de memória, e assim alocamos dinamicamente e não há espaço desperdiçado.

3.1. QuickSort

A técnica utilizada no método QuickSort consiste em divisão e conquista.

A ideia básica é dividir o problema de ordenar um conjunto com n itens em dois problemas menores. Os problemas menores são ordenados independentemente, e os resultados são combinados para produzir a solução final.

3.2. MergeSort

Semelhante ao método anterior, o MergeSort também utiliza uma técnica de divisão e conquista, porém, nesse caso baseado em *merging*, ou seja, combina dois vetores ordenados em um vetor maior que também esteja ordenado.

3.3. Seleção

O método de Seleção consiste em selecionar o n-ésimo menor (ou maior) elemento da lista. O próximo passo realizado trocar o n-ésimo menor (ou maior) elemento com a n-ésima posição da lista, de forma que apenas uma troca é realizada por vez.

3.4. Inserção

A ideia do método de inserção é ordenar cada nova posição do array (ou qualquer estrutura que armazene os dados) como se fosse uma nova posição recebida, que é inserida no lugar correto de uma estrutura auxiliar ordenada à esquerda daquela posição.

4. Classes

O sistema contém uma classe representando as civilizações par domínio. Esta classe contém dados como o nome do planeta, a distância do planeta sede do Novo Império e o tamanho de sua população.

5. Organização e fluxo do sistema

Primeiramente, o imperador irá informar quantas civilizações deseja organizar para domínio, e após uma iteração sobre esse número, o sistema recolhe os dados recebidos de cada civilização.

Depois da coleta, um método referente à organização das civilizações é chamado, obedecendo como dito anteriormente aos critérios de prioridade para menores distâncias, e nos casos em que a distância for igual, priorizar as civilizações com maior população.

6. Complexidade de Tempo

Algoritmo	Tempo		
	Melhor	Médio	Pior
Merge sort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

6.2. Complexidade de Espaço

6.2.1. Quick Sort

Possui complexidade de *teta(log n(base 2))* no melhor caso e no caso médio, e *teta(n)* no pior caso.

6.2.2. Merge Sort

Possui complexidade *O(n)* para todos os casos.

6.2.3. Insertion Sort

Possui complexidade *O(1)* para todos os casos.

6.2.4. Selection Sort

Possui complexidade *O(1)* para todos os casos.

7. Comparação de tempo de execução dos algoritmos

Como observamos, existe uma diferença evidente nos métodos escolhidos. Para os métodos mais eficientes, Quick Sort e Merge Sort, notamos a semelhança nos resultados para o

tempo de execução, o que é comprovado pela complexidade de tempo de cada um, explicitada no item anterior.

Já para os métodos menos eficientes, Insertion e Selection, o tempo de execução aumenta com muito mais rapidez em relação às entradas, o que também é explicado pela complexidade de tempo de cada um, explicitada no item anterior.

Além disso, existem também outras várias variáveis, na máquina utilizada por exemplo, que influenciam tais resultados.

Entrada	QuickSort	MergeSort	Insertion	Selection
50	0.000045	0.000123	0.000150	0.000080
100	0.000112	0.000206	0.000191	0.000167
500	0.000430	0.001131	0.001866	0.001806
1000	0.000356	0.003192	0.004737	0.010289
10000	0.006037	0.021198	0.385236	0.577302
100000	0.042996	0.227222	44.197.083	62.889.866
250000	0.114180	0.581866	290.073.725	429.074.035

8. Conclusão

Após os estudos da matéria, foi possível identificar rapidamente as estruturas necessárias para a resolução e entendimento do problema. A contextualização também permitiu o afloramento da criatividade sobre a implementação.

Referências

- Material disponibilizado pela disciplina de Estrutura de Dados;
- https://www.geeksforgeeks.org/;
- https://pt.wikipedia.org/wiki/Wikipédia;
- Ziviani, N. (2006). Projetos de Algoritmos com Implementações em Java e C++.