Trabalho Prático 1

A ordenação da estratégia de dominação do imperador

Gilliard Gabriel Rodrigues
Matrícula: 2019054609

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

gilliard2019@ufmg.br

1. Introdução

O problema proposto foi implementar um plano de dominação de civilizações para o imperador Vader, baseado em dois critérios: distância das civilizações em relação ao quartel general do imperador e tamanho da população.

Segundo as especificações do imperador, dada duas civilizações *C1* e *C2*, com distâncias *D1* e *D2* respectivamente, se $D1 < D2$, *C1* deve ser dominada antes de *C2*. Caso $D1 = D2$, o critério de desempate deve ser pela análise do tamanho da população da civilização. Desse modo, se duas civilizações *C1* e *C2* tiverem a mesma distância, será necessário analisar suas populações *P1* e *P2* respectivamente, e se $P1 > P2$, então *C1* deverá ser dominada primeiro, em caso contrário, *C2* deverá ser dominada antes.

Foi solicitado que quatro versões desse sistema fossem desenvolvidas: duas para o imperador e duas para a Aliança Rebelde, de forma que se as quatro versões forem executadas simultaneamente, a Aliança Rebelde receba o plano de dominação antes do Darth Vader, ou seja, as versões do sistema desenvolvidas para o imperador deveriam ser mais lentas que as desenvolvidas para a Aliança.

Com base nessas especificações, serão apresentadas nesta documentação quatro implementações para o plano de dominação utilizando modificações dos algoritmos de ordenação apresentados em aula.

2. Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

2.1 Classe

A fim de representar uma civilização foi implementada a classe *Civilizacao*, que possui três atributos privados: *nome_civilizacao* (string), *distancia_civilizacao* (int) e *tamanho_populacao* (int). A classe também possui um construtor vazio e três métodos públicos (getters e setters).

2.2 Métodos de ordenação

Como citado na introdução, para desenvolver os planos de dominação foram utilizadas versões modificadas de quatro métodos de ordenação vistos em aula, são eles: o Bubble Sort e o Selection Sort para o plano do imperador, o Quicksort e o Heapsort para o plano

da Aliança Rebelde. Na figura 1 é possível visualizar a implementação do Bubble Sort modificado.

```
void troca(Civilizacao *x, Civilizacao *y){
    Civilizacao *aux = new Civilizacao;
    *aux = *x;
    *x = *y;
    *y = *aux;
    delete aux;
}

void bubble_sort(Civilizacao *c, int n){
    int i, j;
    for(i = 0; i < n - 1; i++)
        for(j = 1; j < n - i; j++){
            if(c[j].GetDistancia() < c[j-1].GetDistancia())
                troca(&c[j-1], &c[j]);
            else if(c[j].GetDistancia() == c[j-1].GetDistancia())
                if(c[j].GetPopulacao() > c[j-1].GetPopulacao())
                    troca(&c[j-1], &c[j]);
        }
}
```

Figura 1. bubbleSortModificado.cpp

Com o objetivo de atender às especificações do plano de dominação, foi adicionado um *else if* dentro do *for* interno que verifica se a distância entre duas civilizações é igual e se essa condição for verdadeira, entra em outro if que verifica se a população da civilização na posição [j] é maior que a população da civilização na posição [j-1] e, se for verdadeiro, troca as duas civilizações de lugar no vetor de civilizações. Dessa forma, o algoritmo ordena as civilizações conforme solicitado pelo imperador.

No caso do segundo plano de dominação para o imperador, foi feita uma mudança parecida no algoritmo do Selection Sort, como podemos ver na figura 2.

```
void troca(Civilizacao *x, Civilizacao *y){
    Civilizacao *aux = new Civilizacao;
    *aux = *x;
    *x = *y;
    *y = *aux;
    delete aux;
}

void selection_sort(Civilizacao *c, int n){
    int i, j, menor;
    for(i = 0; i < n - 1; i++){
        menor = i;
        for(j = i + 1; j < n; j++){
            if(c[j].GetDistancia() < c[menor].GetDistancia())
                menor = j;
            else if(c[j].GetDistancia() == c[menor].GetDistancia())
                if(c[j].GetPopulacao() > c[menor].GetPopulacao())
                    menor = j;
        }
        troca(&c[i], &c[menor]);
    }
}
```

Figura 2. selectionSortModificado.cpp

Como é possível visualizar, a fim de colocar o critério de desempate ao algoritmo para casos em que duas civilizações possuem a mesma distância, foram adicionados um *else if* e um *if* iguais ao do Bubble Sort, com a diferença de que neste caso, ao invés de executar a troca, há a atribuição de *j* à variável *menor*. Dessa forma, o algoritmo ordena as civilizações conforme solicitado pelo imperador.

A seguir, serão apresentados os algoritmos de ordenação utilizados para implementar os planos para a Aliança Rebelde. O primeiro plano utiliza como método de ordenação uma versão modificada do Quicksort visto em aula. Abaixo é possível visualizar sua implementação (figura 3).

```
void troca(Civilizacao *x, Civilizacao *y){
    Civilizacao *aux = new Civilizacao;
    *aux = *x;
    *x = *y;
    *y = *aux;
    delete aux;
}

void particao(int esq, int dir, int *i, int *j, Civilizacao *c){
    Civilizacao x, aux;
    *i = esq;
    *j = dir;
    x = c[(*i + *j)/2]; //Elemento central como pivô;
    do{
        while(c[*i].GetDistancia() < x.GetDistancia() || (c[*i].GetDistancia() ==
x.GetDistancia() && c[*i].GetPopulacao() > x.GetPopulacao()))
            (*i)++;
        while(c[*j].GetDistancia() > x.GetDistancia() || (c[*j].GetDistancia() ==
x.GetDistancia() && c[*j].GetPopulacao() < x.GetPopulacao()))
            (*j)--;
        if(*i <= *j){
            troca(&c[*i], &c[*j]);
            (*i)++;
            (*j)--;
        }
    } while(*i <= *j);
}

void ordena(int esq, int dir, Civilizacao *c){
    int i, j;
    particao(esq, dir, &i, &j, c);
    if(esq < j)
        ordena(esq, j, c);
    if(i < dir)
        ordena(i, dir, c);
}

void quick_sort(Civilizacao *c, int n){
    ordena(0, n - 1, c);
}
```

Figura 3. quickSortModificado.cpp

Dessa vez, com o propósito de incrementar o critério de desempate, a modificação foi feita na condição do *while* que incrementa o *i* e no *while* que incrementa o *j*, no primeiro, além de verificar se a distância do elemento na posição [*i] é menor que a do pivô (*x*), temos uma outra condição possível: a distância ser igual e a população de *c[*i]* ser maior. No caso do *while* que incrementa o *j*, a condição nova é o contrário: ter a distância igual, porém a população menor. Desse modo, o algoritmo ordena as civilizações conforme solicitado pelo imperador.

Para visualizar melhor, eis um exemplo: seja *c[*i]* maior que o pivô e *c[*j]* uma civilização com distância igual mas população maior que a do *x*, nesse caso o programa para de iterar o *i* e *j* e caso eles ainda não tiverem se cruzado, efetua a troca de *c[*i]*

com *c[*j]*, após isso, teremos o antigo *c[*j]*, que possuía a mesma distância que o pivô e população maior no seu devido lugar: antes do pivô (critério de desempate).

Por fim, no segundo plano para a Aliança Rebelde, foi utilizado uma versão modificada do algoritmo de ordenação Heapsort. Para incrementar o critério de desempate foi adicionado um else if e um if no método que constrói o *heap* a fim de que o heap seja montado levando em consideração o tamanho da população nos casos em que a distância de *c[esq]* for igual à distância de *c[maior]* ou a distância de *c[dir]* for igual à de *c[maior]*. Na figura 4 é possível visualizar a implementação do Heapsort modificado.

```
void troca(Civilizacao *x, Civilizacao *y){
    Civilizacao *aux = new Civilizacao;
    *aux = *x;
    *x = *y;
    *y = *aux;
    delete aux;
}

void heap(Civilizacao *c, int n, int i) {
    int maior = i;
    int esq = 2 * i + 1;
    int dir = 2 * i + 2;

    if(esq < n && c[esq].GetDistancia() > c[maior].GetDistancia()) {
        maior = esq;
    } else if(esq < n && c[esq].GetDistancia() == c[maior].GetDistancia()) {
        if(c[esq].GetPopulacao() < c[maior].GetPopulacao()) {
            maior = esq;
        }
    }

    if(dir < n && c[dir].GetDistancia() > c[maior].GetDistancia())
        maior = dir;
    else if(dir < n && c[dir].GetDistancia() == c[maior].GetDistancia())
        if(c[dir].GetPopulacao() < c[maior].GetPopulacao())
            maior = dir;

    if( maior != i){
        troca(&c[i], &c[maior]);
        heap(c, n, maior);
    }
}

void heap_sort(Civilizacao *c, int n){
    // Construindo o heap:
    for(int i = n/2 - 1; i>=0; i--)
        heap(c, n, i);

    for(int i = n -1; i >= 0; i--){
        troca(&c[0], &c[i]);
        heap(c, i, 0);
    }
}
```

Figura 4. heapSortModificado.cpp

A fim de deixar os códigos bem modularizados, os métodos de ordenação foram divididos em arquivos *.h* e *.cpp*.

2.3 Entradas e saídas

Conforme especificado pelo novo imperador, a entrada e a saída são padrões do sistema (stdin e stdout, respectivamente). A primeira linha da entrada é composta por um valor inteiro **N** ($0 < \mathbf{N} \leq 2000000$), que indica o total de civilizações a serem dominadas. A seguir, haverão **N** linhas compostas por três informações, que caracterizam as civilizações: uma string, representando o nome da civilização, um inteiro, representando a distância da civilização em relação ao quartel general do imperador e outro inteiro, representando o tamanho da população dessa civilização. As informações são separadas por espaço e as entradas são armazenadas em um vetor do tipo *Civilizacao*, que foi chamado de "conjunto_de_civilizacoes" em todos os arquivos main. O final da entrada é indicado por *EOF* (Ctrl + D no terminal).

A saída é representada pelas civilizações ordenadas de acordo com os critérios especificados pelo imperador. Os arquivos main são bem simples: basicamente há um for para a leitura das informações, armazenando-as no vetor *conjunto_de_civilizacoes*, em seguida há a chamada do método de ordenação e, por último, um for que imprime as civilizações ordenadas. Para alocar memória dinamicamente foi utilizada a função *malloc*, com o cuidado de liberar a memória após o uso.

2.4 Instruções de compilação e execução

Para compilar o trabalho, acesse o diretório do plano de dominação que deseja executar via terminal, e digite:

make

Em seguida, para executar o código, digite:

./tp2

Logo após, insira o número de civilizações, em seguida, insira as informações de cada civilização (nome, distância e tamanho da população, separados por espaço). Por fim é só aguardar as civilizações serem ordenadas e impressas no terminal.

3. Análise de complexidade

3.1 Tempo

Segue abaixo as análises de complexidade de tempo de cada plano de dominação. O trecho de código do main que domina assintoticamente é a chamada do método em todos os planos de dominação. Logo, serão analisadas as complexidades dos métodos de ordenação chamados.

3.1.1 Plano 1 do imperador

O método de ordenação chamado aqui é o *bubble_sort.* Como visto em aula, se esse algoritmo for implementado com uma melhoria (parar quando não forem efetuadas nenhuma troca numa passagem), seu melhor caso será *O(n)*, mas como nesse caso a melhoria não foi implementada, em termos de comparações, a complexidade do algoritmo é sempre $O(n^2)$. Portanto, o plano 1 de dominação do imperador tem custo $O(n^2)$.

3.1.2 Plano 2 do imperador

O método de ordenação chamado aqui é o *selection_sort.* Como visto em aula, o custo de complexidade de tempo desse algoritmo é $O(n^2)$ no pior caso, no melhor caso e no caso médio. Portanto, o plano 2 de dominação do imperador tem custo $O(n^2)$.

3.1.3 Plano 1 da Aliança Rebelde

O método de ordenação chamado aqui é o *quick_sort.* Como visto em aula, o melhor caso desse algoritmo ocorre quando cada partição divide o conjunto em duas partes iguais, sendo $O(n \log n)$. O pior caso ocorre quando, sistematicamente, o pivô é escolhido como sendo um dos extremos de um arquivo, sendo $O(n^2)$. O caso médio é $O(n \log n)$.

3.1.4 Plano 2 da Aliança Rebelde

O método de ordenação chamado aqui é o *heap_sort.* Como visto em aula, o comportamento desse método é sempre $O(n \log n)$, independentemente da entrada

3.2 Espaço

Quanto ao espaço, no caso dos algoritmos implementados para a Aliança Rebelde, a ordem de complexidade é O(n), pois os métodos de ordenação utilizam memória auxiliar. Já no caso dos algoritmos implementados para o imperador, a ordem de complexidade é O(1) independentemente da entrada, não há uso de memória auxiliar.

4. Comparação do tempo de execução dos algoritmos

A tabela da figura 5 mostra os tempos de execução dos quatro algoritmos para diferentes tamanhos de entrada (testes 0 ao 6).

Entrada	Bolha	Seleção	QuickSort	HeapSort
50	0,000000	0,000000	0,000000	0,000000
100	0,000000	0,000000	0,000000	0,000000
500	0,015625	0,000000	0,000000	0,000000
1000	0,062500	0,000000	0,000000	0,000000
10000	7,250000	0,390625	0,015625	0,015625
100000	896,953125	51,281250	0,156250	0,468750
250000	5655,187500	348,796875	0,437500	1,328125

Figura 5. Tempos de execução.

Com base nesses dados, foram plotados três gráficos para facilitar a visualização e a comparação dos tempos de execução em segundos de cada algoritmo utilizado (figuras 6, 7 e 8).

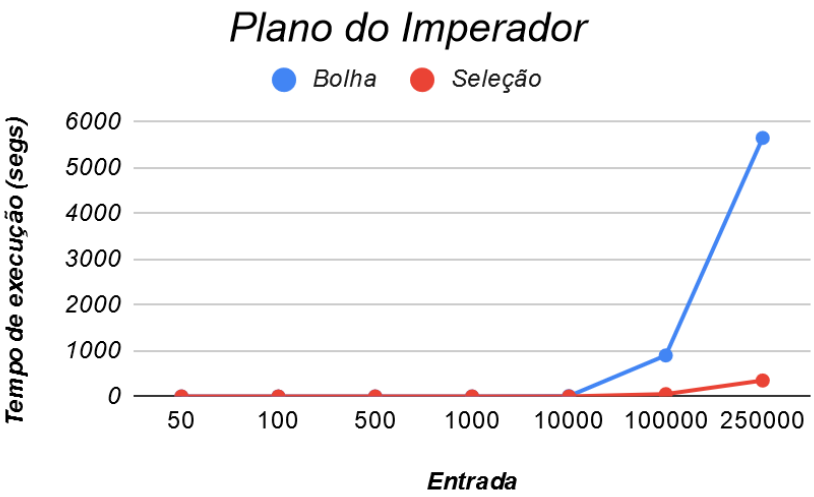

Figura 6. Gráfico Bubble Sort x Selection Sort.

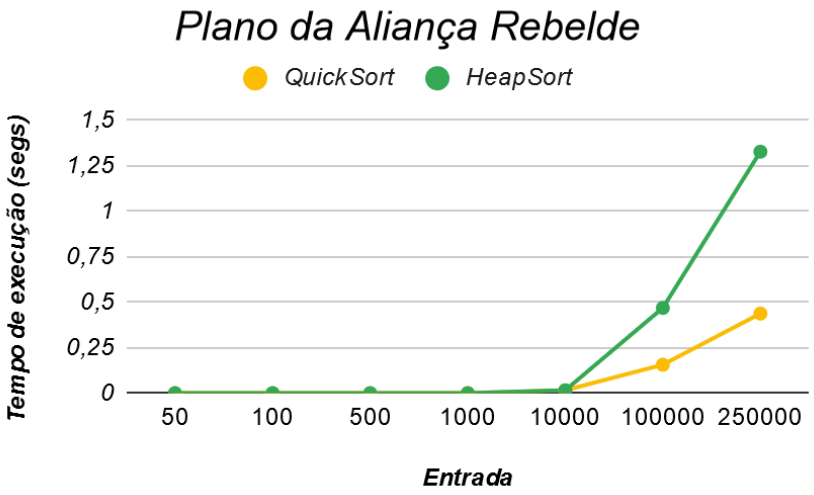

Figura 7. Gráfico Quicksort x Heapsort.

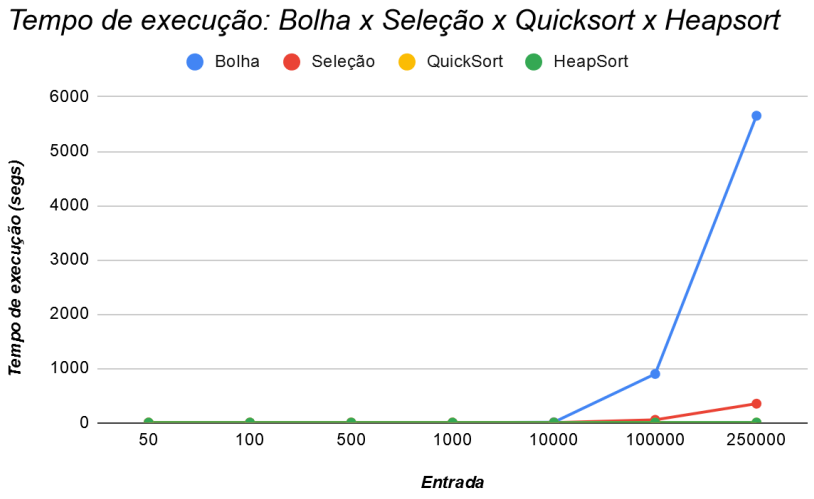

Figura 8. Gráfico entrada x tempo de execução dos 4 algoritmos.

Com base nos gráficos apresentados, é possível notar que a partir de 10000 entradas, fica nítida a diferença que a complexidade de tempo dos algoritmos de ordenação faz no tempo de execução dos planos de dominação de civilizações. O uso dos métodos simples para a implementação dos algoritmos a serem utilizados pelo imperador Vader sem dúvidas farão com que ele receba a resposta bem mais tarde em relação à Aliança Rebelde, que terá acesso aos algoritmos eficientes e por isso receberá os planos de dominação primeiro.

5. Conclusão

O ponto mais importante para conseguir implementar os planos de dominação foi sem dúvidas definir quais métodos de ordenação utilizar para o imperador Darth Vader e quais utilizar para a Aliança Rebelde, levando em consideração que a Aliança precisava receber os planos de dominação com antecedência em relação ao imperador.

O próximo passo foi implementar os métodos com as modificações necessárias para atender às especificações do imperador (ordenar pela distância da civilização em relação ao quartel general e utilizar como critério de desempate o tamanho da população). Outra parte crucial foi a medição do tempo de execução dos algoritmos, sem ela não seria possível plotar os gráficos apresentados na seção 4.

No decorrer desse trabalho, ficou evidente a importância de fazer testes para ter certeza de que o algoritmo estava ordenando de forma correta após fazer as modificações. Por fim, foi dada uma atenção especial ao processo de refatoração e modularização do código para atender às boas práticas de programação e facilitar a identificação e correção de erros.

Após esse trabalho, ficou clara a diferença em termos de eficiência dos métodos linear logarítmicos em relação aos métodos quadráticos, complementando o aprendizado adquirido durante as aulas sobre métodos de ordenação.

Referências

CHAIMOWICZ, Luiz e PRATES, Raquel. Slides da disciplina Estrutura de Dados. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. 2020.

HEAP Sort Algorithm. Programiz. Disponível em: <https://www.programiz.com/dsa/heap-sort>. Acesso em: 10 de outubro de 2020.