Universidade Federal de Viçosa - Campus Florestal
CCF251 – Algoritmos e Estrutura de Dados I
2018/2
Profa. Thais R. M. Braga Silva

Nomes: Josué Nunes Campos – 03465
Lucas Barros Pereira Costa – 03511
Mateus Coelho Santos – 03488

RELATÓRIO FINAL

- **Objetivo**

O programa implementado tem como objetivo ordenar um vetor, de seis formas diferentes, que possui uma matriz de voos em cada uma das suas posições, sendo que, o número de posições do vetor, a quantidade de matrizes preenchidas e a quantidade de voos nas matrizes pode variar de acordo com o cenário.

O intuito do trabalho por completo é estudar os 6 métodos de ordenação abordados nas aulas, ao ver um mesmo vetor ser ordenado por diferentes métodos, e as diferenças de tempo, movimentações e comparações dos algoritmos.

- **Da Implementação**

Demos início ao processo de implementação do trabalho criando um vetor capaz de possuir uma matriz e um identificador dessa matriz em cada uma das suas posições, para isso, Criamos um TAD que abriga a estrutura de dados desse vetor, como também suas funções essenciais e os algoritmos de ordenação estudados.

Feito isso, começamos o processo de criar a lógica para geração aleatória dos identificadores das matrizes e dos voos do modo iterativo, de modo que no final, restaria apenas ordenar o vetor preenchido de forma completa. Já na parte de ordenação, nos deparamos com a questão de haver a possibilidade do usuário querer ordenar o mesmo vetor com algoritmos diferentes, para isso, decidimos criar dois vetores, um vetor fixo que permanece sempre desordenado e um vetor auxiliar, sendo que este é ordenado pelos algoritmos, para que no

final da ordenação de cada algoritmo, atribuímos a esse vetor auxiliar, o vetor desordenado novamente, resolvendo o problema.

Abaixo mostramos o trecho do código que realiza tais soluções:

```c
#include "TADMATRIZVOOS.h"//Inclusão do TAD da Matriz de Voos

typedef struct{//Estrutura de um elemento do vetor de matrizes
  TipoMatriz Matriz;//Matriz de voos
  int ID;//Identificador da matriz
}TipoVetor;

void InicializaVetor(TipoVetor *Vetor, int TAM);//Função para inicializar o vetor
void geraVoos(TipoVoo *Voo);//Função para gerar os voos aleatórios do modo iterativo
void bubble_sort(TipoVetor *Vetor, int TAM);//Algoritmo bolha
void selection_sort(TipoVetor *num, int tam);//Algoritmo de Seleção
void insertion_sort(TipoVetor *vetor, int tamanhoVetor);//Algoritmo de Inserção
void shell_sort(TipoVetor *vetor, int n);//Algoritmo Shell Sort
void Particao(int Esq, int Dir, int *i, int *j, TipoVetor *vetor, int *Comp, int *Mov);//Função Partição do Quick Sort
void Ordena(int Esq, int Dir, TipoVetor *vetor, int *Comp, int *Mov);//Função Ordena do Quick Sort
void quick_sort(TipoVetor *vetor, int n);//Algoritmo Quick Sort
void troca(TipoVetor *a, TipoVetor *b);//Função que troca as chaves do vetor
void heapify(TipoVetor *vetor, int n, int i, int *Mov, int *Comp);//Função que constrói e refaz o heap
void heap_sort(TipoVetor *vetor, int n);//Algoritmo Heap Sort
```

TAD que possuí as funções necessárias para realização do trabalho.

```c
  Vetor = (TipoVetor *)malloc(TAM*sizeof(TipoVetor));//Criando o vetor dinamicamente alocado
  Vetor2 = (TipoVetor *)malloc(TAM*sizeof(TipoVetor));//Criando o vetor auxiliar dinamicamente alocado
  InicializaVetor(Vetor, TAM);//Inicializando o Vetor de matrizes
  for(i=0; i<TAM_div; i++){
    for(j=0; j<In; j++){
      geraVoos(&Voo);//Gerando os Voos aleatoriamente
      InserirVoo(&Vetor[i].Matriz, Voo);//Inserindo o voo na matriz correspondente à posição do vetor
    }
    Vetor2[i] = Vetor[i];//Atribuíndo ao segundo vetor os dados do vetor principal
  }
```

Trechos do códigos que mostram a relação entre o vetor principal e do vetor

```c
for(i=0; i<TAM; i++){
  Vetor2[i] = Vetor[i];//Desordena o vetor auxiliar para ser usado novamente
}
```

Lógica que ocorre ao final do uso de cada algoritmo de ordenação.

Por fim, as manipulações com o vetor de matrizes ocorreu de forma esperada, mas, vale ressaltar que, obtivemos problemas ao colocarmos estruturas de repetições para executar vários cenários em uma mesma compilação, pois a partir do momento que a máquina atingisse seu limite de memória, o programa parava de funcionar. Então, para resolver isso, decidimos deixar a possibilidade do usuário escolher apenas vários algoritmos para ordenar o mesmo vetor várias vezes com algoritmos diferentes.

- **Heap Sort**

Um grande impasse que enfrentamos durante o teste com os algoritmos foi codificar o algoritmo heap sort de maneira que ele se encaixasse no vetor criado de posições com índices indo de 0 até n-1, visto que em sala de aula aprendemos o heap sort ordenando um vetor que tem posições com índices indo de 1 até n. Depois de várias tentativas de reconstruir o código do algoritmo para que o vetor de matrizes fosse ordenado por ele, partimos para pesquisas na internet a fim de buscar uma implementação do heap que ordenasse um vetor começando da posição de índice 0.

Como resultado, obtivemos uma implementação do algoritmo que segue fielmente o conceito de árvores, assunto da disciplina AEDS II, mas que atuou perfeitamente no nosso trabalho. A implementação original estava em C++, portanto, fizemos as alterações necessárias para que ele se encaixasse no trabalho proposto.

A diferença entre a implementação do heap sort aprendido em sala de aula para esse é que, além de ordenar vetores indo de 0 até n-1, na implementação aprendida em sala, o algoritmo de ordenação necessitava de dois métodos para deixar o vetor na condição de um heap, que são o Constrói e o Refaz. Já nessa implementação que conseguimos, o algoritmo de ordenação necessita de apenas um método que realiza as funções que o Constrói e o Refaz desempenham separadamente, para tanto, esse método é recursivo.

Exibimos tanto o algoritmo original, quanto o mesmo modificado para ser utilizado no trabalho:

```
void heapify(int arr[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int l = 2*i + 1; // left = 2*i + 1
    int r = 2*i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i)
    {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// main function to do heap sort
void heapSort(int arr[], int n)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i=n-1; i>=0; i--)
    {
        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

Fonte: *https://www.geeksforgeeks.org/heap-sort/*

```c
void heapify(TipoVetor *vetor, int n, int i, int *Mov, int *Comp){
  int Maior = i;
  int Esq = 2*i + 1;
  int Dir = 2*i + 2;
  (*Comp) ++;
  if((Esq < n) && (vetor[Esq].ID > vetor[Maior].ID)){
    Maior = Esq;
  }
  (*Comp) ++;
  if((Dir < n) && (vetor[Dir].ID > vetor[Maior].ID)){
    Maior = Dir;
  }
  if(Maior != i){
    troca(&vetor[i], &vetor[Maior]);
    (*Mov) += 3;
    heapify(vetor, n, Maior, Mov, Comp);
  }
}
```

Método que realiza as funções do Constrói e do Refaz.

```c
void heap_sort(TipoVetor *vetor, int n){
  int i, Mov = 0, Comp = 0;
  clock_t TempFinal, TempInicial;
  double Tempo_ms;
  TempInicial = clock();
  for(i = n / 2 - 1; i >= 0; i--){
    heapify(vetor, n, i, &Mov, &Comp);
  }
  for(i=n-1; i>=0; i--){
    troca(&vetor[0], &vetor[i]);
    Mov += 3;
    heapify(vetor, i, 0, &Mov, &Comp);
  }
  TempFinal = clock();
  Tempo_ms = (TempFinal - TempInicial) * 1000.0 / CLOCKS_PER_SEC;
  printf("Algoritmo: Heap Sort\n");
  printf("Comparacoes: %d\nMovimentacoes: %d\nTempo gasto: %lf ms\n", Comp, Mov, Tempo_ms);
  printf("--------------------------------------------------------------------------------\n\n");
}
```

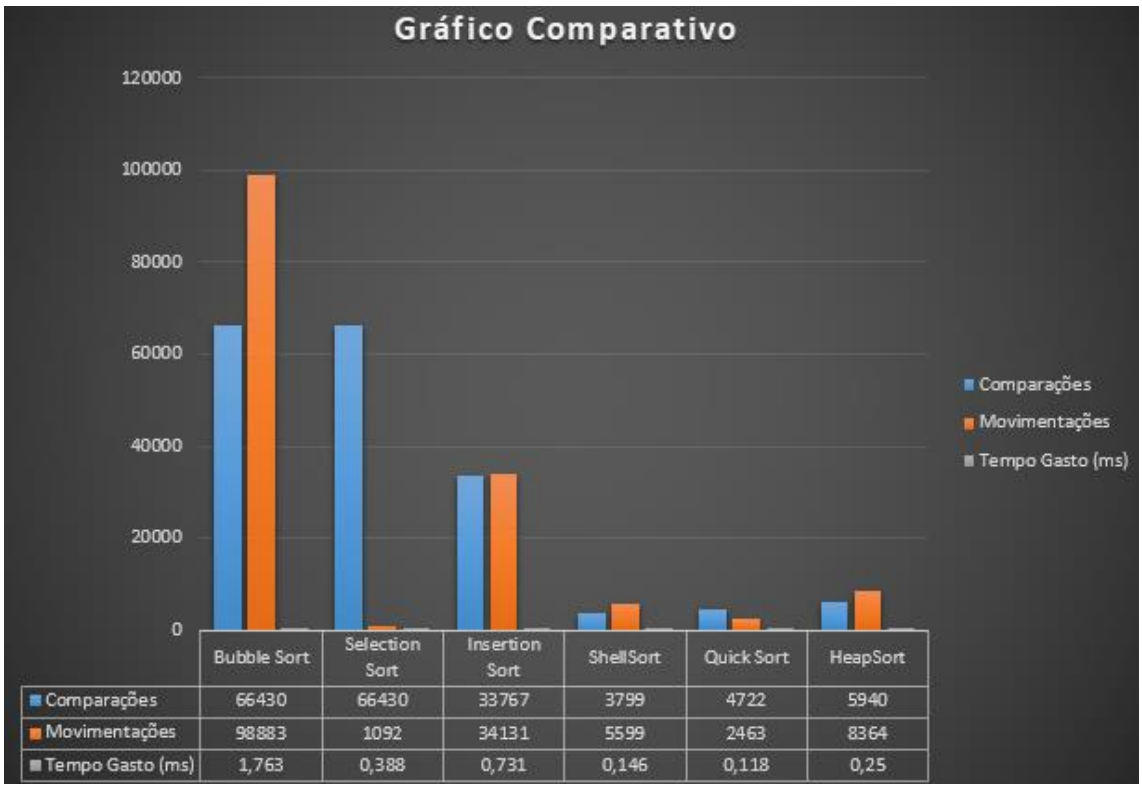
Método principal do algoritmo com as alterações necessárias realizadas.

- **Gráficos Comparativos de Cada Cenário**

Para terminarmos a análise do trabalho, estabelecemos abaixo os gráficos de cada cenário (Comparações, movimentações, tempo gasto), para os 6 algoritmos de ordenação aprendidos. Identificamos também as configurações da máquina utilizada nos testes.

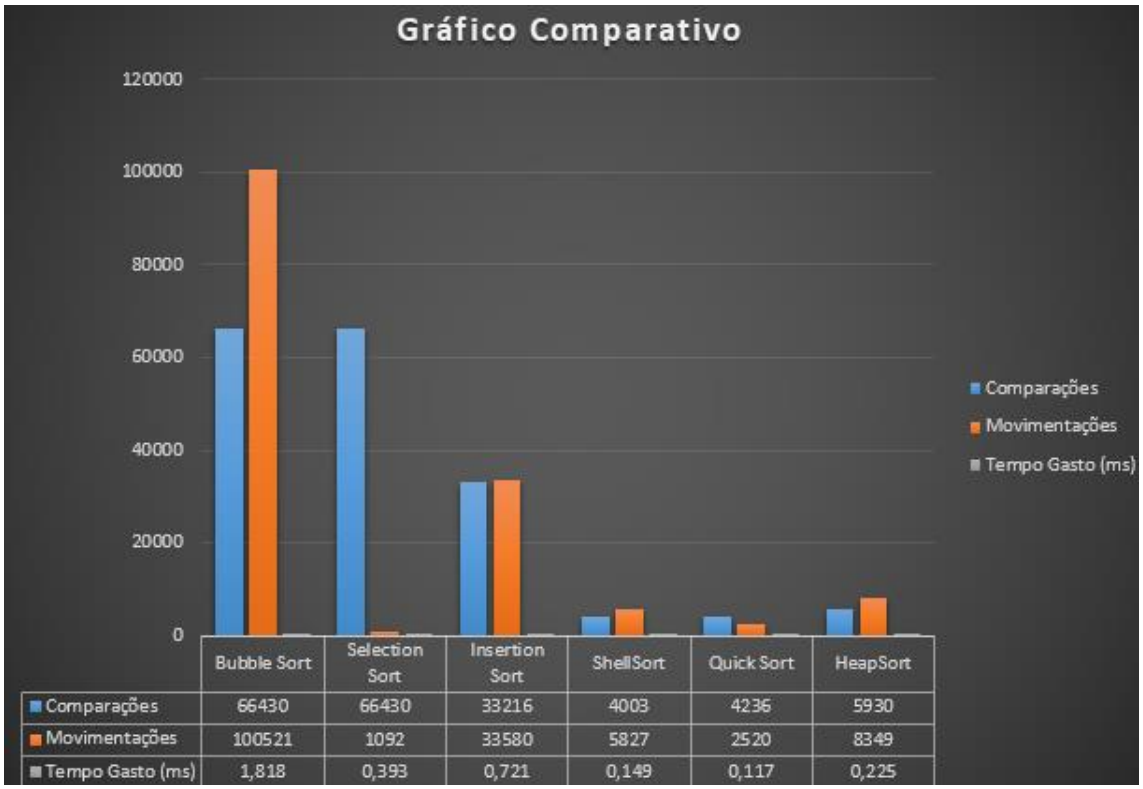
Processador: Intel ® Core™ i5 - 3470U CPU @ 3.20GHz x 4;
Memória RAM: 8 GB;
HD: 200 GB;
Sistema Operacional: Linux Ubuntu 16.04 LTS;
IDE utilizada na implementação: Atom;
Compilador: GCC Compiler.

- Gráfico do cenário 1:



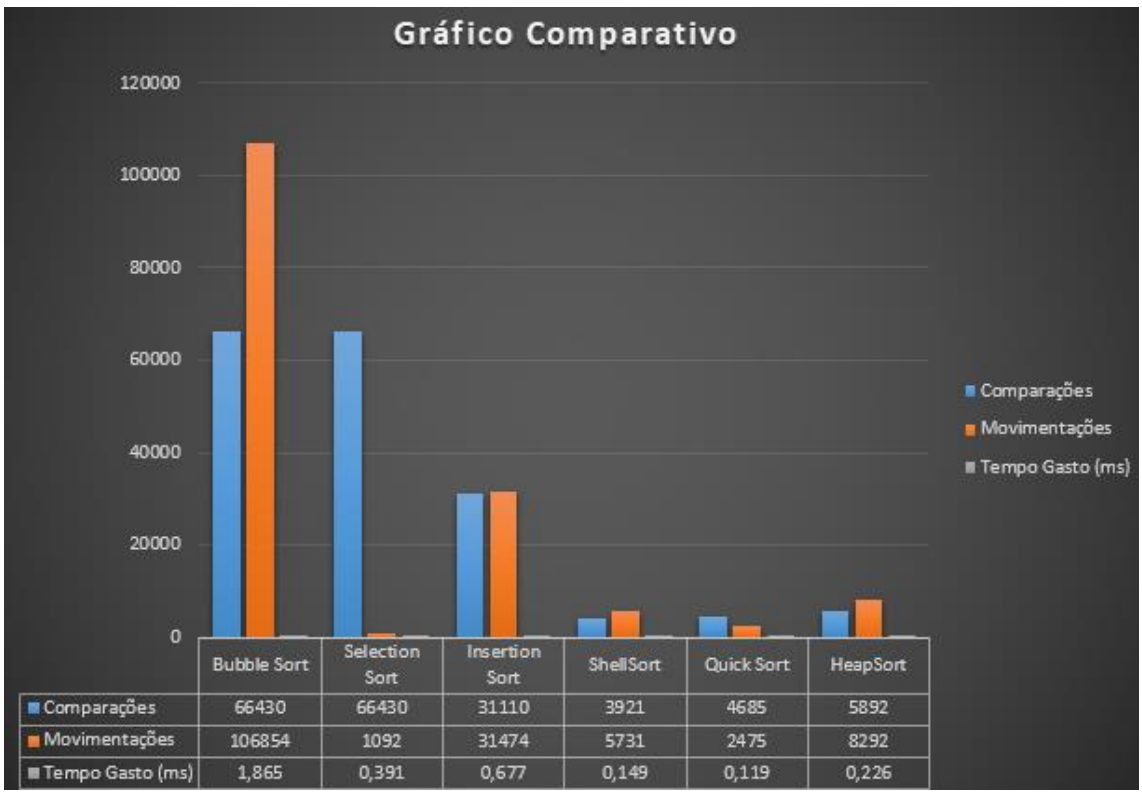
	Bubble Sort	Selection Sort	Insertion Sort	ShellSort	Quick Sort	HeapSort
Comparações	66430	66430	33767	3799	4722	5940
Movimentações	98883	1092	34131	5599	2463	8364
Tempo Gasto (ms)	1,763	0,388	0,731	0,146	0,118	0,25

- Gráfico do cenário 2:



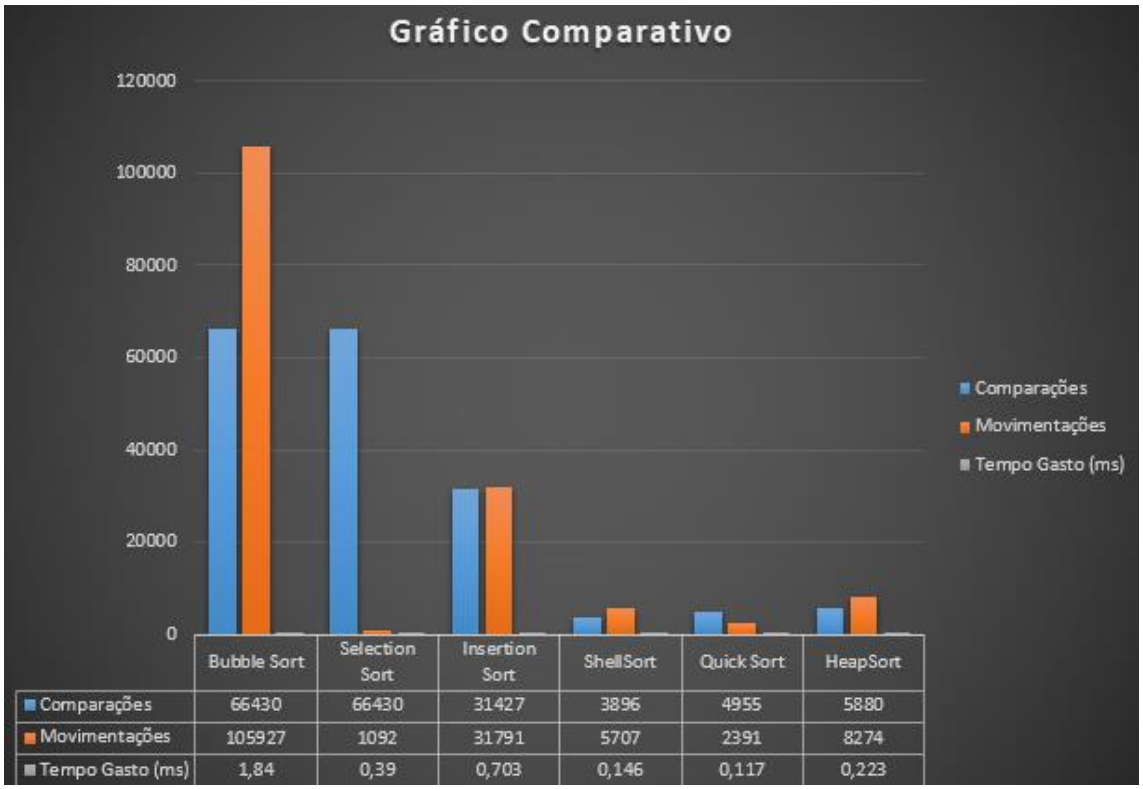
	Bubble Sort	Selection Sort	Insertion Sort	ShellSort	Quick Sort	HeapSort
Comparações	66430	66430	33216	4003	4236	5930
Movimentações	100521	1092	33580	5827	2520	8349
Tempo Gasto (ms)	1,818	0,393	0,721	0,149	0,117	0,225

- Gráfico do cenário 3:



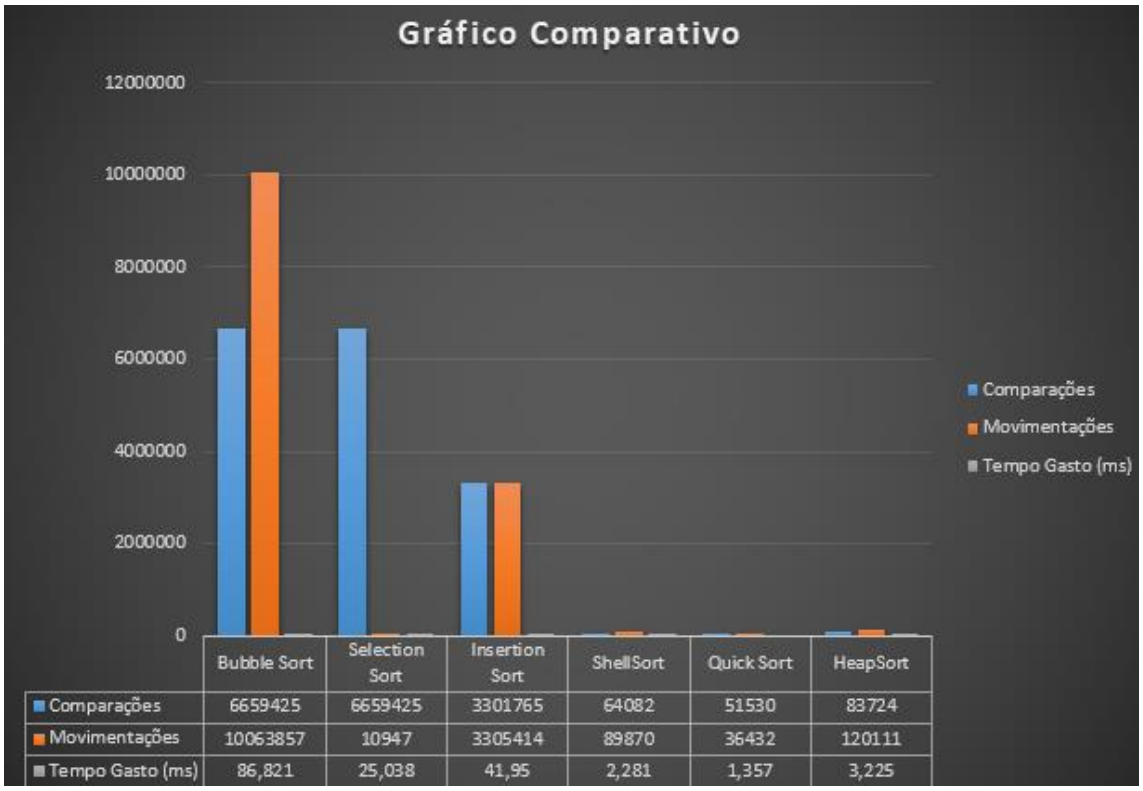
	Bubble Sort	Selection Sort	Insertion Sort	ShellSort	Quick Sort	HeapSort
Comparações	66430	66430	31110	3921	4685	5892
Movimentações	106854	1092	31474	5731	2475	8292
Tempo Gasto (ms)	1,865	0,391	0,677	0,149	0,119	0,226

- Gráfico do cenário 4:



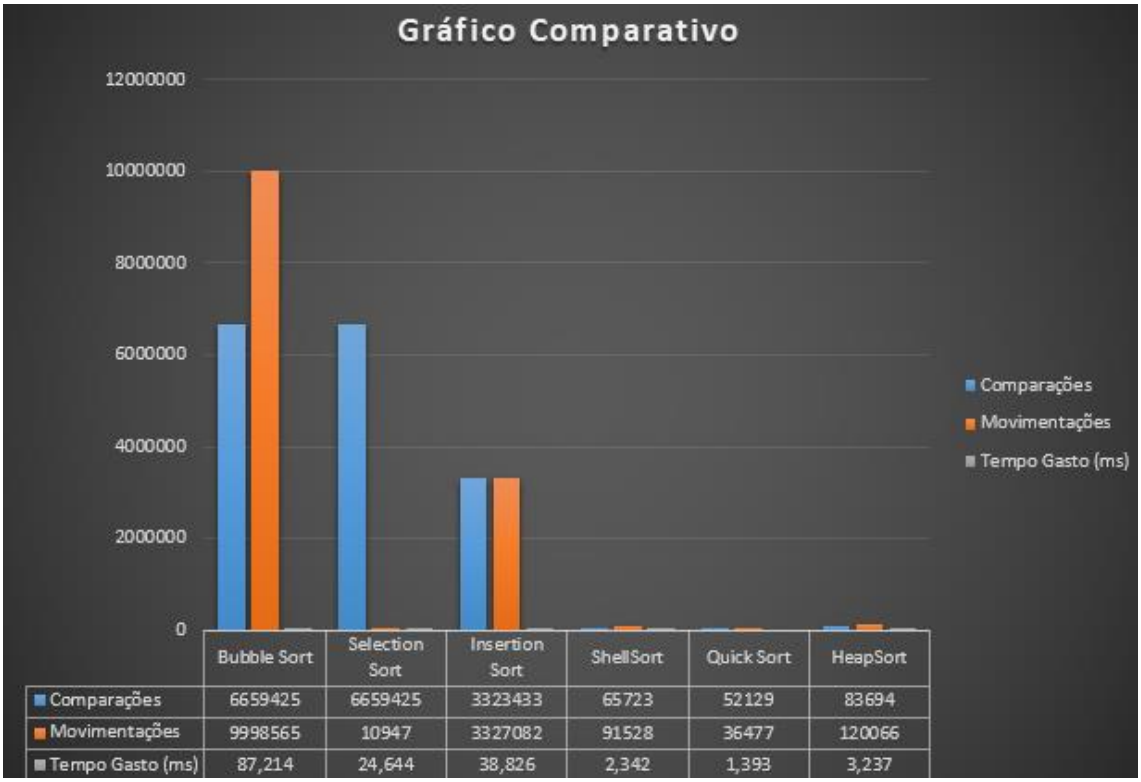
	Bubble Sort	Selection Sort	Insertion Sort	ShellSort	Quick Sort	HeapSort
Comparações	66430	66430	31427	3896	4955	5880
Movimentações	105927	1092	31791	5707	2391	8274
Tempo Gasto (ms)	1,84	0,39	0,703	0,146	0,117	0,223

- Gráfico do cenário 5:



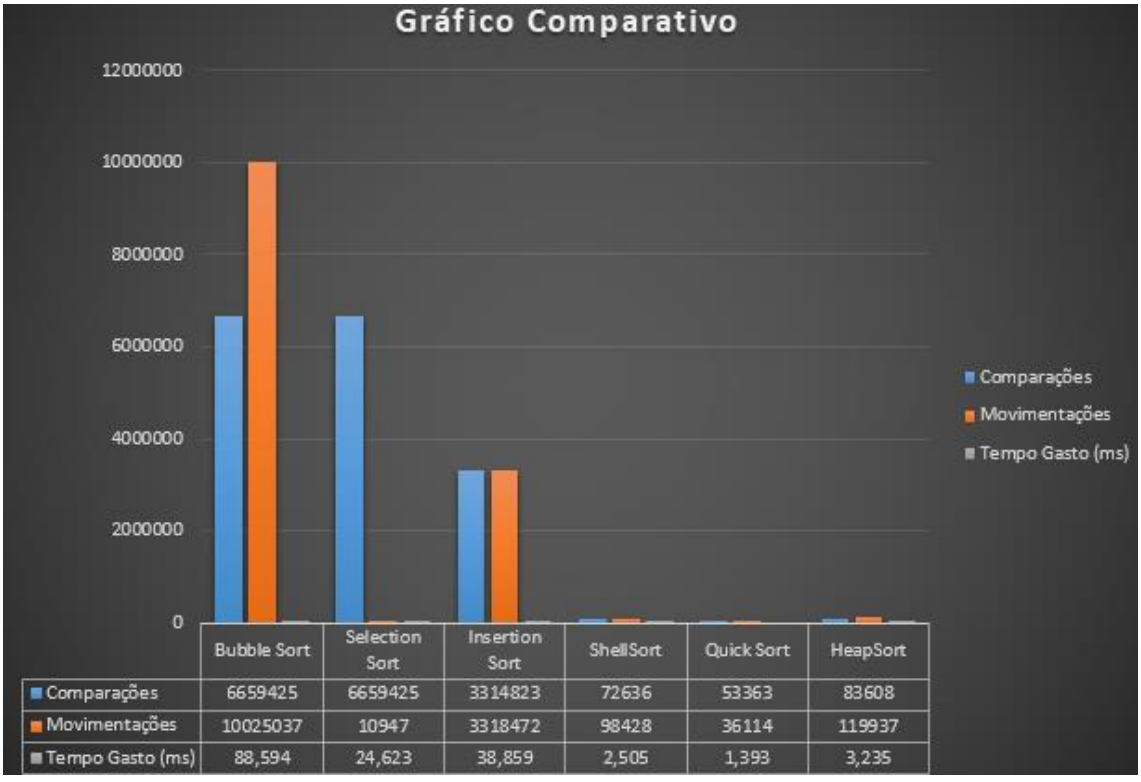
	Bubble Sort	Selection Sort	Insertion Sort	ShellSort	Quick Sort	HeapSort
Comparações	6659425	6659425	3301765	64082	51530	83724
Movimentações	10063857	10947	3305414	89870	36432	120111
Tempo Gasto (ms)	86,821	25,038	41,95	2,281	1,357	3,225

- Gráfico do cenário 6:



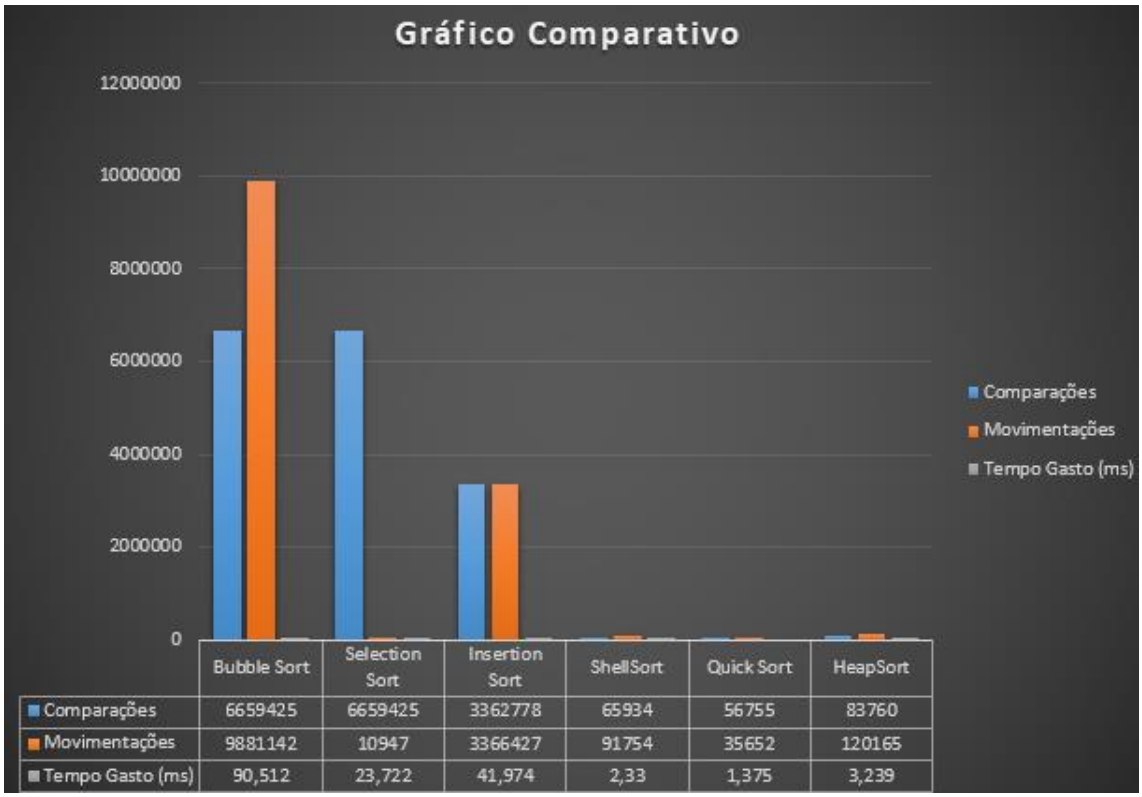
	Bubble Sort	Selection Sort	Insertion Sort	ShellSort	Quick Sort	HeapSort
Comparações	6659425	6659425	3323433	65723	52129	83694
Movimentações	9998565	10947	3327082	91528	36477	120066
Tempo Gasto (ms)	87,214	24,644	38,826	2,342	1,393	3,237

- Gráfico do cenário 7:



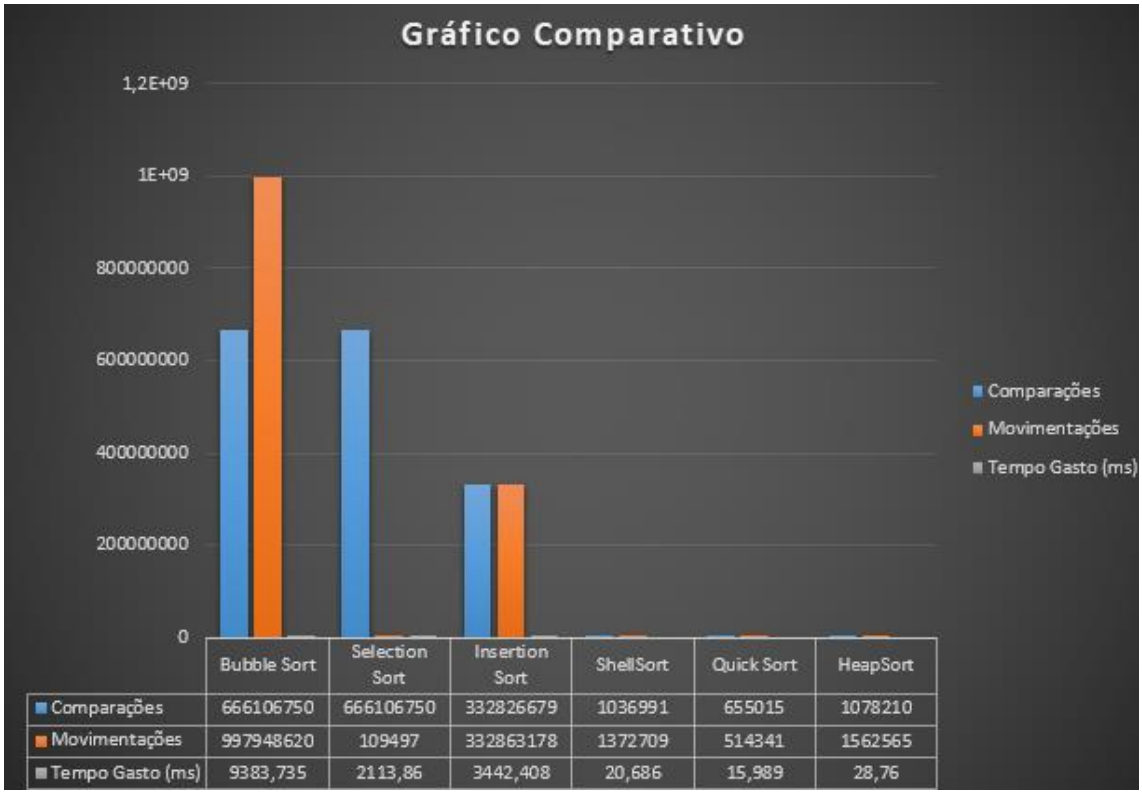
	Bubble Sort	Selection Sort	Insertion Sort	ShellSort	Quick Sort	HeapSort
Comparações	6659425	6659425	3314823	72636	53363	83608
Movimentações	10025037	10947	3318472	98428	36114	119937
Tempo Gasto (ms)	88,594	24,623	38,859	2,505	1,393	3,235

- Gráfico do cenário 8:



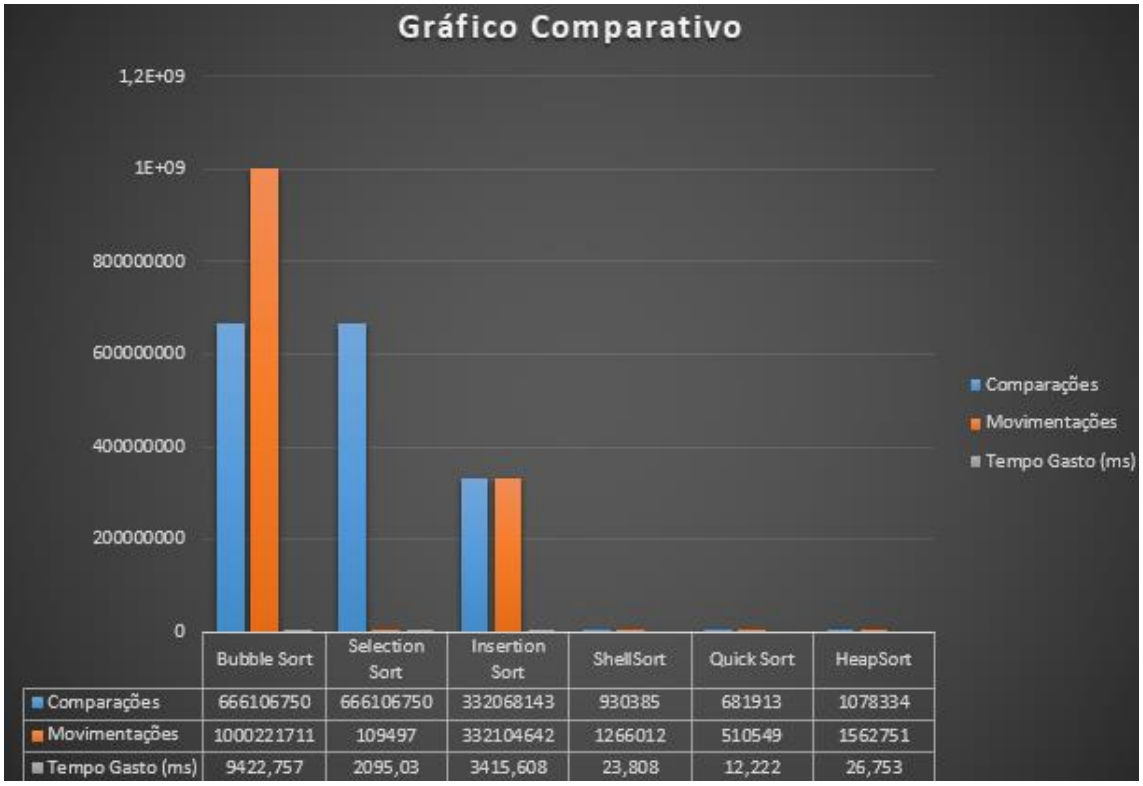
	Bubble Sort	Selection Sort	Insertion Sort	ShellSort	Quick Sort	HeapSort
Comparações	6659425	6659425	3362778	65934	56755	83760
Movimentações	9881142	10947	3366427	91754	35652	120165
Tempo Gasto (ms)	90,512	23,722	41,974	2,33	1,375	3,239

- Gráfico do cenário 9:



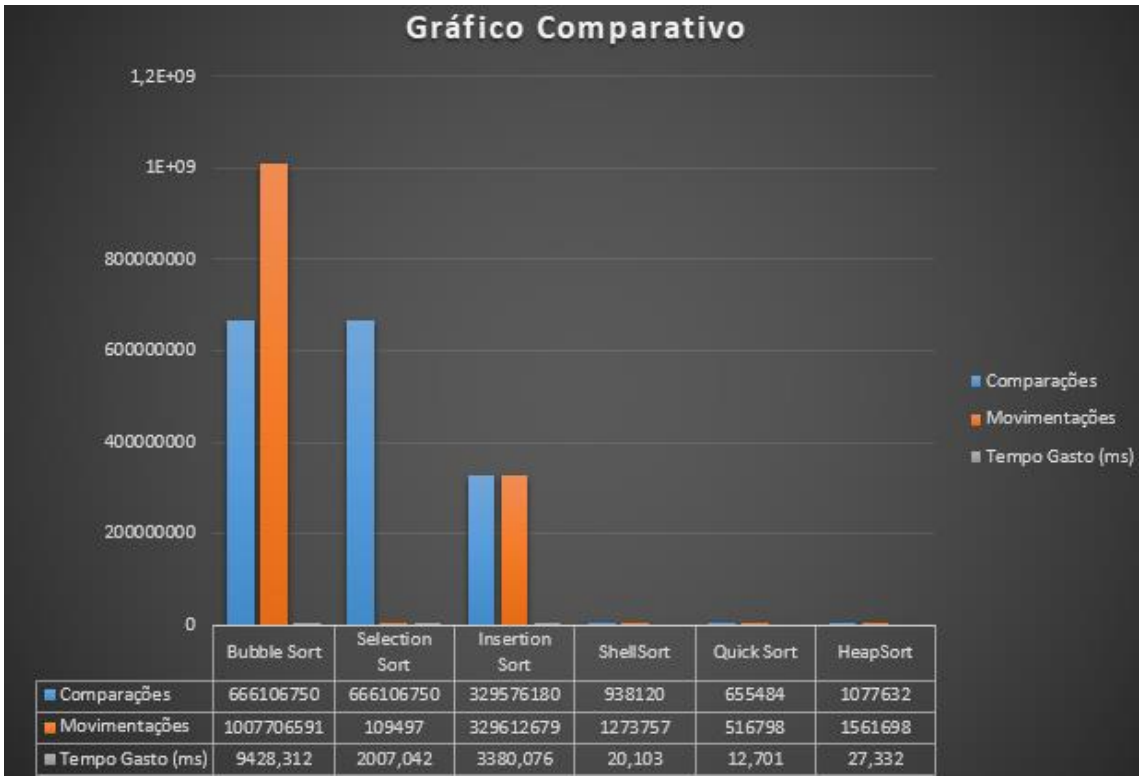
	Bubble Sort	Selection Sort	Insertion Sort	ShellSort	Quick Sort	HeapSort
Comparações	666106750	666106750	332826679	1036991	655015	1078210
Movimentações	997948620	109497	332863178	1372709	514341	1562565
Tempo Gasto (ms)	9383,735	2113,86	3442,408	20,686	15,989	28,76

- Gráfico do cenário 10:



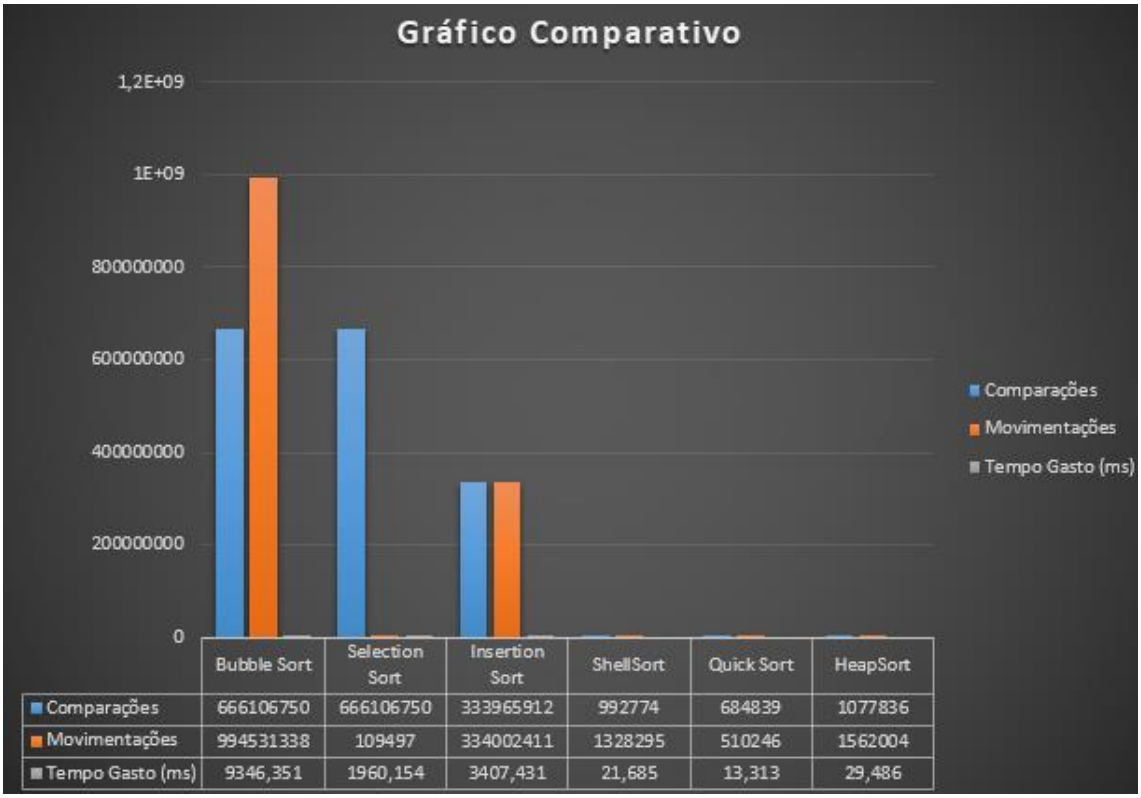
	Bubble Sort	Selection Sort	Insertion Sort	ShellSort	Quick Sort	HeapSort
Comparações	666106750	666106750	332068143	930385	681913	1078334
Movimentações	1000221711	109497	332104642	1266012	510549	1562751
Tempo Gasto (ms)	9422,757	2095,03	3415,608	23,808	12,222	26,753

- Gráfico do cenário 11:



	Bubble Sort	Selection Sort	Insertion Sort	ShellSort	Quick Sort	HeapSort
Comparações	666106750	666106750	329576180	938120	655484	1077632
Movimentações	1007706591	109497	329612679	1273757	516798	1561698
Tempo Gasto (ms)	9428,312	2007,042	3380,076	20,103	12,701	27,332

- Gráfico do cenário 12:



	Bubble Sort	Selection Sort	Insertion Sort	ShellSort	Quick Sort	HeapSort
Comparações	666106750	666106750	333965912	992774	684839	1077836
Movimentações	994531338	109497	334002411	1328295	510246	1562004
Tempo Gasto (ms)	9346,351	1960,154	3407,431	21,685	13,313	29,486

- **Conclusão**

Comparando os gráficos de cada cenário, pudemos concluir que os métodos de ordenação sofisticados diferem de forma gigantesca ao serem comparados com os métodos de ordenação simples.

Com esse trabalho, visualizamos as 3 matérias da disciplinas atuando juntas, de forma que, conseguimos entender as aplicações de tais conteúdos em problemas do mundo real.