Universidade Federal do Maranhão - UFMA
Departamento de Informática - DEINF
Estrutura de Dados II

Maria Jucimara Pereira Ferreira

RELATÓRIO

IMPLEMENTAÇÃO E ANALISE

DE

MÉTODOS DE ORDENAÇÃO

São Luís - MA
2020

1. Introdução

O seguinte relatório de Implementação e Análise de Métodos de Ordenaçao do curso de Ciência da Computacão da Universidade Federal do Maranhão (UFMA), tem como objetivo verifificar qual o método usar em determinadas situaçoes, com base no seu desempenho. Para isso, a metodologia utilizada foi a utilização do algoritmo de Kruskal, por ter uma etapa que necessita da execução de um algoritmo de ordenação.

2. Kruskal

O algoritmo de Kruskal inicia com V diferentes árvores (V são vértices do grafo). Este algoritmo encontra a aresta segura (e com menor peso) procurando em todas as arestas que conectam quaisquer duas árvores na floresta.O algoritmo de Kruskal utiliza conjuntos disjuntos, em cada conjunto contém os vértices de uma árvore na floresta atual. A operação FIND-SET(u) retorna um elemento representativo do conjunto que contém u . Assim, para determinar se dois vértices u e v pertencem à mesma árvore, basta testar se FIND-SET(u) é igual a FIND-SET(v). A combinação de árvores é executada com o procedimento UNION.

Obs: A implementação deste algoritmo não será discutida neste relatório

3. Metodos de Ordenação

- Ordenação por Particionamento (QuickSort)
- Ordenação por Inserção (Insertion Sort)
- Ordenação por Inserção através de incrementos decrescentes (ShellSort)
- Ordenação por Árvores (HeapSort)
- Ordenação por Particionamento e Inserção (QuickSort + Insert)

Notação O

Algoritmo	Comparações			Movimentações			Espaço	Estável	In situ
	Melhor	Médio	Pior	Melhor	Médio	Pior			
Bubble	$O(n^2)$			$O(n^2)$			$O(1)$	**Sim**	**Sim**
Selection	$O(n^2)$			$O(n)$			$O(1)$	**Não***	**Sim**
Insertion	$O(n)$	$O(n^2)$		$O(n)$	$O(n^2)$		$O(1)$	**Sim**	**Sim**
Merge	$O(n\ log\ n)$			–			$O(n)$	**Sim**	**Não**
Quick	$O(n\ log\ n)$		$O(n^2)$	–			$O(n)$	**Não***	**Sim**
Shell	$O(n^{1.25})$ ou $O(n\ (ln\ n)^2)$			–			$O(1)$	**Não**	**Sim**

* Existem versões estáveis.

Analise dos algoritmos de ordenacao: (número de vertices) X (tempo em ms)

	7	100	1000	10000	100000
Quick	0	0.001	0.002	0.032	1.904
Merge	0	0.001	0.008	0.092	5.079
Insert	0	0.002	0.024	0.0159	33.601
Shell	0	0.001	0.008	0.012	0.123
Heap	0	0.001	0.005	0.016	0.108
QuickIP	0.001	0.004	0.027	0.172	14.692
QuickIF	0.002	0.002	0.006	0.016	0.174

3.1 Quick Sort

```
public static List<Edge> quicksort(List<Edge> lista, int inicio, int
fim) {
        int pos;

        if (inicio < fim) {

            pos = particionar(lista, inicio, fim);
```

```
            quicksort(lista, inicio, pos - 1);
            quicksort(lista, pos + 1, fim);

        }
        return lista;
    }

    // pivo como elemento mais a esquerda
    public static int particionar(List<Edge> lista, int esquerda, int
direita){
        int i = esquerda + 1;
        int j = direita;
        Edge pivo = lista.get(esquerda);


        while(i <= j){

            if((int)lista.get(i).peso <= pivo.peso) {
                i++;
            } else if(pivo.peso <= (int)lista.get(j).peso) {
                j--;
            } else {
                trocar(lista, i, j);
                i++;
                j--;
            }
        }
        trocar(lista, esquerda, j);

        return j;
    }

    public static void trocar(List<Edge> lista, int i, int j){
        Edge aux = lista.get(i);
        lista.set(i, lista.get(j));
        lista.set(j, aux);

    }
```

3.2 MergeSort

```
public static List<Edge> mergesort(List<Edge> vertices, int esquerda,
int direita) {
        int meio;
        if (esquerda < direita) {

            meio = ((esquerda + direita) / 2);
            mergesort(vertices, esquerda, meio);   // div pela esquerda
            mergesort(vertices, meio + 1, direita); // div pela direita
            Intercalar(vertices, esquerda, meio, direita);
        }
        nComparacoes++;
        return vertices;
    }

    public static List<Edge> Intercalar(List<Edge> vertices, int
esquerda, int meio, int direita) {
        List<Edge> verticesCopia = new ArrayList<Edge>(vertices);

        int contEsquerda = esquerda;    // contador da esquerda | i
        int contDir = meio + 1; // contador da direita | j
        int contCopia = esquerda;   //contador da copia da vertices

        while (contCopia <= direita) {
            if (contEsquerda > meio) {
                vertices.set(contCopia, verticesCopia.get(contDir));
                contDir +=1;
            } else if (contDir > direita) {
                vertices.set(contCopia,
verticesCopia.get(contEsquerda));

                contEsquerda++;

            } else if ((verticesCopia.get(contEsquerda).peso) <
(verticesCopia.get(contDir).peso)) {
                vertices.set(contCopia,
verticesCopia.get(contEsquerda));

                contEsquerda++;
            } else {
```

```
                vertices.set(contCopia, vertices.get(contDir));
                contDir++;
            }
            contCopia++;
        }
        return vertices;
    }
```

3.3 InsertSort

Como o primeiro elemento é dito ja ordenado, em cada passo, a partir de i = 2, o i-ésimo item da sequência da lista é apanhado e tranferido para seu devido lugar

```
public static List<Edge> InsertSort(List<Edge> lista) {
        int i;
        int j;
        // o primeiro  elemento e dito ja ordenado
        for (i = 1; i < lista.size(); i++) {
            Edge chave = lista.get(i);      // inicializar o auxiliar
            j = i;

            while((j > 0) && (lista.get(j - 1).peso > chave.peso)) {
                lista.set(j,lista.get(j - 1));
                j--;
                            }
            lista.set(j, chave);
        }
        return lista;

    }
```

3.4 ShellSort

Extensão do algoritmo de Inserção, proposto por Shell(1959).

Permite a troca de resgistros que estão longe um do outro.
Os itens da lista são rearranjados H posições, de tal forma que todo h-ésimo item é dito h ordenado.

```
public static List<Edge> Shell(List<Edge> lista) {
        int n = lista.size();

        // Comece com um grande h, em seguida, reduzir o h
        for (int h = n / 2; h > 0; h /= 2) {
            // Faça um tipo de insercao escancarada para este tamanho
de lacuna.
            // Os primeiros elementos de lacuna a[0..h-1] já estão
            // em ordem. Continua adicionando mais um elemento
            // até que toda a lista seja classificada
            for (int i = h; i < n; i += 1) {
                // adicionar a[i] aos elementos que foram h
                //  classificado salvar um [i] em temperatura e fazer
um buraco na posicao i
                Edge temp = lista.get(i);
                // mudar elementos anteriores classificados até o local
                // correto para a[i] ser encontrado
                int j;
                for (j = i; ((j >= h) && (lista.get(j - h).peso >
temp.peso)); j -= h){
                    lista.set(j, lista.get(j - h));
                     // coloca o temporario o a[i] original em sua
localizacao correta
                }
                lista.set(j, temp);
            }
        }
        return lista;
    }
```

3.5 HeapSort

```
public static List<Edge> heapSort(List<Edge> arestas) {
```

```
        int n = arestas.size();

        for (int i = n/2 - 1; i >= 0; i--) { // Constroi heap
(reorganizar array)
            heapify(arestas, n, i);
        }
        for (int i = n - 1; i >= 0; i--) { // Extrai um elemento do
heap um por um
            // Move a raiz atual para o fim
            Edge aux = arestas.get(0);
            arestas.set(0, arestas.get(i));
            arestas.set(i, aux);
            heapify(arestas, i, 0);

        }

        return arestas;

    }

    public static void heapify(List<Edge> arestas, int n, int i) {
        int largura = i;
        int esquerda = 2*i + 1;
        int direita = 2*i + 2;

        if (esquerda < n && arestas.get(esquerda).peso >
arestas.get(largura).peso) { // Se o filho esquerdo for maior que a
raiz

            largura = esquerda;
        }

        if (direita < n && arestas.get(direita).peso >
arestas.get(largura).peso) { // Se o filho direito for maior do que o
maior até agora

            largura = direita;
        }

        if (largura != i) { // Se o maior não for raiz
            Edge aux = arestas.get(i);
            arestas.set(i, arestas.get(largura));
```

```
            arestas.set(largura, aux);

            heapify(arestas, n, largura);
        }
    }
```

3.6 QuickSort + Insercao Parcial

O QuickSort devera ser executado recursivamente (como padrao) e, quando voce obtiver uma particao de tamanho menor ou igual a L, essa particao devera ser ignorada e continuar particionando as particoes maiores. Uma vez que todas as particoes tenham comprimento menor ou igual a L, o algoritmo devera executar o algoritmo de ordenacao.

```
public static int ParticionarQuick (List<Edge> arestas, int inicio, int
fim) {
        // Escolha o elemento mais à direita como pivô da matriz
        Edge pivot = arestas.get(fim);

        // elementos menos que pivô será empurrado para a esquerda do
pIndex
        // elementos mais do que pivô será empurrado para a direita do
pIndex
        // elementos iguais podem ir de qualquer maneira
        int pIndex = inicio;

        // cada vez que encontramos um elemento menor ou igual ao pivô,
        // pIndex é incrementado e esse elemento seria colocado
        // antes do pivô.
        for (int i = inicio; i < fim; i++) {
            if (arestas.get(i).peso <= pivot.peso) {
                Edge temp = arestas.get(i);
                arestas.set(i, arestas.get(pIndex));
                arestas.set(pIndex,temp);
                pIndex++;
            }
        }
        // trocar pIndex com Pivô
        Edge temp = arestas.get(fim);
```

```
        arestas.set(fim,arestas.get(pIndex));
        arestas.set(pIndex,temp);

        return pIndex;
    }

    public static void QuickSort(List<Edge> arestas, int inicio, int
fim) {
        if (inicio >= fim)
            return;

        // rearrange the elements across pivot
        int pivot = ParticionarQuick(arestas, inicio, fim);

        // recursar em sub-matriz contendo elementos menores que pivô
        QuickSort(arestas, inicio, pivot - 1);

        // recursar em sub-matriz contendo elementos mais do que pivô
        QuickSort(arestas, pivot + 1, fim);
    }

    public static List<Edge> partialQuickInsert(List<Edge> arestas, int
inicio, int fim) {
        while (inicio < fim) {
            // A ordenacao por insercao e feita se menor que 10
            if (fim - inicio < 10) {
                InsertionSort.InsertSort(arestas);
                break;
            } else {
                int pivot = ParticionarQuick(arestas, inicio, fim);

                // otimizações de chamada de cauda - recur em sub array
menor
                if (pivot - inicio < fim - pivot) {
                    partialQuickInsert(arestas, inicio, pivot - 1);
                    inicio = pivot + 1;
                } else {
                    partialQuickInsert(arestas, pivot + 1, fim);
                    inicio = pivot - 1;
                }
            }
        }
        return arestas;
```

```
    }
```

Nota: O principal cuidado a ser tomado é com relação à escolha do pivô, por isso a mediana de uma amostra de três elementos é usada.

3.7 QuickSort + Inserção Final

O QuickSort é executado recursivamente e, quando obtitem uma particao de tamanho menor ou igual a L, essa particao é ignorada e continua particionando as particoes maiores. Uma vez que todas as particoes tenham comprimento menor ou igual a L, o algoritmo executa o algoritmo de ordenacao. Para fins de teste o L foi predeterminado com 10

```
public static List<Edge> QuicksortF(List<Edge>  arestas, int p, int r)
{

        if((r-p)>10) { // listas maiores que 10
            int q;
            int p1 = p;
            int r1 = r;

            Med3(arestas, p1,r1); //  pivo
            Edge x = arestas.get(p1);

            while(true) {

                do{
                    r1--;
                } while(arestas.get(r1).peso > x.peso);

                arestas.set(p1, arestas.get(r1));

                do{
                    p1++;
                }while(arestas.get(p1).peso < x.peso);

                if(p1<r1){
```

```
                arestas.set(r1, arestas.get(p1));
            } else {
                if(arestas.get(r1+1).peso <= x.peso)
                    r1++;
                    arestas.set(r1, x);
                    q = r1;
                    break;
            }
        }
        QuicksortF(arestas, p, q-1);
        QuicksortF(arestas, q+1, r);
    }
    return arestas;
}

public static List<Edge> Med3(List<Edge>  arestas, int p, int r) {
// Selects the median and sets the sentinels
    int mid = (p+r)/2;
    int largura;

    if(arestas.get(p).peso > arestas.get(mid).peso) {
        largura = p;
    } else{
        largura = mid;
    }

    if(arestas.get(largura).peso > arestas.get(r).peso) {
        Edge temp = arestas.get(r);
        arestas.set(r, arestas.get(largura));
        arestas.set(largura, temp);
    }

    if(arestas.get(mid).peso > arestas.get(p).peso) {
        Edge temp = arestas.get(p);
        arestas.set(p, arestas.get(mid));
        arestas.set(mid, temp);
    }
    return arestas;
}
```

Para uso dos metodos de ordenacao com o kruskal foi criado o seguinte método em cada umas das classes de ordenacao. Ex:

```
ublic static void sorteio(List<Edge> lista) {
        long inicio;
        long fim;
        // System.out.println("Arquivo de teste carregado");
        inicio = System.currentTimeMillis();    // para verificar
otempo execucao
        lista = InsertSort(lista);  // lista sera ordenado em ordem
crescente
        // System.out.println("Aplicado metodo InsertSort");
        fim = System.currentTimeMillis();


        System.out.println("Ordenado");
        for(Edge e : lista) {
            System.out.println(e.v+", " + e.peso+", " + e.w);   //
printar na tela os valores já ordenados
        }

        System.out.println("Comparacoes: " + nComparacoes);
        System.out.println("Movimentacoes: "+ nMovimentacoes);
        System.out.printf("Tempo de processamento do Insertion: %.3f
ms%n\n", (fim - inicio) / 1000d);
    }
```

Onde aresta é declarado como sendo do tipo List<Edge> que por sua vez vai ser carregada com o arquivos com uma lista de arestas do tipo Edge

```
public class Edge {
   public int v;
   public int peso;
   public int w;

   public Edge(int v, int peso, int w) {
      this.v = v;
      this.peso = peso;
      this.w = w;
   }
}
```

Essas arestas formaram um Grafo (Kruskal), que produzirá uma mst do tipo kruskal, de acordo com o número de vertices escolhido

```
arestas = readCSVfile("../TRABALHO/src/teste_vertices/"+nVertices+"_vertices.csv");

    Graph graph = new Graph(arestas); // inicializa o grafo com edges
    List<Edge> mst= graph.Kruskal(arestas);
```

Exemplo de Entrada e Saida

==========

Escolha a quantidade de elementos no arquivo

==========

1 - 7 elementos

2 - 100 elementos

3 - 1.000 elementos

4 - 10.000 elementos

5 - 100.000 elementos

0 - Sair

Opcao:

1

1, 7, 2

1, 5, 4

2, 8, 3

2, 7, 5

4, 6, 6

5, 9, 7

==========

Escolha o algoritmo de ordenacao:

==========

1 - QuickSort

2 - MergeSort

3 - InserctionSort

4 - ShellSort

5 - HeapSort

6 - QuickSort + Insercao Parcial

7 - QuickSort + Insercao Final

0 - Sair

Opcao:

3

Ordenado

1, 5, 4

4, 6, 6

1, 7, 2

2, 7, 5

2, 8, 3

5, 9, 7

Comparacoes: 15

Movimentacoes: 5

Tempo de processamento do InsertSort: 0,000 ms

Para mais detalhes: TRABALHO\Relatorio.txt possui todas as comparações, modificações e tempo de execução de cada um dos algoritmos.

Quantidade de teste:

7 - Média de 4 a 6 teste

100 - Média de 5 a 7 testes

1000 - Média de 6 teste

10000 - Média de 5 teste

100000 - Média de 3 teste

4. Considerações Finais

Os teste foram realizados em uma máquina com Processador: Intel(R) Core(TM i5-8265U CPU @1,60GHz 1,80 GHz

O tempo calculado só leva em consideração a ordenacão dos vertices

As Comparacoes e Modificacoes apresentadas tem como base a pasta teste_vertices

O Quick sort se mostrou o algoritmo mais eficiente, não importando a quantidade de vertices do arquivo. Além de o demandar uma pequena quantidade de memória adicional, no pior caso ele irá realizar O(n^2) operações, porém o pior caso tem uma probabilidade muito pequena de ocorrer quando os elementos forem aleatório.. A parte mais delicada é na hora de particionar, porque temos que rearranjar a lista por meio da escolhar de um pivo x tal que os elementos da esquerda são menores ou iguais a x e os elementos da direita maiores.

O método de inserção é melhor utilizado quando a lista está quase ordenada, por essa razão o QuickSort com inserção final tem seu tempo de desempenho melhorado significativamente.

O ShellSort é rápido, eficiente e ótimo para arquivos de tamanho moderado, porém como não é estável ele realiza trocas desnecessárias, como no caso de elementos iguais.

Dependendo do tamanho do arquivo o ShellSort pode ser mais rápido que o QuickSort.

O QuickSort e o Heap obtiveram temos aproximados, todavia, o Quick é mais rápido, porque o tempo necessário para construir a heap é alto.

Bibliografia:

Ziviani, N. Projeto de Algoritmos Com Implementações em Pascal e C, Pioneira Thomson Learning, 2007.

Simulador de ordenação
Disponível em: https://visualgo.net/pt/sorting?slide=1
Acesso em: 05 out. 2020.

Analysis of Algorithms
Disponível em: https://www.geeksforgeeks.org/fundamentals-of-algorithms/#AnalysisofAlgorithms. Acesso em: 21 set. 2020.