Universidade Federal
de São João del-Rei

UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL-REI

Algoritmo e estrutura de dados III

HIPERCAMPOS

Prof.: Dr. Leonardo Rocha

Gustavo Henriques

Matheus N Silva

São João del-Rei, 11 de abril de 2023

Sumário

1 INTRODUÇÃO

1.1 Proposta

Foi proposto como primeiro trabalho prático da disciplina de Algoritmo e Estrutura de Dados III a implementação de uma aplicação onde, a partir de uma entrada de N pontos (sendo que dois desses pontos são ancoras), conseguisse computar o número máximo de pontos que seja possível conectar através de segmentos de reta, sendo que sua intersecção só exista nos pontos ancoras, ou seja, sem que esses segmentos se intersectem no caminho.

1.2 Objetivo

Após a implementação da aplicação o objetivo principal torna a ser a obtenção de dados referente à sua execução, tanto dados finais, tais como resultados corretos, como seu desempenho durante a mesma.

Para chegarmos à esses dados a aplicação foi testada inúmeras vezes com vários parâmetros distintos de entrada

Através deste trabalho foi testado e aprimorado os conhecimentos até aqui adquiridos durante o curso.

1.3 Execução

Para compilar e executar a aplicação deve-se abrir o terminal na pasta do arquivo (ou navegar até a pasta do arquivo) e digitar os seguintes comandos:

- Compilar:

  $ make

- Executar:

  $ ./main -i "nome_arquivo_entrada".txt -o "nome_arquivo_saida".txt

- Teste:

  $ ./auto.sh

2 IMPLEMENTAÇÃO

Este capitulo apresenta de forma breve algumas explicações e comentários a respeito da implementação dos algoritmos utilizados para a resolução da proposta.

2.1 Main

A main apresenta as chamadas dos métodos principais da aplicação. É nela também que estão declaradas algumas variáveis (obrigatórias) utilizadas.

```
int main (int argc, char **argv) {

    struct rusage usage_start, usage_end;
    struct timeval time_start, time_end, time_diff;

    getrusage(RUSAGE_SELF, &usage_start);
    gettimeofday(&time_start, NULL);

    FILE (*inFile);
    FILE (*outFile);
    if (openFile(argc, argv, &inFile, &outFile) != 1) {
        perror("ERRO!");
        exit(0);
    }

    lista* list = readFile(&inFile);
    mergeSort(list, 0, list->tamanho-1);
    maxLinkedPoints(list);

    getrusage(RUSAGE_SELF, &usage_end);
    gettimeofday(&time_end, NULL);

    timeval_subtract(&time_diff, &time_end, &time_start);
    double user_time = (double) (usage_end.ru_utime.tv_sec - usage_start.ru_utime.tv_sec) +
    (double) (usage_end.ru_utime.tv_usec - usage_start.ru_utime.tv_usec) / 1000000.0;
    double sys_time = (double) (usage_end.ru_stime.tv_sec - usage_start.ru_stime.tv_sec) +
    (double) (usage_end.ru_stime.tv_usec - usage_start.ru_stime.tv_usec) / 1000000.0;
    double in_out_time = (double) (time_diff.tv_sec) + (double) (time_diff.tv_usec) / 1000000.0;

    fprintf(outFile, "%d;%d;%f;%f;%f;%f\n", list->tamanho, list->ligacoesMaximas,
        user_time + sys_time, in_out_time, user_time, sys_time);

    free(list->listaPontos);
    free(list->ancoras);
    free(list);
    fclose(inFile);
    fclose(outFile);

    return 0;
}
```

Nota-se que estão declaradas variáveis de estruturas Rusage e Timeval onde serão armazenadas o tempo de programa e o tempo de relógio respectivamente. Após a inicialização dessas variáveis são abertos os arquivos de entrada e saida passados via terminal, sendo

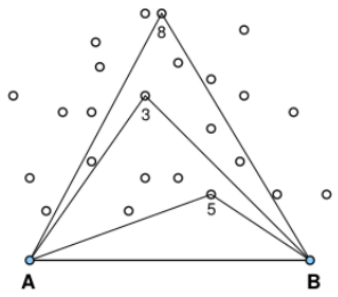
acionado o método de leitura do arquivo de entrada preenchendo assim à estrutura de armazenamento dos dados. Na sequencia é realizado a ordenação dessa estrutura para que possa ocorrer o calculo da quantidade maxima de pontos que podem ser interligados a partir das ancoras sem ocorrer intersecção entre eles.

2.2 Cálculo de intersecção

O método maxLinkedPoint é o próximo a ser chamado pelo main. Ele recebe uma lista com os pontos a serem interligados e verifica através do método isInside se os pontos estão abaixo da intersecção superior.

```
void maxLinkedPoints (lista* list) {
    //list->listaPontos[0].pontosDentro = 1;
    for (int i = 1; i < list->tamanho; i++) {
        for (int j = (i-1); j >= 0; j--) {
            //verificando se os pontos "abaixo" de [i] estao dentro dele
            if (isInside(list->ancoras[0].x, list->ancoras[1].x,
             list->listaPontos[i].x, list->listaPontos[i].y,
             list->listaPontos[j].x, list->listaPontos[j].y) == 1) {
                //verificando quantos pontos existem dentro de [j]
                if (list->listaPontos[i].pontosDentro <= list->listaPontos[j].pontosDentro) {
                    list->listaPontos[i].pontosDentro = list->listaPontos[j].pontosDentro + 1;
                }
            }
        }
        if (list->listaPontos[i].pontosDentro > list->ligacoesMaximas) {
            list->ligacoesMaximas = list->listaPontos[i].pontosDentro;
        }
    }
    /*o numero de ligacoes maximas e o numero do maior conjunto de pontos
     dentro de outros pontos mais o proprio ponto*/
    list->ligacoesMaximas++;
}
```

Observando o algoritmo acima é possivel notar que após calcular a quantidade maxima de ligações possíveis é necessário adicionar mais uma ligação, que seria a intersecção superior.



Obs.: No exemplo ao lado a intersecção superior é apresentada pelos segmentos de reta A8 e B8.

O método isInside recebe as duas ancoras e as coordenadas x e y de cada ponto à ser verificado.

```
int isInside (int xA, int xB, int xptC, int yptC, int xptD, int yptD) {
    if ((xptC >= xA) && (xptD < xA)) {
        //D nao esta contido em C
        return -1;
    }

    if ((xptC <= xB) && (xptD > xB)) {
        //D nao esta contido em C
        return -1;
    }

    if ((xptC <= xA) && (xptD > xA)) {

    } else {
        if (findSlope(xA, 0, xptC, yptC) < findSlope(xA, 0, xptD, yptD)) {
            //D nao esta contido em C
            return -1;
        }
    }

    if ((xptC >= xB) && (xptD < xB)) {

    } else {
        if (findSlope(xB, 0, xptC, yptC) > findSlope(xB, 0, xptD, yptD)) {
            //D nao esta contido em C
            return -1;
        }
    }
    //D esta contido em C
    return 1;
}
```

Para que ocorra a verificação de forma correta é necessário que se calcule o coeficiente angular da reta tarefa essa designada ao método findSlope

```
//calcula o coeficiente angular da reta
//tga = yb - ya / xb - xa
double findSlope (int xA, int yA, int xB, int yB) {
    double tg;
    if (xA == xB) {
        tg = 0;
        return tg;
    }
    tg = (double) (yB - yA) / (xB - xA);
    return tg;
}
```

3 COMPLEXIDADE

Para calcularmos a complexidade do algoritmo desenvolvido iremos utilizar apenas o calculo de intersecção, tendo em vista que de todos os métodos, ele é o mais custoso.

Devemos então observar o algoritmo novamente.

```
void maxLinkedPoints (lista* list) {
    //list->listaPontos[0].pontosDentro = 1;
    for (int i = 1; i < list->tamanho; i++) {
        for (int j = (i-1); j >= 0; j--) {
            //verificando se os pontos "abaixo" de [i] estao dentro dele
            if (isInside(list->ancoras[0].x, list->ancoras[1].x,
             list->listaPontos[i].x, list->listaPontos[i].y,
             list->listaPontos[j].x, list->listaPontos[j].y) == 1) {
                //verificando quantos pontos existem dentro de [j]
                if (list->listaPontos[i].pontosDentro <= list->listaPontos[j].pontosDentro) {
                    list->listaPontos[i].pontosDentro = list->listaPontos[j].pontosDentro + 1;
                }
             }
        }
        if (list->listaPontos[i].pontosDentro > list->ligacoesMaximas) {
            list->ligacoesMaximas = list->listaPontos[i].pontosDentro;
        }
    }
    /*o numero de ligacoes maximas e o numero do maior conjunto de pontos
     dentro de outros pontos mais o proprio ponto*/
    list->ligacoesMaximas++;
}
```

Podemos notar que o algoritmo possui um laço **for** aninhado dentro de outro laço **for**, instintivamente ja falaríamos $\mathbf{O(n^2)}$ mas porque a complexidade dele é essa? Observando cada parte do algoritmo notamos que o laço interno é executado n vezes, ja o laço externo é executado n-1 vezes. Então para descobrirmos a complexidade precisamos utilizar um somatório:

$$\sum_{1}^{n-1}(n-1) = \frac{n(n-1)}{2} = \frac{n^2-n}{2}$$

Obs.: Ignoramos o custo das atribuições, incrementações e comparações pois em todos os casos o custo de tais operações é sempre constante.

Como sempre tomamos o termo mais "custoso"da função como base, temos que a aplicação é $O(n^2)$. Após esse calculo ainda seria necessário adicionar o custo da ordenação, mas como o merge sort é, notoriamente, O(n log n), podemos "ignorar", já que a ordem do método principal o domina assintoticamente.

4 AVALIAÇÃO DE RESULTADOS

Foram realizados testes com mais de cinquenta entradas diferentes, utilizando um shellscript para tal, sendo o mesmo executado algumas vezes para a obtenção de um maior número de dados para comparação. Os resultados obtidos após essa bateria de testes estão representados na tabela abaixo:

Entrada	Saida	Tempo (Sys + User)	Tempo Relógio
7590	156	0.567s	0.566s
7666	31	0.446s	0.445s
2002	62	0.040s	0.040s
7897	95	0.567s	0.566s
6233	72	0.373s	0.373s
5500	102	0.302s	0.308s
3034	59	0.080s	0.079s
5781	100	0.298s	0.300s
5128	64	0.221s	0.220s
6629	149	0.441s	0.441s
282	18	0.006s	0.004s
353	24	0.004s	0.003s
5421	36	0.272s	0.271s
662	13	0.008s	0.007s
9448	128	0.833s	0.833s
4433	115	0.196s	0.195s
2407	28	0.047s	0.046s
3920	16	0.134s	0.134s
2912	76	0.113s	0.112s
4301	79	0.176s	0.177s

Obs.: Nessa tabela são apresentados apenas 20 resultados obtidos tendo em visa que não há necessidade de se avaliar todos os testes efetuados.
Obs².: As configurações de hardware utilizados para os testes são as seguintes:
CPU: Intel i7-10750H, RAM: 16gb - 2933mhz, SSD: WD blue 250gb - 2400Mb/s,
SO: Ubuntu 22.04.2 lts, COMPILADOR: GCC v11.3.0.

Podemos notar que o tempo de relógio não difere tanto quando comparado ao tempo de sistema somado ao de usuário. O resultado final de ambos esta diretamente relacionado ao tamanho de entrada fornecida.

5 CONCLUSÃO

Para realização desse trabalho prático foram necessárias algumas, várias, horas de pesquisas, alem de teste e mais testes, muitas frustrações com métodos que não funcionavam do jeito que queríamos mas ao fim de tudo a aplicação saiu como desejamos.

Esse trabalho prático exigiu muito mas também agregou muito conhecimento à todos os membros do grupo.

Referências

[BACKES, 2012] BACKES, A. R. (2012). *Linguagem C completa e descomplicada*. Elsevier.

[BACKES, 2016] BACKES, A. R. (2016). *Estrutura de dados descomplicada em linguagem C*. Elsevier.

[CORMEN et al., 2012] CORMEN, T. H., Leiserson, C., Rivest, R., and Stein, C. (2012). *Algoritmos: teoria e prática*. LTC.

[GNU, 2023] GNU (1993-2023). Calculating elapsed time. `https://www.gnu.org/software/libc/manual/html_node/Calculating-Elapsed-Time.html`.