Universidade Federal de São João del Rei - UFSJ.
Departamento de Ciência da Computação.
Disciplina de Análise de Algoritmos e Estrutura de Dados III.

Aplicação e Análise de Algoritmo para Verificação de Hipercampo.

Autores: Daniel Luiz e Rian Wagner.

Abril, 2023.

1. Introdução:

A verificação de hipercampo é uma junção da geometria espacial com a lógica computacional. É possível usar os conhecimentos sobre hipercampo em áreas como: design de jogos, para criar elementos 3D; na engenharia, para testes matemáticos usados na criação de estruturas. Já na química, pode ser usada para modelagem molecular [1].

2. Formato de entrada e saída:

A entrada é feita com arquivos .txt. O arquivo entrada.txt deve conter na primeira linha, o número de pontos a ser inseridos *(n),*sendo esse contido no intervalo [1,100], e as âncoras *(Xa e Xb)*, contidas no intervalo [0,10000] e Xa < Xb, nas demais linhas, deve ser colocado os *n* pares ordenados *(x,y)*. Cada informação é separada por um espaço entre os dados.

A saída será escrita no terminal de execução e deverá mostrar o número de pontos que antecedem somente na âncoras.

Ex:

entrada.txt:

---

8 3 15
4 4
10 3
11 4
12 5
13 7
8 8
6 8
6 6

---

saída:

7

---

3. Modelagem

Um hipercampo é um conjunto de pontos sem ordenação no plano cartesiano. O problema dos hipercampos (também chamados de hipercubo) vem de uma ideia simples.

Os pontos pertencentes ao plano cartesiano são provenientes de uma sequência infinita de números racionais.

Em um conjunto de *n* pontos colocados no plano cartesiano e duas âncoras, *Xa* e *Xb* fixas, devemos encontrar quantos desses *n* pontos formam uma espécie de triângulo com as âncoras, sem que haja interação com os outros *n-1* pontos ou seus segmentos que também se conectam com as âncoras. Em seguida, dentre os *n* pontos, deve-se verificar qual o máximo de pontos possíveis se interligam somente pelas âncoras.

Para verificar a interação entre os n pontos, foram utilizados pontos auxiliares para as manipulações geométricas e as transformações lineares.

Definida uma reta que passa nas âncoras, podemos ter:

- Ponto simétrico ao *n* analisado, em relação a reta que passa pelas âncoras.
- Ponto de rotação, definido a partir da rotação em - 90 graus do ponto *n* analisado.
- Ponto de reflexão, definido pelo resultado da intersecção entre o ponto *n* e a reta que passa pelas âncoras.

Tais pontos serão verificados nos limites das âncoras para identificar se o ponto *n* está no espaço definido. Isso garante que somente pontos definidos dentro do limite de *Xa* e *Xb* sejam considerados no cálculo do hipercampo.

Caso os pontos estejam definidos dentro do espaço em questão, verifica-se o ponto *n* intercepta somente as âncoras.

4. Estrutura de dados e teste iniciais:

As entradas definidas nas instruções do trabalho e outras cedidas por outros discentes da disciplina foram a base de teste.

A primeira entrada definida nas instruções contém: 4 pontos, Xa = 1, Xb = 10 e os respectivos pontos (2,4); (5,1); (6,5) e (7,8).

A segunda entrada definida nas instruções contém: 2 pontos, Xa = 2, Xb = 2 e os respectivos pontos (3,4) e (7,4).

Com base nesse dados montamos a estrutura de dados para o algoritmo:

```
Struct pontos{
     int x;
     int y;
}
```

As cordenadas (x,y) dos pontos foram definidas nas especificações como inteiros.

5. Funções:

5.1 Abrir, fechar e ler arquivo de entrada:

Como as entradas seriam lidas de uma arquivo .txt, instanciamos uma função *abre_arquivo*, para conseguir buscar e abrir o arquivo *entrada.txt:*

```
void abre_arquivo (char argv[]){
     entrada = fopen(argv, "r");
     if (entrada == NULL){
          printf("Falha na abertura do arquivo. \n")
     }
}
```

que recebe como parâmetro a string *argv[],* a qual identifica o argumento do terminal.

Definimos uma função para fechar o arquivo *entrada.txt:*

```
void fecha_arquivo{
     fclose(entrada);
}
```

Também foi definida a função *scan_arquivo,* para ler os dados no arquivo:

```
int scan_arquivo{
     int x;
     fscanf(entrada,%d,&x);
     return x;
}
```

Para ler os dados de acordo com a ordem definida, a função usada foi a *get_dados*:

```
void get_dados(int *qtdpontos, int *anchorA, int *anchorB) {
    *qtdpontos = scan_arquivo();
    *anchorA = scan_arquivo();
    *anchorB = scan_arquivo();
}
```

A função *get_dados* recebe como parâmetros a quantidade *n* de pontos e as âncoras fixas, *Xa* e *Xb*. Esses dados serão lidos do arquivo entrada.txt, usando a função *scan_arquivo.*

5.2 Função de análise das interações:

A função *math_func* é a responsável por fazer todos os cálculos, transformações lineares, para obtenção dos pontos auxiliares, e verificações.

math_func recebe como parâmetros: o vetor de pontos, alocado dinamicamente; a quantidade de pontos; a âncora Xa e a âncora Xb.

```
int math_func(Pontos *pontos, int qtdpontos, int anchorA, int anchorB) {
```

A função pode ser dividida em três partes:

- 1ª parte: Realiza as transformações lineares para obter os pontos de simetria, de rotação e reflexão.

```
for (int i = 0; i < qtdpontos; i++) {
     int cont = 0;
          for (int j = 0; j < qtdpontos; j++) {
               if (j != i) {
               int xn = pontos[i].x, yn = pontos[i].y;
               int xsimetrico = 2 * xn - anchorA - anchorB;
               int ysimetrico = 2 * yn;
               int xrotacao = xn + yn - ysimetrico;
               int xreflexao = anchorA + anchorB - xrotacao;
```

- 2ª parte: Verifica se o ponto *n* está definido nos limites de *Xa* e *Xb*, utilizando os pontos auxiliares

```
     if ((xn < anchora && xsimetrico < anchora && xrotacao < anchora && xreflexao < anchora) || (xn > anchorb && xsimetrico > anchorb && xrotacao > anchorb && xreflexao > anchorb)) {
          continue;
     }
     if ((xn < anchora && xsimetrico < anchora && xrotacao >= anchora && xreflexao >= anchora) || (xn > anchorb && xsimetrico > anchorb && xrotacao <= anchorb && xreflexao <= anchorb)) {
```

```
            continue;
        }



        if ((xn < anchora && xsimetrico >= anchora && xrotacao <
        anchora && xreflexao >= anchora) || (xn > anchorb &&
        xsimetrico <= anchorb && xrotacao > anchorb && xreflexao
        <= anchorb)) {
            continue;
        }
        if ((xn < anchora && xsimetrico >= anchora && xrotacao >=
        anchora && xreflexao >= anchora) || (xn > anchorb &&
        xsimetrico <= anchorb && xrotacao <= anchorb && xreflexao
        <= anchorb)) {
            continue;
        }
```

- 3ª parte: Verifica se o ponto *n* intercepta somente as âncoras Xa e Xb.

```
int intersecta_apenas_ancoras = 1;
        for (int k = 0; k < qtdpontos; k++) {
            if (k != i && k != j) {
                int xk = pontos[k].x, yk = pontos[k].y;
                    if ((yk > yn && yk > ysimetrico) || (yk <
                    yn && yk < ysimetrico)) {
                        if ((xk < xn && xk < xsimetrico) ||
                        (xk > xn && xk > xsimetrico)) {
                        intersecta_apenas_ancoras = 0;
                        break;
                        }
                    }
            }
        }
        if (intersecta_apenas_ancoras) {
            cont++;
        }
    }
 }

if (cont > pconect) {
             pconect = cont;
```

```
        }
    }
return pconect;
}
```

O retorno da função é a quantidade máxima de pontos que se interceptam somente nas âncoras.

5.3 Função Main:

A Main responsável por inicializar as variáveis, chamar as funções e executar o código, começa chamando a função *abre_arquivo*, inicializando as entradas, lendo as entradas do arquivo através da função *get_dados.*

Os pontos lidos são alocados dinamicamente em um array do tipo *Pontos.*

Chama a função *math_func* dentro de uma variável que será exibida com o resultado.

O array de pontos dinamicamente alocados é liberado, o arquivo de entrada é fechado e se encerra o programa.

Também foi utilizado as funções *gettimeofday* e *getrusage* para a medição de tempo e uso tanto do sistema quanto do usuário.

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

    struct timeval start, end;
    struct rusage usage;
    gettimeofday(&start, NULL);

    abre_arquivo(argv[2]);

    int qtdpontos = 0, anchorA = 0, anchorB = 0;
    get_dados(&qtdpontos, &anchorA, &anchorB);

    Pontos *pontos;
    pontos = malloc(qtdpontos * sizeof(Pontos));

    for (int i = 0; i < qtdpontos; i++) {
    pontos[i].x = scan_arquivo();
    pontos[i].y = scan_arquivo();
    }

    int answer = math_func(pontos, qtdpontos, anchorA,
anchorB);
```

```
    printf("\nA quantidade maxima de pontos eh: %d\n",
answer);

    free(pontos);
    fecha_arquivo();

    gettimeofday(&end, NULL);
    getrusage(RUSAGE_SELF, &usage);

    printf("Tempo de execução: %ld microsegundos\n",
(((end.tv_sec - start.tv_sec) * 1000000) + (end.tv_usec -
start.tv_usec)));
    printf("Tempo de usuário: %ld microsegundos\n",
usage.ru_utime.tv_usec);
    printf("Tempo de sistema: %ld microsegundos\n",
usage.ru_stime.tv_usec);

}
```

---

6. Análise de complexidade:

6.1 Operações:

No programa há basicamente três tipos de operações:

1. Operações de entrada e saída (E/S): leitura do arquivo de entrada e escrita do arquivo de saída.
2. Operações aritméticas: realizadas dentro da função *math_func*, que consistem principalmente em cálculos de distância e ângulo entre pontos.
3. Controle de fluxo: utilizado para controlar a execução do programa, incluindo chamadas de funções e iterações (loops).

Essas operações são realizadas repetidamente ao longo da execução do programa, e é a combinação delas que determina a complexidade de tempo deste código.

6.2 Análise assintótica:

Para determinar a complexidade do tempo do código, é necessário analisar quanto tempo leva para executar o pior caso em relação ao tamanho da entrada.

O código possui duas funções principais: *get_dados* e *math_func*. A função *get_dados* chama a função *scan_arquivo* três vezes, que tem complexidade de tempo constante O(1) para cada chamada, portanto, a complexidade dessa função é O(1).

Já a função *math_func* possui dois loops aninhados. O loop externo executa *qtdpontos* iterações, enquanto o loop interno executa *qtdpontos - 1* iterações em cada iteração do loop externo. Dentro do loop interno, há operações de tempo constante O(1) e um loop adicional que executa *qtdpontos* iterações. Dessa forma, a complexidade de tempo da função *math_func* é $O(n^3)$ onde *n* é a quantidade de pontos (*qtdpontos*). Como *math_func* é a única função que fará diferença na execução, a complexidade de tempo do programa no pior caso é $O(n^3)$.

6.3 Tempo de execução:

Ambiente de compilação e execução : "WSL 2 Ubuntu 22.04 LTS"
Compilador : MinGW gcc
Processador : "Ryzen 5 5600H" e "Intel(R) Core(TM) i5-7200U " .

Cada teste foi executado 3 vezes. Os tempos de execução foram obtidos a partir da média aritmética de cada execução.

Obs: 'ms' = microsegundos.

Tabela tempo de execução

Tamanho da entrada (N)	Tempo de execução	Tempo do Usuário
N = 2	43ms	278ms
N = 4	44ms	313ms
N = 8	53ms	309ms
N = 10	74ms	395ms
N = 50	78ms	338ms
N = 99	226ms	499ms

8. Bibliotecas:

Para o funcionamento do programa, é necessário incluir as seguintes bibliotecas:

- `#include 'stdio.h'`
- `#include 'stdlib.h'`
- `#include 'sys/time.h'`
- `#include 'sys/resource.h'`

Referências:

[1] Melo, E.S.N.; Melo, J.R.F. Softwares De Simulação No Ensino De Química Uma Representação Social Na Prática Docente Educação Temática Digital, v.6, n.2, p.43-52, 2005.