Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

DCC004 - ALGORITMOS E ESTRUTURAS DE DADOS II - TURMA TW

Trabalho Prático 0

Guilherme Saulo Alves

Professora - Raquel Cardoso de Melo

1. Introdução

O Trabalho Prático 0 possui como objetivo revisar conceitos básicos de programação em C. O trabalho exige conhecimento em alocação dinâmica , manipulação de *strings*, apontadores, conceitos de tipos abstratos de dados e analise de complexidade.

O proposito do Trabalho consiste em implementar estruturas em um tipo abstrato de dados “geometria” para representar um ponto no espaço. Nesse TAD deverá armazenar as coordenadas x e y do ponto. Com uma sequencia desses pontos são criadas linhas poligonais e polígonos no espaço. Quatro funções implementadas no TAD que realizam processamento geométrico.

2. Implementação

As primeiras implementações foram das funções mais simples, começando pelas estruturas ponto, linha e polígono. Cada função e estrutura serão explicadas detalhadamente ao longo da documentação.

2.1. Considerações Iniciais

O ambiente de desenvolvimento do código foi no CodeBlocks IDE 10.05 no Sistema Operacional Ubuntu Linux 12.04. Foi usado o compilador GNU GCC Compiler via linha de comando.

O código está dividido em três arquivos principais: *main.c* com o programa principal, *geometria.c* e *geometria.h.* com a implementação do TAD Geometria.

2.2. Estruturas e Funções

As estruturas e funções que formam o TAD serão apresentadas a seguir:

2.2.1 Estrutura *ponto*

A estrutura é formada por duas variaveis do tipo *double x* e *y* que representam as coordenadas dos pontos para os testes geométricos.

2.2.2 Estrutura *linha*

A estrutura é formada por uma variável *numVertices* do tipo inteiro que armazena a quantidade de vértices de uma linha e por um vetor estático do tipo *ponto* que grava as coordenadas de cada vértice da linha.

2.2.3 Estrutura *poligono*

A estrutura é formada por uma variável *numVertices* do tipo inteiro que armazena a quantidade de vértices de um polígono e por um vetor estático do tipo *ponto* que grava as coordenadas de cada vértice do polígono.

2.2.4 void criaPonto(ponto *p, double x, double y);

A função recebe como parâmetro um ponteiro para o TAD ponto e os valores x e y (lidos do aqrquivo) que serão repassados para a função e modicarão internamente os valores do TAD.

2.2.5 int pontoCoincide(ponto P, ponto Q);

A função recebe como parâmetro dois valores de pontos do tipo *ponto.* Tem como objetivo retornar 1 se os pontos forem iguais ou 0 caso contrario.

2.2.6 void imprimePonto(ponto P);

A função recebe como parâmetro um valor de ponto e imprime na tela seu valor.

2.2.7 void criaLinha(linha *l, int numVertices, ponto *vertices);

A função recebe como parâmetro um ponteiro do tipo *linha*, o numero de vértices do tipo inteiro e um ponteiro do tipo *ponto*. O objetivo dessa função é copiar as coordenadas da variável vértices para a linha na posição numVertices. Essa função é chamada **n** vezes no main, no qual n é a posição do vértice.

2.2.8 int linhaInterceptaPoligono(linha L, poligono P);

A função tem como parâmetro uma *linha* e um *poligono* com seus respectivos sequencia de vértices. Tem como objetivo retornar 1 caso a linha intercepta o polígono ou 0 caso contrario. Supondo um segmento de reta entre os pontos i e i+1 e outro entre os pontos j e j+1, podemos achar o

ponto de interseção entre dois segmentos de reta usando a equação paramétrica das retas. Calculamos o determinante entre os segmentos de reta e as variáveis s e t. Se o valor de s e t estiverem entre [0 e 1), então a interseção dos segmentos existe, senão não. A descrição completa do algoritmo utilizado pode ser encontrada aqui.

2.2.9 int linhaSimples(linha L);

A função tem como parâmetro uma *linha* com seus respectivos sequencia de vértices. A idéia da função é a mesma do item 2.2.9.

2.2.10 void criaPoligono(poligono *p, int numVertices, ponto *vertices);

A função recebe como parâmetro um ponteiro do tipo *poligono*, o numero de vértices do tipo inteiro e um ponteiro do tipo *ponto*. O objetivo dessa função é copiar as coordenadas da variável vértices para o poligono na posição numVertices. Essa função é chamada **n** vezes no main, no qual n é a posição do vértice.

2.2.11 int pontoEmPoligono(ponto P, poligono Pol);

A função recebe como parâmetro um ponto e um poligono com suas sequencias de vértices. O objetivo é extender a coordenada x do ponto e criar um segmento de reta. Depois calcular a interseção com cada segmento do poligono e contar quantas vezes ocorre interseção. Se for impar, o ponto esta dentro do poligono e retorna 1, senão o ponto esta fora e retorna 0. A ideia de calcular a interseção é a mesma dos itens anteriores.

2.2.12 int poligonoSimples(poligono Pol);

A função tem como parâmetro um *poligono* com seus respectivos sequencia de vértices. A idéia da função é a mesma do item 2.2.9.

2.3 Programa Principal

O programa recebe um arquivo de dados *entrada.txt* contendo a geometria de pontos, linhas e polígonos e executa testes entre eles chamando o TAD geometria e gera o resultado da saída stdout.

O primeiro passo depois da declaração das variáveis é ler o numero de pontos e posteriormente alocar dinamicamente espaço na memoria para a quantidade de pontos a ser lida. Seguindo é feito a leitura das coordenadas x e y e chamada a função cria ponto n vezes, tal que n é o numero de pontos.

Depois é feita um procedimento parecido para ler a linha do arquivo. Primeiro lemos o numero de linhas, alocamos dinamicamente espaço na memoria para a quantidade de linhas a ser lida. Depois lemos a quantidade de vértices da primeira linha e alocamos dinamicamente espaço na memoria para de vértices da linha. Lemos cada coordenada x e y posteriormente chamamos a função cria ponto e cria linha. A chamada das funções é feita n x m vezes tal que n é o numero de linhas e m o numero de vértices. O procedimento de criar polígono é idêntico a criar linha, só muda o nome das variáveis.

No final do programa é feito os testes. É lida do arquivo a quantidade de testes e alocamos dinamicamente espaço na memoria para a quantidade de teste a ser lida. Lemos cada string, armazenamos na memoria e comparamos se os nomes são iguais. Feito os testes, liberamos memoria alocada e fechamos os arquivos.

3. Análise de Complexidade

A análise de complexidade será feita em função da variável n que representa o número de vezes que a função é chamada.

Função criaPonto: Essa função é bem simples e não envolve nenhuma laço de repetição. Portanto, sua complexidade é constante dada por O(c) tal que c é uma constante qualquer. As instruções do algoritmo são executadas um número fixo de vezes. Logo, no fim das contas sua complexidade é dada por **O(1)**.

Função pontoCoincide: Possui um if e um else que retorna 1 ou 0. No melhor caso os pontos são iguais e apenas um uma comparação é feita. Portanto O(1). No pior caso, o não são iguais, ou seja, 2O(1). Logo a função é O(1)+2O(1) = **O(1)**.

Função imprimePonto: As instruções do algoritmo são executadas uma única vez. Portanto **O(1)**.

Função criaLinha: As instruções do algoritmo são executadas um número fixo de vezes. Complexidade **O(1)**;

Função linhaInterseptaPoligono: A função tem um for dentro do outro. No segundo tem um if O(1). Como essa função é executada n ao quadrado vezes, sua complexidade será $O((n+1)^2)$ = **$O(n^2)$**.

Função linhaSimples: Essa função tem ordem de complexidade igual a ordem de complexidade da função linhaInterseptaPoligono, pois essas funções possuem 2 for aninhados. Portanto a complexidade será **$O(n^2)$**.

Função criaPoligono: As instruções do algoritmo são executadas somente duas vezes. Complexidade **O(1)**;

Função pontoEmPoligono: A função executa alguns comandos O(1) e depois entra em um loop que é executado n vezes. Dentro do loop, são executados 4 comandos O(1). A complexidade até aqui é O(1)+O(n+4) = O(n). Posteriormente, possui um if e um else que retorna 1 ou 0. No melhor caso apenas uma comparação é feita. Portanto O(1). No pior caso duas comparações são feitas, logo 2O(1). Logo a função é O(n)+O(1)+2O(1) = **O(n)**.

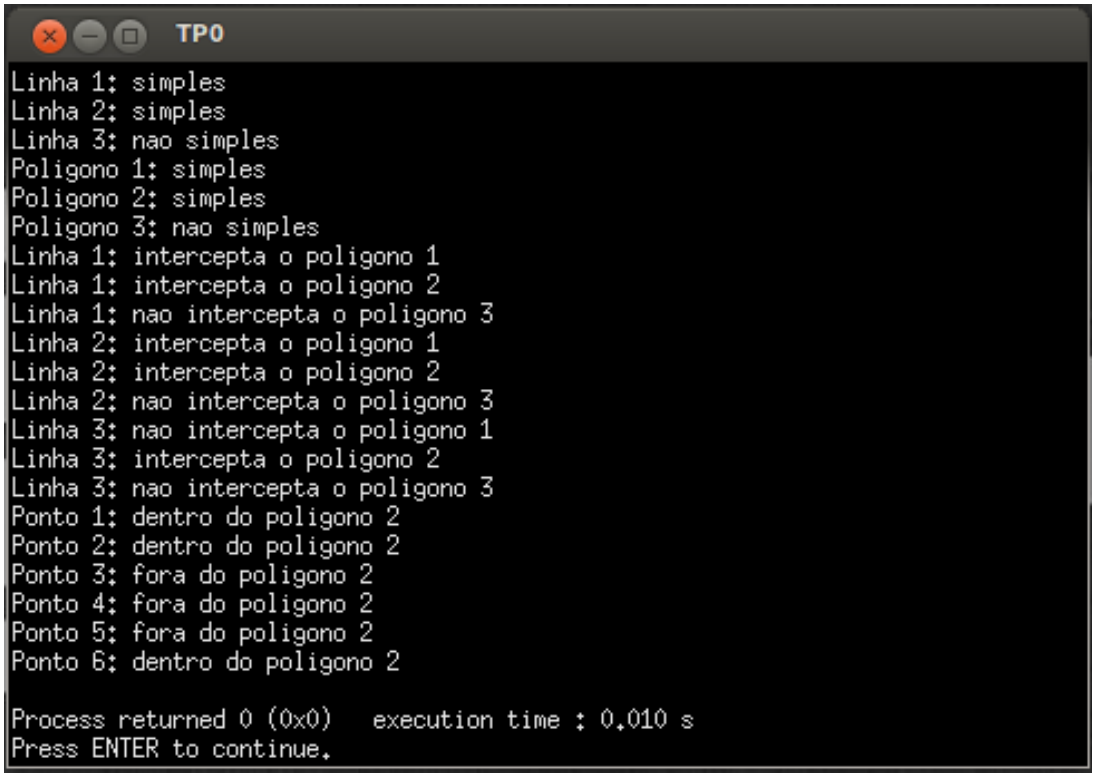
Função poligonoSimples: Complexidade identica da função criaLinha, pois essas funções possuem 2 for aninhados. Portanto a complexidade será **$O(n^2)$**.

Programa principal: Dividimos o calculo da complexidade em 4 partes. O programa principal executa alguns comandos O(1) antes de aparecer o primeiro ‘for’ para chamar a função cria ponto n vezes. A complexidade da parte 1 é O(1)+O(n(O(1))) = O(n). Depois, temos mais alguns comando O(1) e outro 'for' responsável por criar uma linha que executa 3 comando O(1) n vezes com um segundo for aninhado que chama a função cria ponto e cria polígono n vezes. Logo a complexidade da parte 2 é, $O(3nO(1)*n(O(1)+O(1)) = O(n^2)$. Na parte três temos outro 'for' que cria polígono e possui complexidade igual a primeira parte, $O(n^2)$, pois ambos chamam as mesmas funções e possuem 'for' aninhado. Na parte 4 temos a realização dos testes, com alguns comando O(1) mais um 'for' com 4 condições. No melhor caso possuímos somente a execução do ‘if’ que

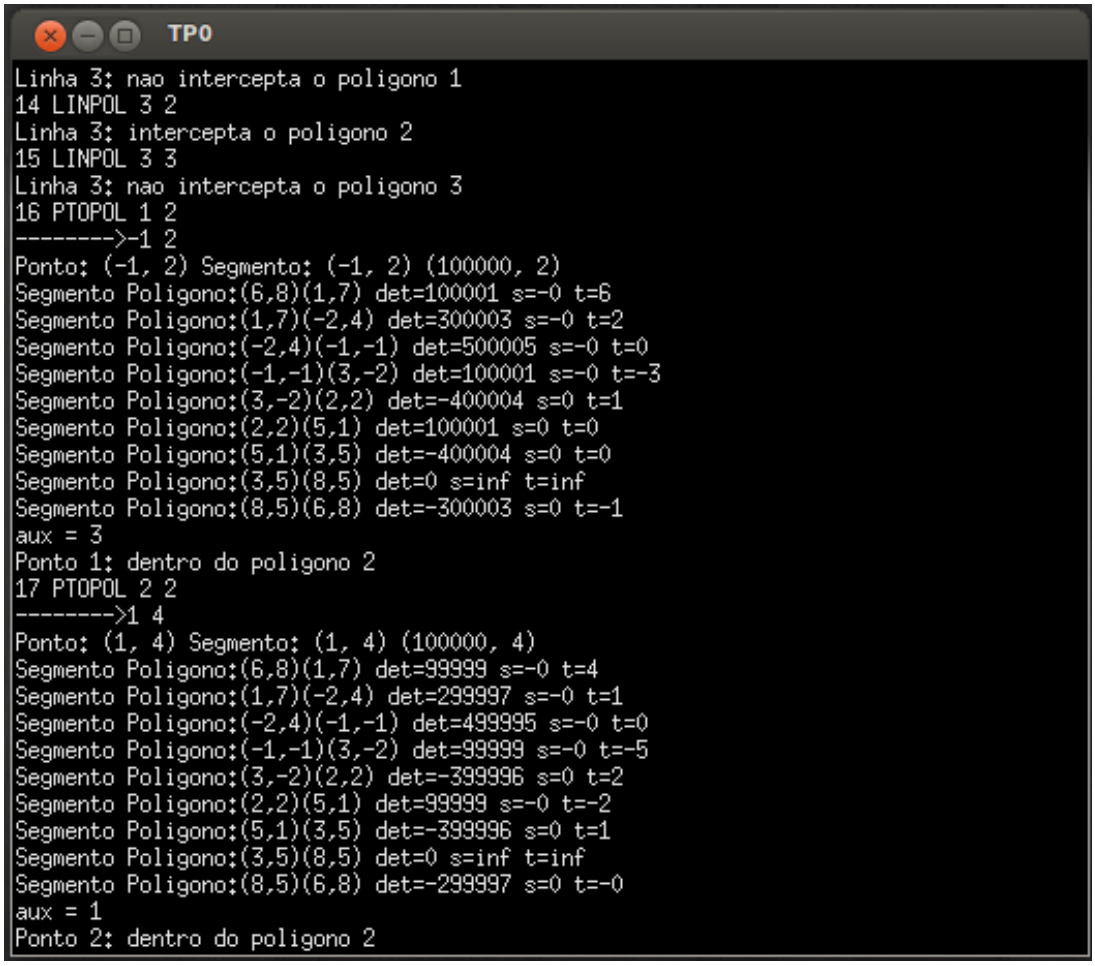
possui complexidade $O(n(O(n^2))) = O(n^3)$. No pior caso, teremos a execução do ‘else’ mais caro, que é o primeiro ou segundo ‘else’, com $O(n^2)$ de complexidade. Logo a complexidade da parte 4 é $O(n^3) + O(n^2) = O(n^3)$. Finalizando, a ordem de complexidade de tempo e espaço total do programa é dada pela maior ordem de complexidade existente entre as partes, logo **$O(n^3)$**.

4. Testes

Vários testes foram realizados com o programa de forma a verificar o seu funcionamento. Os testes foram realizados em um Atom CPU D525, com 2 Gb de memória. A figura abaixo mostra a saída de uma execução típica.



Também foi realizados testes durante a elaboração do trabalho com ‘printf’ para verificar os valores de entrada e saída de algumas variáveis.



5. Conclusão

Com elaboração desse trabalho foi possível relembrar as técnicas de programação aprendidas na disciplina de AEDS I e também fixar os novos conceito de TADs e analise de complexidade. Além disso, alguns conceitos de geometria analítica plana foram relembrados.

A principal dificuldade encontrada foi na implementação de um algoritmo para calcular interseção de dois segmentos de reta. As ideias e fórmulas foram pesquisadas para o desenvolvimento de algumas partes do programa.

6. Referências

[1] Prof. Dr. Márcio Sarroglia Pinho - Computação Gráfica I - http://www.inf.pucrs.br/~pinho/CG/Aulas/OpenGL/Interseccao/CalcIntersec.html , visitado em 27/08/2013.