**Triangulação**

Caio de Freitas Valente – 6552442

# Problema:

Três soluções diferentes para triangulação de polígonos:

* Remoção de orelhas
* Algoritmo para polígonos Y-monótonos
* Algoritmo de Lee-Preparata

Para que os algoritmos funcionem corretamente há duas suposições:

* Polígono de entrada é simples
* Os vértices do polígono de entrada são dados no sentido anti-horário

Há ainda a suposição de que o polígono passado para o algoritmo Y-monótono é de fato Y-monótono, ou seja, não há verificação de erros.

# Arquivos:

Há um arquivo do projeto original que foi modificado:

* geocomp/\_\_init\_\_.py 🡪 Para inserção das funções de triangulação.

Os arquivos novos estão contidos na pasta geocomp/triangulation. Os arquivos novos são:

* \_\_init\_\_.py
* avl.py
* dcel.py
* stack.py
* mergesort.py
* brute.py
* monotone.py
* leepreparata.py

Há ainda alguns arquivos de testes na pasta Dados/teste/

Todos polígonos simples com arestas no sentido anti-horário.

# Estruturas de dados:

Para a implementação do projeto foram necessárias três estruturas de dados:

* Arvore binária balanceada – Foi implementada uma AVL – avl.py
* Doubly connected edge list – dcel.py
* Pilha – stack.py

# Consumo de tempo:

ABB: Inserções, buscas e remoções consomem tempo O(logn)

Pilha: Inserção e remoção consomem tempo O(1)

Douby connected edge list: Inserção de uma aresta em uma face consome tempo O(f) – sendo f o número de arestas na face do polígono

Mergesort: O(nlogn)

Algoritmo força bruta usando orelhas: O(n²)

Algoritmo para polígonos y-monótonos: O(n)

Algoritmo Lee-Preparata:

Seja:

* P - Numero de pontas interiores
* f - Numero de elementos em uma face

A implementação consome tempo um pouco superior a O(nlogn), Isso se deve ao fato de inserções na estrutura de dados dcel consumir tempo O(f). Essa inserção leva tempo linear pois mantemos para cada aresta o índice da face a que pertence. Como temos p pontas interiores, então o consumo de tempo para inserção de diagonais é de O(pf).

Além disso utilizamos uma estrutura auxiliar baseada nas arestas, uma lista de arestas indexada por seu ponto de origem**\***, isso é usado para que inserções de arestas com um sejam feitas sempre na mesma face. – od 🡪 orderedDCEL

Essa busca consome tempo O(1), afinal sabemos que há inserção de arestas adicionais em um ponto no máximo duas vezes, totalizando no pior dos casos três arestas com origem no em um dado vértice. Obs: Face 0, que representa a parte externa do polígono é ignorada.

Logo o consumo de tempo total é: O(nlogn + pf)

**\***Exemplo da estrutura od:

Seja p[i] = (20, 30)

Suponha que há duas arestas, e1 e e2, que tem como origem p[i].

e1.origin = (20,30)

e1.destiny = (42, 50)

e2.origin = (20,30)

e2.destiny = (30, 10)

Então od[i] = {e1, e2}

# Observações:

No algoritmo de remoção de orelhas usamos um vetor de Python. Vetores em Python tem um comportamento similar ao de listas, logo aproveitamos essa característica para fazermos a remoção dos vértices que já geraram diagonais.

# Cores usadas na animação:

* Branco 🡪 O polígono base.
* Vermelho 🡪 Diagonais.

## Remoção de orelhas:

* Amarelo 🡪 Ponto que está sendo verificado para determinar se é orelha.
* Verde 🡪 Orelha.
* Ciano 🡪 A orelha que está gerando uma diagonal e causando a verificação de orelha nos dois vértices adjacentes.

## Y-Monótono:

* Amarelo 🡪 Ponto que está sendo verificado – ponto evento.
* Verde 🡪 Topo da pilha – st.
* Ciano 🡪 Primeiro elemento da pilha – s1

## Lee-Preparata:

* Amarelo 🡪 Ponto que está sendo verificado – ponto evento.
* Verde 🡪 Ponto superior e arestas do trapézio acima do ponto que está sendo verificado.
* Ciano 🡪 Ponto superior e arestas do segundo trapézio acima do ponto que está sendo verificado.

# Desenvolvimento:

O projeto foi desenvolvido e testado usando Python 2.7.8 – 64bits para Windows 8.1

A parte gráfica foi testada usando tkgeocomp.py