Trabalho Prático I - Algoritmos II

Bernardo Amorim
Matrícula: 2019006469
bernardoamorim@dcc.ufmg.br

Repositório no GitHub:
`https://github.com/bernardoamorim/trabalho_pratico_I_algoritmos_II`
Link clicável para o repositório

1 Algoritmos e Implementação

Para resolver o problema proposto, segui os princípios da orientação à objetos e encapsulamento, criando duas classes fundamentais: `Point` e `Polygon`.

1.1 `Point`

Nessa classe implementei apenas um container com as duas coordenadas (x e y) e três operações fundamentais, $+, -, \times$ i.e. soma coordenada à coordenada, subtração coordenada à coordenada e produto vetorial. Além disso usei essa classe em três primitivas geométricas, sendo elas:

- `clockwise(a, b, c)`, retorna (usando o algoritmo visto em sala) se os pontos a, b, c estão em sentido horário.
- `counter_clockwise(a, b, c)`, retorna (usando o algoritmo visto em sala) se os pontos a, b, c estão em sentido anti-horário.
- `inside_triangle(p, a, b, c)` retorna, por meio do algoritmo visto em aula e com o auxílio das primitivas acima se o ponto p está dentro do triângulo $\triangle abc$.
- `share_side(T, U)` retorna se $\triangle T$ (dado como uma `list` de 3 `Point`s) compartilha alguma aresta com $\triangle U$.

1.2 `Polygon`

Essa classe é formada por uma `list` (da biblioteca padrão do Python) de elementos do tipo `Point`, os vértices do polígono dados em sentido anti-horário. Além disso, a classe tem um método, `triangulate()`:

- `P.triangulate()` retorna, por meio do algoritmo de ear-clipping, uma lista de triângulos correspondente à triangulação do polígono P. A implementação do algoritmo segue o que foi visto em aula e não usa nenhuma estrutura de dados

além da `list`, dado que para sua execução basta ser capaz de deletar elementos e checar, por meio das primitivas de ponto, se sua orientação é válida e se há algum vértice dentro do triângulo investigado.

1.3 Resolvendo o problema da galeria de arte

Além das classes indicadas acima, foram usadas algumas funções para resolver o problema:

- `dual_graph(triangulation)` recebe uma lista de triângulos (como a retornada pelo método `Polygon.triangulate()` e retorna a lista de adjacência do grafo dual da triangulação. Para calculá-lo basta usar a primitiva `share_side`.
- `color_vertices(P)` retorna uma lista com a 3-coloração dos vértices do polígono P. Para fazê-la basta criar o grafo dual da triangulação de P (as funções que o executam já foram definidas) e, em seguida, fazer uma lista em profundidade (feita de maneira iterativa usando um `list` como uma pilha), colorindo os vértices gulosamente como foi definido em aula (sempre escolhendo as cores que restam para 3-colorir cada triângulo).

Agora que temos todas as funções necessárias para a execução do código, basta escolher a menor partição dentre as 3 induzidas pela 3-coloração do grafo dual da triangulação:

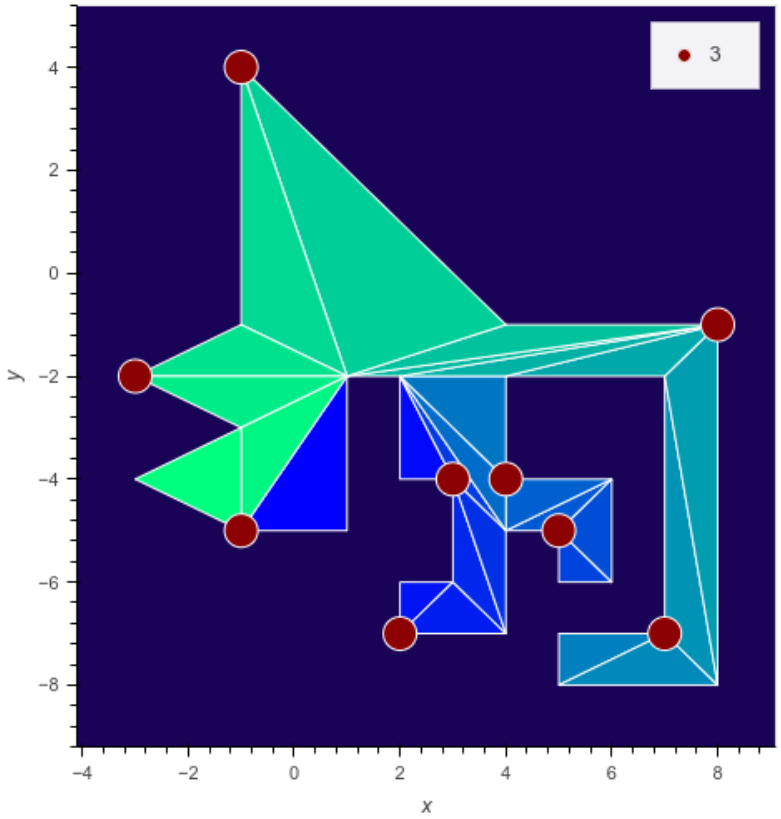
- `art_gallery(P)` escolhe, por meio das funções definidas acima, a menor dentre as 3 partições que cobrem a triangulação e plota os resultados.

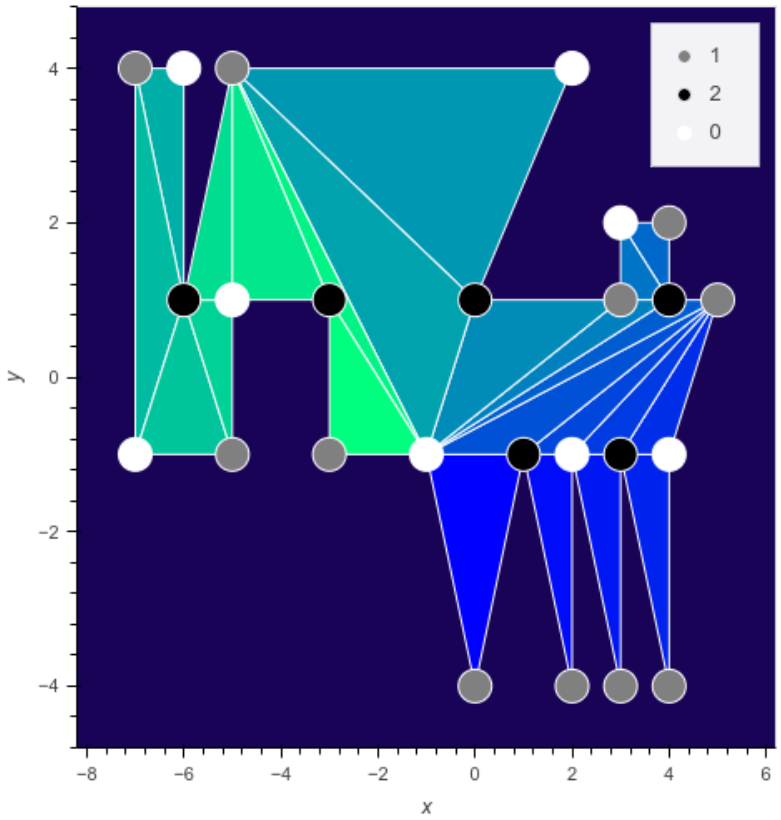
1.4 Gráficos

Além das funções definidas acima foram criadas várias funções auxiliares para plot das figuras (como foi pedido), não discutirei suas implementações pois não são interessantes do ponto de vista algorítmico e envolvem apenas uso das bibliotecas HoloViews e Bokeh.

2 Testes

Criei alguns testes na mão e os usei como exemplo recorrente para o funcionamento de todas as funções. Para garantir sua efetividade tentei criar polígonos bem variados, abusando da não-convexidade de suas estruturas para tentar provocar uma quebra no código. Todos estes testes estão bem documentados e explicados no notebook com a solução, segue imagens da triangulação de dois destes testes:


3
4
2
0
−2
−4
−6
−8
y
−4
−2
0
2
4
6
8
x


1
2
0
4
2
0
−2
−4
y
−8
−6
−4
−2
0
2
4
6
x