sect

Release 7.1.0

Azat Ibrakov

May 27, 2023

CONTENTS

Note: If object is not listed in documentation it should be considered as implementation detail that can change and should not be relied upon.

CHAPTER

ONE

DECOMPOSITION MODULE

class sect.decomposition.**Graph**(*root: Node*)

Represents trapezoidal decomposition graph.

classmethod from_multisegment(*multisegment: ~ground.hints.Multisegment, *, shuffler: ~typing.Callable[[~typing.MutableSequence], None] = <bound method Random.shuffle of <random.Random object>>, context: ~ground.base.Context*) → *Graph*

Constructs trapezoidal decomposition graph of given multisegment.

Based on incremental randomized algorithm by R. Seidel.

Time complexity:
`O(segments_count * log segments_count)` expected, `O(segments_count ** 2)` worst

Memory complexity:
`O(segments_count)`

where `segments_count = len(multisegment.segments)`

Reference:
https://doi.org/10.1016%2F0925-7721%2891%2990012-4 https://www.cs.princeton.edu/courses/archive/fall05/cos528/handouts/A%20Simple%20and%20fast.pdf

Parameters

- **multisegment** – target multisegment.
- **shuffler** – function which mutates sequence by shuffling its elements, required for randomization.
- **context** – geometric context.

Returns
trapezoidal decomposition graph of the multisegment.

```
>>> from ground.base import get_context
>>> context = get_context()
>>> Multisegment, Point, Segment = (context.multisegment_cls,
...                                 context.point_cls,
...                                 context.segment_cls)
>>> graph = Graph.from_multisegment(
...     Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                   Segment(Point(0, 0), Point(0, 1))]),
...     context=context
```

(continues on next page)

(continued from previous page)

```
... )
>>> Point(1, 0) in graph
True
>>> Point(0, 1) in graph
True
>>> Point(1, 1) in graph
False
>>> graph.locate(Point(1, 0)) is Location.BOUNDARY
True
>>> graph.locate(Point(0, 1)) is Location.BOUNDARY
True
>>> graph.locate(Point(1, 1)) is Location.EXTERIOR
True
```

classmethod from_polygon(*polygon: ~ground.hints.Polygon, *, shuffler: ~typing.Callable[[~typing.MutableSequence], None] = <bound method Random.shuffle of <random.Random object>>, context: ~ground.base.Context*) → *Graph*

Constructs trapezoidal decomposition graph of given polygon.

Based on incremental randomized algorithm by R. Seidel.

Time complexity:
`O(vertices_count * log vertices_count)` expected, `O(vertices_count ** 2)` worst

Memory complexity:
`O(vertices_count)`

where

```
vertices_count = (len(polygon.border.vertices)
                  + sum(len(hole.vertices)
                        for hole in polygon.holes)
                  + len(extra_points) + len(extra_constraints))
```

Reference:
https://doi.org/10.1016%2F0925-7721%2891%2990012-4 https://www.cs.princeton.edu/courses/archive/fall05/cos528/handouts/A%20Simple%20and%20fast.pdf

Parameters

- **polygon** – target polygon.
- **shuffler** – function which mutates sequence by shuffling its elements, required for randomization.
- **context** – geometric context.

Returns
trapezoidal decomposition graph of the border and holes.

```
>>> from ground.base import get_context
>>> context = get_context()
>>> Contour, Point, Polygon = (context.contour_cls, context.point_cls,
...                            context.polygon_cls)
```

(continues on next page)

(continued from previous page)

```
>>> graph = Graph.from_polygon(
...     Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                      Point(0, 6)]),
...             [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                       Point(4, 2)])]),
...     context=context
... )
>>> Point(1, 1) in graph
True
>>> Point(2, 2) in graph
True
>>> Point(3, 3) in graph
False
>>> graph.locate(Point(1, 1)) is Location.INTERIOR
True
>>> graph.locate(Point(2, 2)) is Location.BOUNDARY
True
>>> graph.locate(Point(3, 3)) is Location.EXTERIOR
True
```

property height: int

Returns height of the root node.

locate(*point: Point*) → Location

Finds location of point relative to decomposed geometry.

Time complexity:
`O(self.height)`

Memory complexity:
`O(1)`

CHAPTER

TWO

TRIANGULATION MODULE

class `sect.triangulation.QuadEdge`(*start: Optional[Point] = None*, *left_from_start: Optional[QuadEdge] = None*, *rotated: Optional[QuadEdge] = None*, ***, *context: Context*)

Based on:
quad-edge data structure.

Reference:
https://en.wikipedia.org/wiki/Quad-edge http://www.sccg.sk/~samuelcik/dgs/quad_edge.pdf

classmethod from_endpoints(*start: Point*, *end: Point*, ***, *context: Context*) → *QuadEdge*

Creates new edge from endpoints.

property end: Point

aka "Dest" in L. Guibas and J. Stolfi notation.

property left_from_end: *QuadEdge*

aka "Lnext" in L. Guibas and J. Stolfi notation.

property left_from_start: *QuadEdge*

aka "Onext" in L. Guibas and J. Stolfi notation.

property opposite: *QuadEdge*

aka "Sym" in L. Guibas and J. Stolfi notation.

property right_from_end: *QuadEdge*

aka "Rprev" in L. Guibas and J. Stolfi notation.

property right_from_start: *QuadEdge*

aka "Oprev" in L. Guibas and J. Stolfi notation.

property rotated: *QuadEdge*

aka "Rot" in L. Guibas and J. Stolfi notation.

property start: Point

aka "Org" in L. Guibas and J. Stolfi notation.

connect(*other:* QuadEdge) → *QuadEdge*

Connects the edge with the other.

delete() → None

Deletes the edge.

orientation_of(*point: Point*) → Orientation

Returns orientation of the point relative to the edge.

splice(*other: QuadEdge*) → None

Splices the edge with the other.

swap() → None

Swaps diagonal in a quadrilateral formed by triangles in both clockwise and counterclockwise order around the start.

class sect.triangulation.**Triangulation**(*left_side: QuadEdge*, *right_side: QuadEdge*, *context: Context*)

Represents triangulation.

classmethod **constrained_delaunay**(*polygon: Polygon*, *, *extra_constraints: Sequence[Segment] = ()*, *extra_points: Sequence[Point] = ()*, *context: Context*) → *Triangulation*

Constructs constrained Delaunay triangulation of given polygon (with potentially extra points and constraints).

Based on

- divide-and-conquer algorithm by L. Guibas & J. Stolfi for generating Delaunay triangulation,
- algorithm by S. W. Sloan for adding constraints to Delaunay triangulation,
- clipping algorithm by F. Martinez et al. for deleting in-hole triangles.

Time complexity:
`O(vertices_count * log vertices_count)` for convex polygons without extra constraints, `O(vertices_count ** 2)` otherwise

Memory complexity:
`O(vertices_count)`

where

```
vertices_count = (len(polygon.border.vertices)
                  + sum(len(hole.vertices)
                        for hole in polygon.holes)
                  + len(extra_points) + len(extra_constraints))
```

Reference:
http://www.sccg.sk/~samuelcik/dgs/quad_edge.pdf https://www.newcastle.edu.au/__data/assets/pdf_file/0019/22519/23_A-fast-algortithm-for-generating-constrained-Delaunay-triangulations.pdf https://doi.org/10.1016/j.advengsoft.2013.04.004 http://www4.ujaen.es/~fmartin/bool_op.html

Parameters

- **polygon** – target polygon.
- **extra_points** – additional points to be presented in the triangulation.
- **extra_constraints** – additional constraints to be presented in the triangulation.
- **context** – geometric context.

Returns
triangulation of the border, holes & extra points considering constraints.

classmethod delaunay(*points: Sequence[Point]*, ***, *context: Context*) → *Triangulation*

Constructs Delaunay triangulation of given points.

Based on divide-and-conquer algorithm by L. Guibas & J. Stolfi.

Time complexity:
`O(len(points) * log len(points))`

Memory complexity:
`O(len(points))`

Reference:
http://www.sccg.sk/~samuelcik/dgs/quad_edge.pdf

Parameters

- **points** – 3 or more points to triangulate.
- **context** – geometric context.

Returns
triangulation of the points.

delete(*edge: QuadEdge*) → None

Deletes given edge from the triangulation.

triangles() → List[Contour]

Returns triangles of the triangulation.

PYTHON MODULE INDEX

S

INDEX