Generation of polygonal meshes in compact space

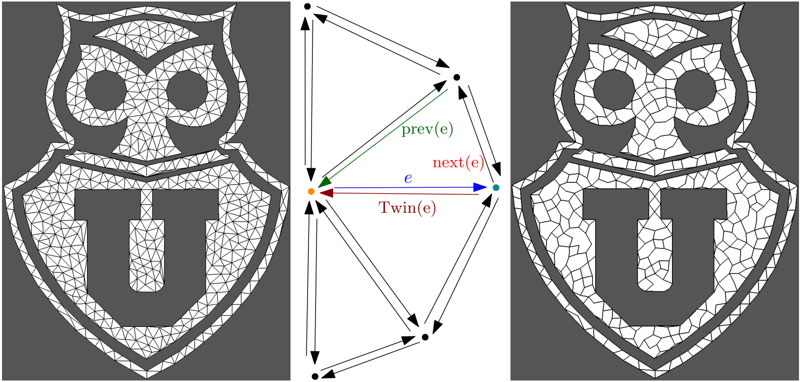
Sergio Salinas-Fernández
José Fuentes-Sepúlveda
Nancy Hitschfeld-Kahler

SIAM International Meshing Roundtable Workshop 2023

March 7, 2023

Abstract

We will show how to encapsulate a compact data structure, for polygonal mesh representation, as the classic Half-Edge data structure.

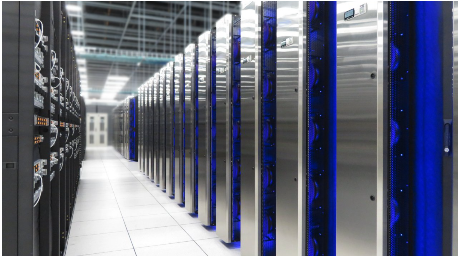


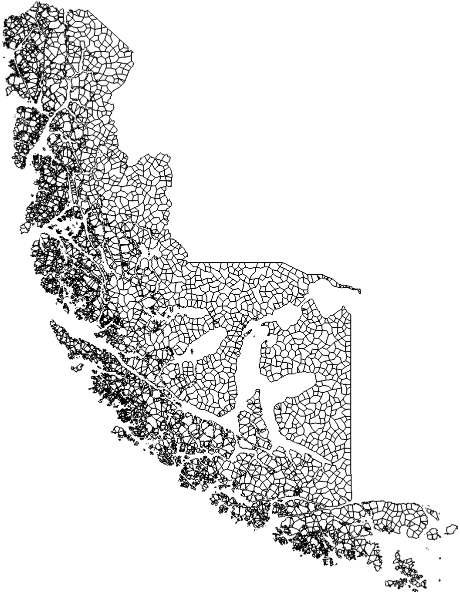
And an application for this data structure (a polygonal mesh generator). We can get a compaction of 99% the memory usage.

Table of contents

Motivation

Motivation



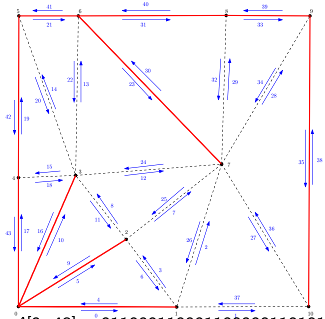


Background

Pemb data structure

Pemb† is a compact data structure that can represent a polygonal mesh with m edges in $4m$ bits. Some advantages of this data structure are:

- Pemb accept polygonal meshes with arbitrary shape polygons.
- Pemb accept polygonal meshes with holes.
- Pemb can be constructed in parallel.
- Pemb operations have a constant time complexity.



$A[0..48] =$ 0110001100011000001101010100000011010100111111110

$B[0..22] =$ 00101010000100011111111

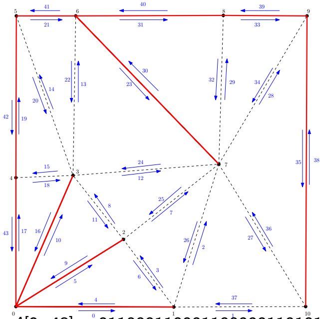
$B^*[0..26] =$ 000010000000011111100011111

Ferres, L., Fuentes-Sepúlveda, J., Gagie, T., He, M., & Navarro, G. (2017). Fast and Compact Planar Embeddings. WADS.

Pemb data structure

Pemb† is a compact data structure that can represent a polygonal mesh with m edges in $4m$ bits. Some advantages of this data structure are:

- Pemb accept polygonal meshes with arbitrary shape polygons.
- Pemb accept polygonal meshes with holes.
- Pemb can be constructed in parallel.
- Pemb operations have a constant time complexity.



$A[0..48] = 0110001100011000001101010100000001101010011111110$

$B[0..22] = 00101010000100011111111$

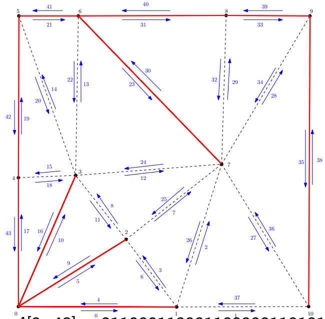
$B^*[0..26] = 000010000000111111100011111$

Ferres, L., Fuentes-Sepúlveda, J., Gagie, T., He, M., & Navarro, G. (2017). Fast and Compact Planar Embeddings. WADS.

Pemb data structure

Pemb† is a compact data structure that can represent a polygonal mesh with m edges in $4m$ bits. Some advantages of this data structure are:

- Pemb accept polygonal meshes with arbitrary shape polygons.
- Pemb accept polygonal meshes with holes.
- Pemb can be constructed in parallel.
- Pemb operations have a constant time complexity.



$A[0..48] = 01100011000110000011010101$
0000001101010011111110

$B[0..22] = 00101010000100011111111$

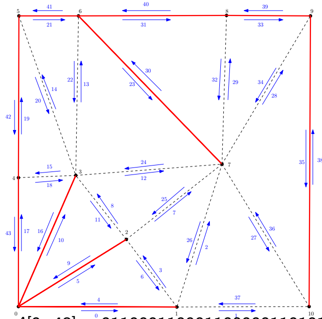
$B^*[0..26] = 000010000000011111100011111$

Ferres, L., Fuentes-Sepúlveda, J., Gagie, T., He, M., & Navarro, G. (2017). Fast and Compact Planar Embeddings. WADS.

Pemb data structure

Pemb† is a compact data structure that can represent a polygonal mesh with m edges in $4m$ bits. Some advantages of this data structure are:

- Pemb accept polygonal meshes with arbitrary shape polygons.
- Pemb accept polygonal meshes with holes.
- Pemb can be constructed in parallel.
- Pemb operations have a constant time complexity.



$A[0..48] =$ 0110001100011000001101010100000011010100111111110

$B[0..22] =$ 00101010000100011111111

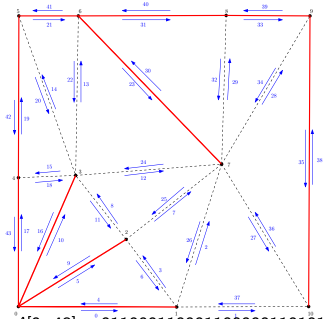
$B^*[0..26] =$ 000010000000011111100011111

Ferres, L., Fuentes-Sepúlveda, J., Gagie, T., He, M., & Navarro, G. (2017). Fast and Compact Planar Embeddings. WADS.

Pemb data structure

Pemb† is a compact data structure that can represent a polygonal mesh with m edges in $4m$ bits. Some advantages of this data structure are:

- Pemb accept polygonal meshes with arbitrary shape polygons.
- Pemb accept polygonal meshes with holes.
- Pemb can be constructed in parallel.
- Pemb operations have a constant time complexity.



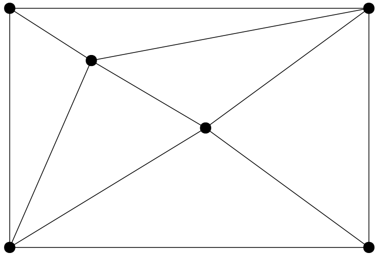
$A[0..48] = $ 01100011000110000011010101 00000011010100111111110

$B[0..22] = $ 00101010000100011111111

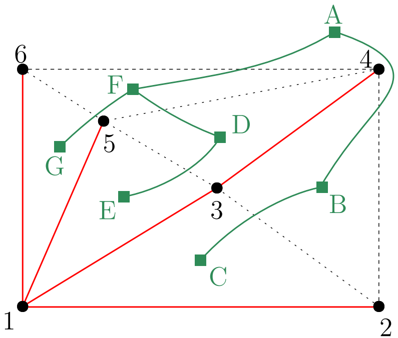
$B^{*}[0..26] = $ 000010000000011111100011111

Ferres, L., Fuentes-Sepúlveda, J., Gagie, T., He, M., & Navarro, G. (2017). Fast and Compact Planar Embeddings. WADS.

Pemb construction

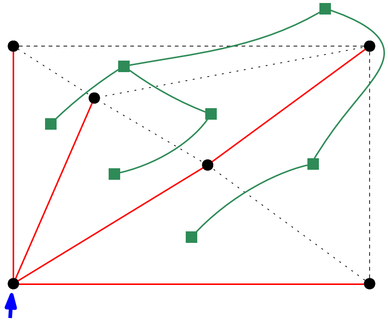


Pemb construction



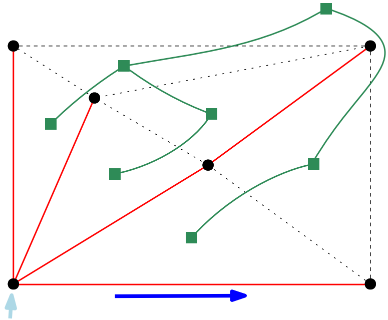
[(([[ ) ( ] ( ] [[ ) [ ) ( ]] [ ) ( ]]))]

Pemb construction



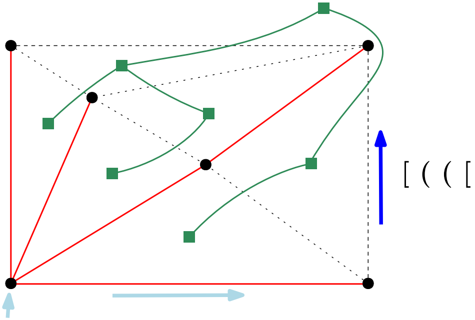
[(

Pemb construction

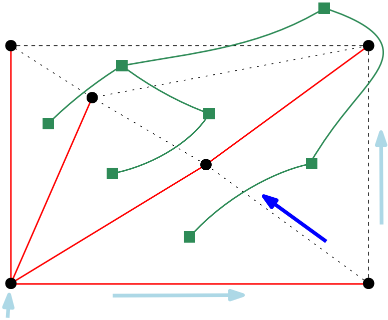


[((

Pemb construction

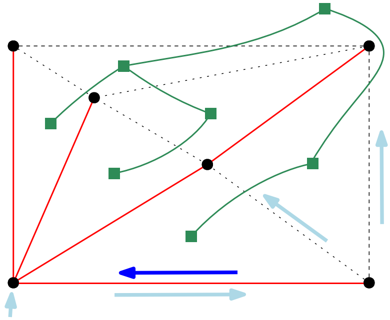


Pemb construction



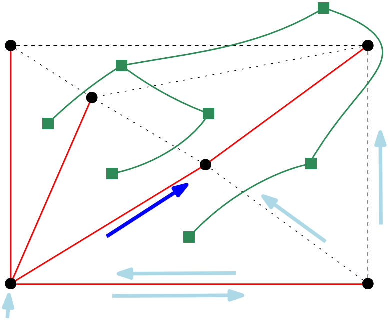
[(([[

Pemb construction



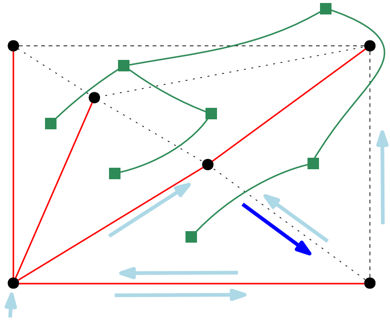
[(([[ )

Pemb construction



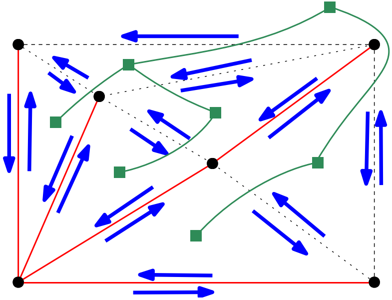
[(([[ ) (

Pemb construction



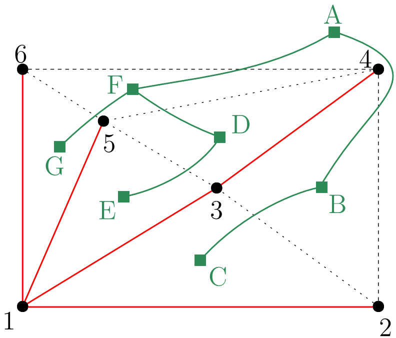
[(([[ ) ( ]

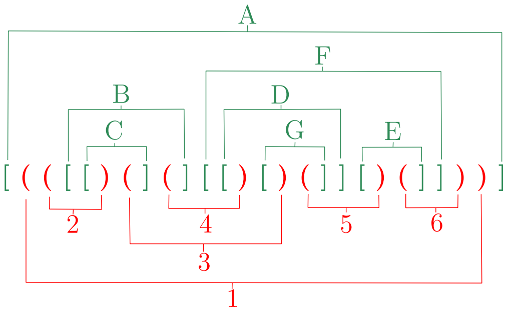
Pemb construction



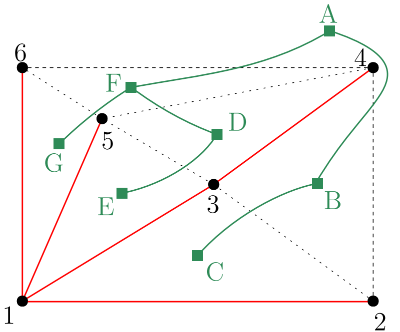
[(([[ ) ( ] ( ] [[ ) [ ) ( ]] [ ) ( ]]))]

Pemb construction





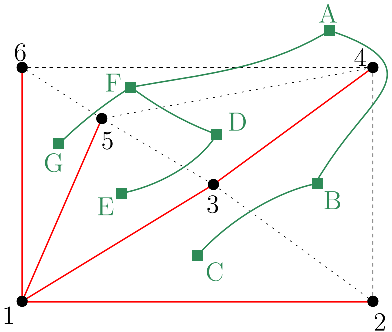
Pemb construction



[(([[ ) ( ] ( ] [[ ) [ ) ( ]] [ ) ( ]]))]

A = **0110011010001011000110010110**

Pemb construction



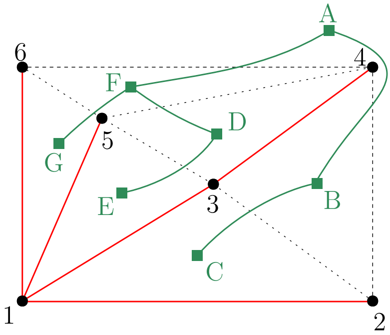
1 2 3 4 5 6

[(([[ ) ( ] ( ] [[ ) [ ) ( ]] [ ) ( ]]))]

A = **01100110100010110001100110**

B = **0 0 1 0 0 1 1 0 1 0 1 1**

Pemb construction



A 1 2 B C 3 4 D E F 5 G 6

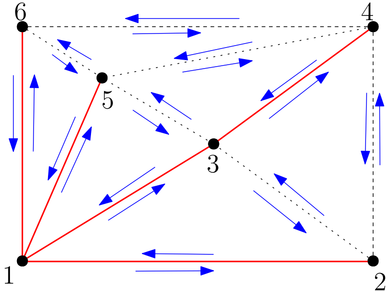
[(([[ ) ( ] ( ] [[ ) [ ) ( ]] [ ) ( ]]))]

A = **01100110100010110001100110**

B = **0 0 1 0 0 1 1 0 1 0 1 1**

B* = **0 0 0 1 1 0 0 0 1 1 0 1 1 1**

Pemb construction



A 1 2 B C 3 4 D E F 5 G 6

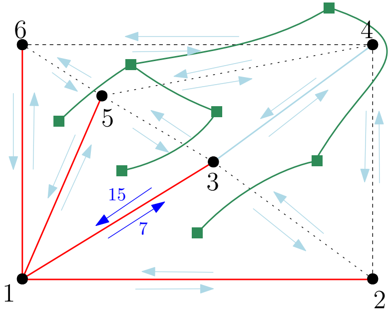
[(([[ ) ( ] ( ] [[ ) [ ) ( ]] [ ) ( ]]))]

A = **0110011010001011000110011 0**

B = **0 0 1 0 0 1 1 0 1 0 1 1**

B* = **0 0 0 1 1 0 0 0 1 1 0 1 1 1**

Return the complementary edge of $1 - 3$



A 1 2 B C 3 4 D E F 5 G 6

[(([[ ) ( ] ( ] [[ ) [ ) ( ]] [ ) ( ]]))]

A = **01100110100010110001100110**

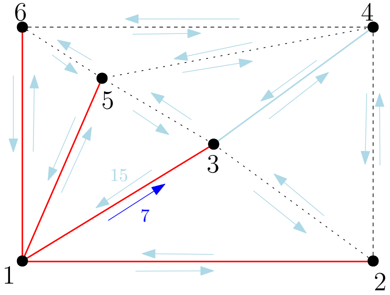
B = **0 0 1 0 0 1 1 0 1 0 1 1**

B* = **0 0 0 1 1 0 0 0 1 1 0 1 1 1**

$$\text{mate(i)} = \begin{cases} A.select_0(B*.match(A.rank_0(i+1))) & \text{if A[i] = 1} \\ A.select_1(B.match(A.rank_1(i+1))) & \text{Otherwise} \end{cases}$$

Example of Pemb query

Return the complementary edge of $1 - 3$



mate(7) = $A.select_1(B.match(A.rank_1(7)))$

A 1 2 B C 3 4 D E F 5 G 6

[(([[ ) ( ] ( ] [[ ) [ ) ( ]] [ ) ( ]]))]

A = 01100110100010110001100110

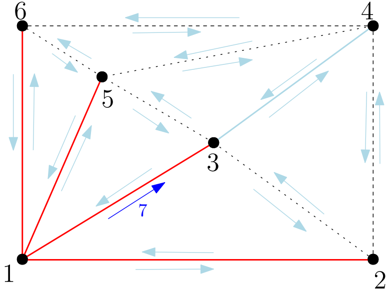
B = 0 0 1 0 0 1 1 0 1 0 1 1

B* = 0 0 0 1 1 0 0 0 1 1 0 1 1 1

Count the number of 1's in A until the position $7 = 4$

Example of Pemb query

Return the complementary edge of $1 - 3$



mate(7) = $A.select_1(B.match(4))$

A 1 2 B C 3 4 D E F 5 G 6

[(([[ ) ( ] ( ] [[ ) [ ) ( ]] [ ) ( ]]))]

A = 01100110100010110001100110

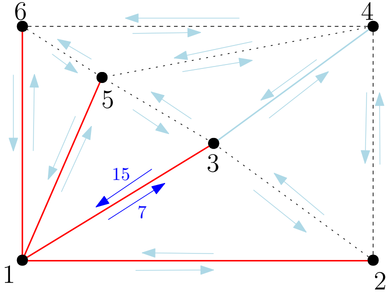
B = 0 0 1 0 0 1 1 0 1 0 1 1

B* = 0 0 0 1 1 0 0 0 1 1 0 1 1 1

Find the position of complementary edge to $1 - 3$ in $B = 7$

Example of Pemb query

Return the complementary edge of $1 - 3$



mate(7) $= A.select_1(7) = 15$

A 1 2 B C 3 4 D E F 5 G 6

[(([[ ) ( ] ( ] [[ ) [ ) ( ]] [ ) ( ]]))]

A = 011001101000101110001100110

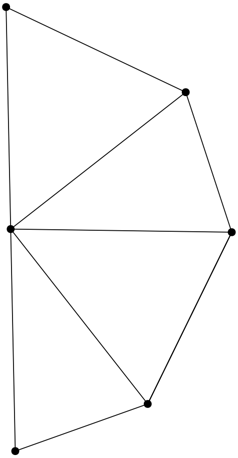
B = 0 0 1 0 0 1 1 0 1 0 1 1

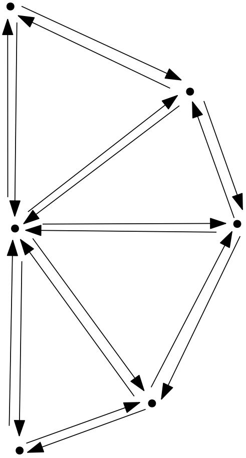
B* = 0 0 0 1 1 0 0 0 1 1 0 1 1 1

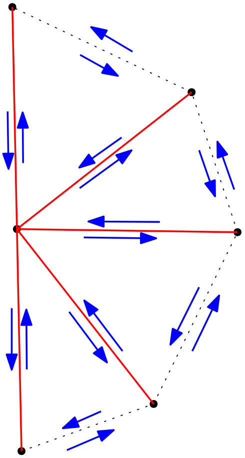
Find the position of seventh 1 in $A = 15$

Compact data structure encapsulation as Half-edge data structure

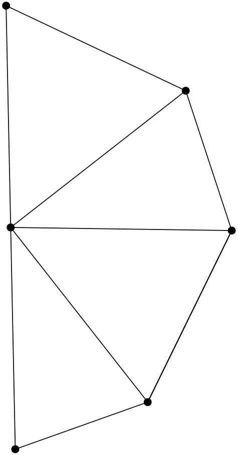
Pemb encapsulation as Half-edge


Triangular Mesh

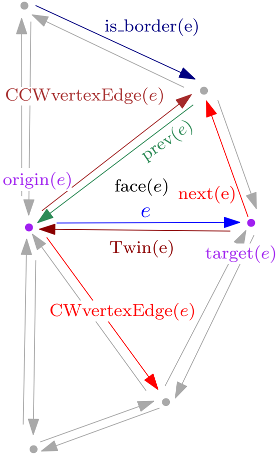

Half-edge representation


Pemb representation

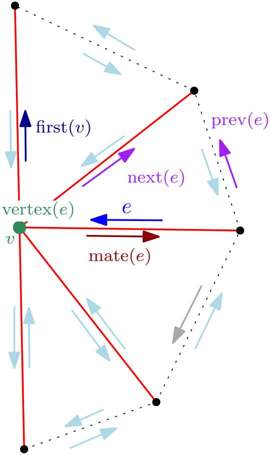
Pemb encapsulation as Half-edge



Triangular Mesh



Half-edge representation

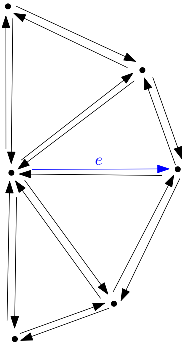


Pemb representation

Half-edge queries as pemb queries

In the Half-edge DS we use 11 queries to get information of the mesh, those queries can be done in pemb by combining pemb's basic queries, for example:

- next(e):
- target(e):

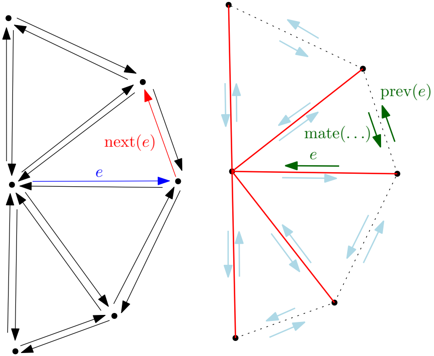


We had to develop 2 new queries for pemb to get the same information as the half-edge DS, those are pemb::first_dual(f) and pemb::get_face(e).

Half-edge queries as pemb queries

In the Half-edge DS we use 11 queries to get information of the mesh, those queries can be done in pemb by combining pemb's basic queries, for example:

- next(e): pemb::mate(pemb::prev(e))
- target(e):

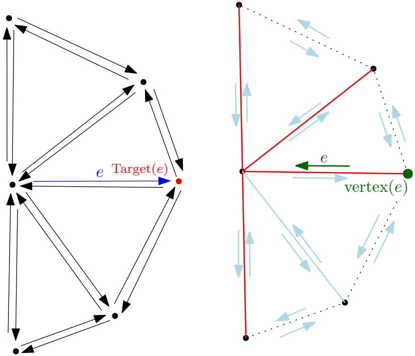


We had to develop 2 new queries for pemb to get the same information as the half-edge DS, those are pemb::first_dual(f) and pemb::get_face(e).

Half-edge queries as pemb queries

In the Half-edge DS we use 11 queries to get information of the mesh, those queries can be done in pemb by combining pemb's basic queries, for example:

- next(e): pemb::mate(pemb::prev(e))
- target(e): pemb::vertex(e)



We had to develop 2 new queries for pemb to get the same information as the half-edge DS, those are pemb::first_dual(f) and pemb::get_face(e).

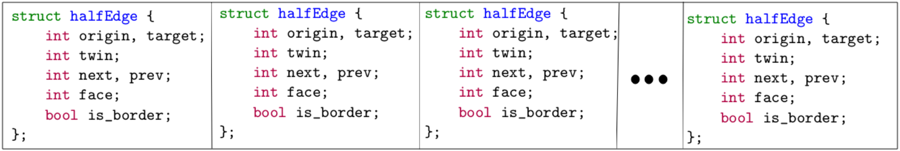
Data structure implementation

Implementation of the non-compact data structure

For the Half-edge implementation we use two array of structures (AoS).

```
struct vertex{
  double x, y;
  bool is_border;
  int incident_halfedge;
};
```

```
struct halfEdge {
  int origin, target;
  int twin;
  int next;
  bool is_border;
};
```



Implementation of the compact data structure

- The compact half-edge (pemb) is stored in only 3 bitvectors!
- Small Extra space is used to support queries in those bitvectors.
- Coordinates of the vertices are stored in an array since they can not be compacted.

Implementation of the compact data structure

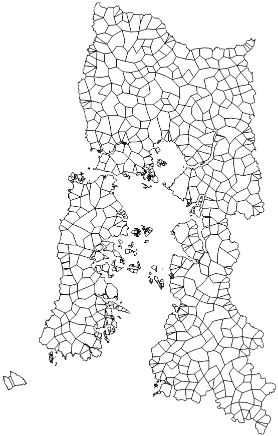
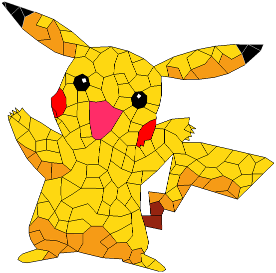
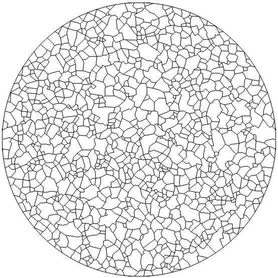
- The compact half-edge (pemb) is stored in only 3 bitvectors!
- Small Extra space is used to support queries in those bitvectors.
- Coordinates of the vertices are stored in an array since they can not be compacted.

Implementation of the compact data structure

- The compact half-edge (pemb) is stored in only 3 bitvectors!
- Small Extra space is used to support queries in those bitvectors.
- Coordinates of the vertices are stored in an array since they can not be compacted.

Mesh generation using compact data structure

Polylla mesh generator

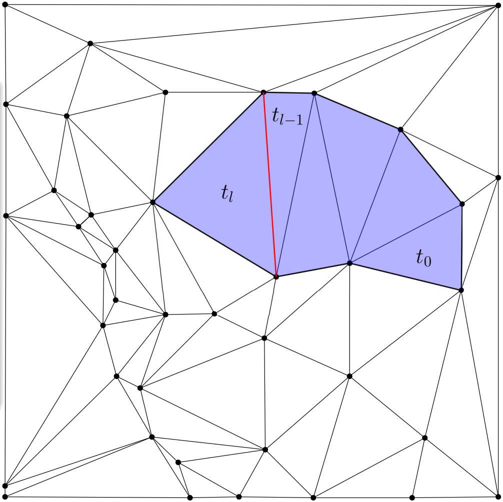




Source: Salinas-Fernández, S., Hitschfeld-Kahler, N., Ortiz-Bernardin, A., & Si, H. (2022). POLYLLA: Polygonal meshing algorithm based on terminal-edge regions. Engineering with Computers, 38(5), 4545–4567. https://doi.org/10.1007/s00366-022-01643-4

Longest-edge propagation path (Lepp) 2D

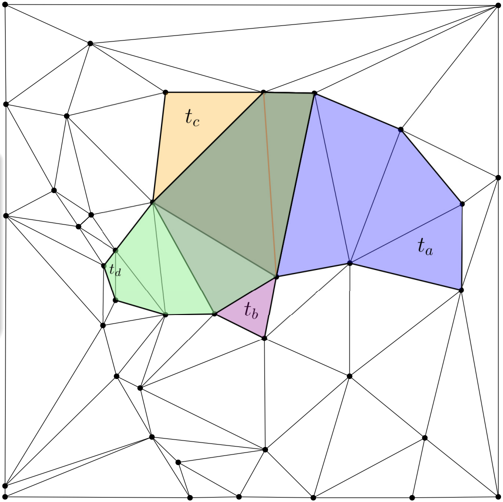
Longest Edge Propagation Path 2D †

The lepp of a triangle t_0 is the ordered list of all adjacent triangles $t_0, t_1, ..., t_l$, such that t_i is the longest edge neighbor triangle of t_{i-1} by the longest-edge of t_{i-1}, for $i = 1, 2, ..., l$.



† **Source** Maria-Cecilia Rivara. New longest-edge algorithms for the refinement and/or improvement of unstructured triangulations. International Journal for Numerical Methods in Engineering, 40(18):3313-3324, 1997.

Terminal-edge regions



Terminal-edge region ‡

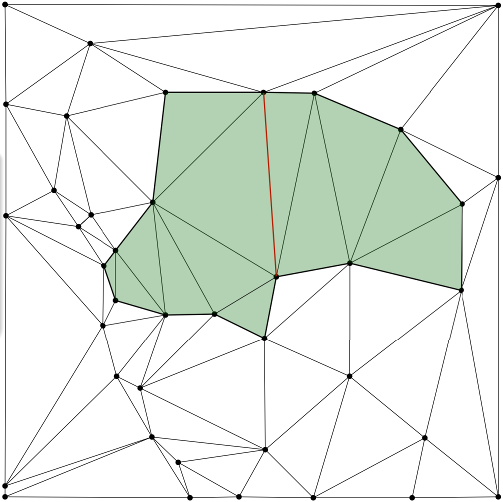
A *terminal-edge region R* is a region formed by the union of all triangles t_i such that Lepp(t_i) has the same terminal-edge.

‡R. Alonso, J. Ojeda, N. Hitschfeld, C. Hervías, and L.E. Campusano. Delaunay based algorithm for finding polygonal voids in planar point sets. Astronomy and Computing, 22:48 - 62, 2018. 20

Terminal-edge regions

Terminal-edge region ‡

A *terminal-edge region R* is a region formed by the union of all triangles t_i such that Lepp(t_i) has the same terminal-edge.

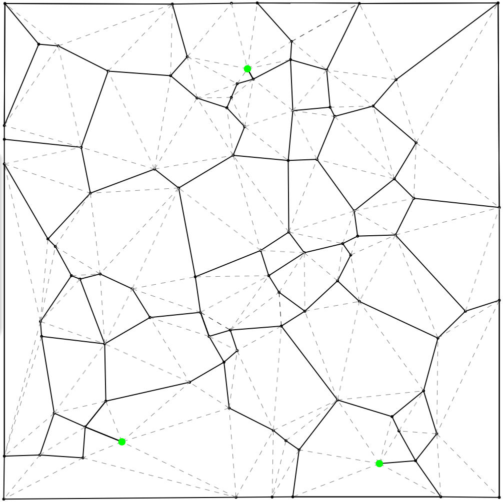


‡R. Alonso, J. Ojeda, N. Hitschfeld, C. Hervías, and L.E. Campusano. Delaunay based algorithm for finding polygonal voids in planar point sets. Astronomy and Computing, 22:48 - 62, 2018. 20

Frontier-edges

Frontier-edges ‡

A frontier-edge is an edge that is shared by two triangles, each one belonging to a different terminal-edge region.



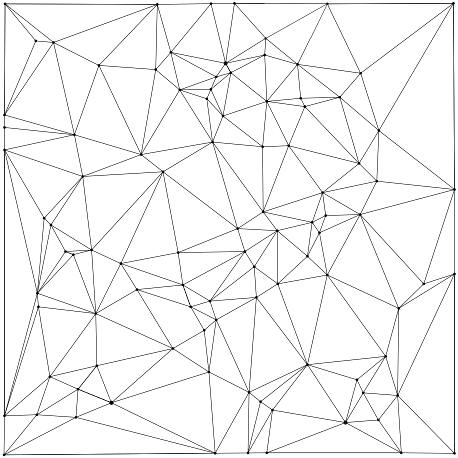
‡R. Alonso, J. Ojeda, N. Hitschfeld, C. Hervías, and L.E. Campusano. Delaunay based algorithm for finding polygonal voids in planar point sets. Astronomy and Computing, 22:48 - 62, 2018. 20

Polylla 2D algorithm

- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh

Algorithm has three main phases

1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase

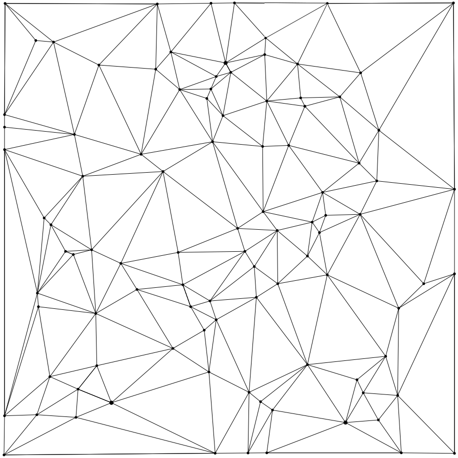


Polylla 2D algorithm

- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh

Algorithm has three main phases

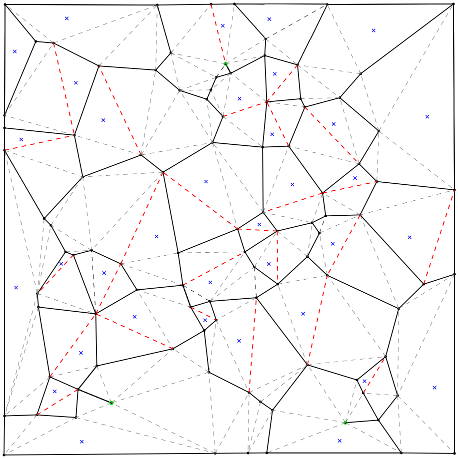
1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase



- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh

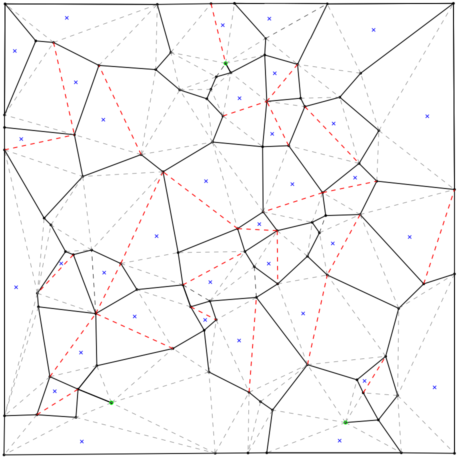
Algorithm has three main phases

1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase



Polylla 2D algorithm

- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh



Algorithm has three main phases

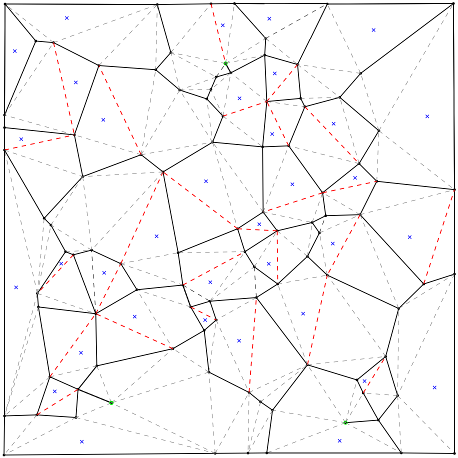
1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase

Traversal phase

- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh

Algorithm has three main phases

1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase

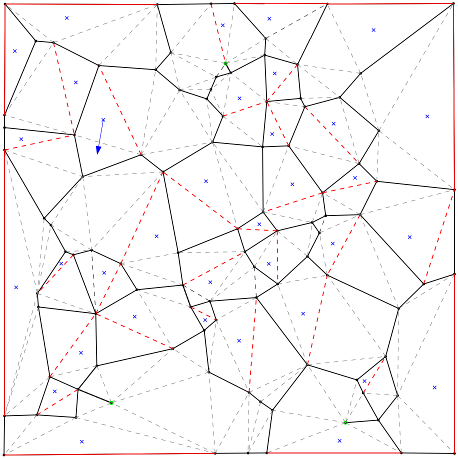


Traversal phase

- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh

Algorithm has three main phases

1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase

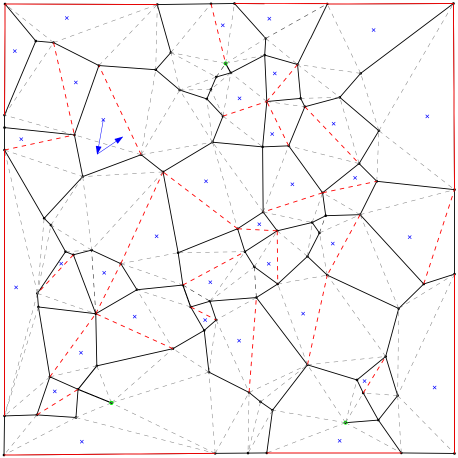


Traversal phase

- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh

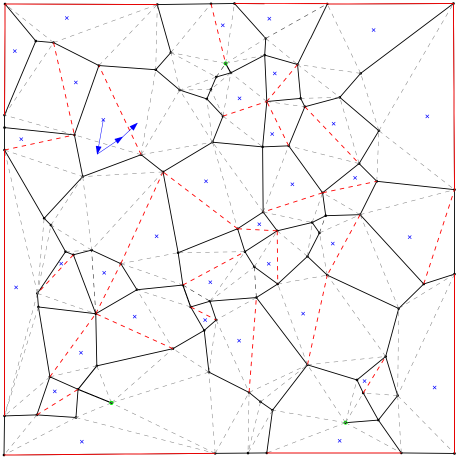
Algorithm has three main phases

1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase



Traversal phase

- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh

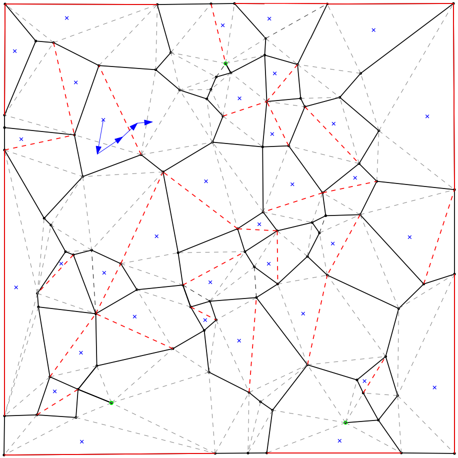


Algorithm has three main phases

1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase

Traversal phase

- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh

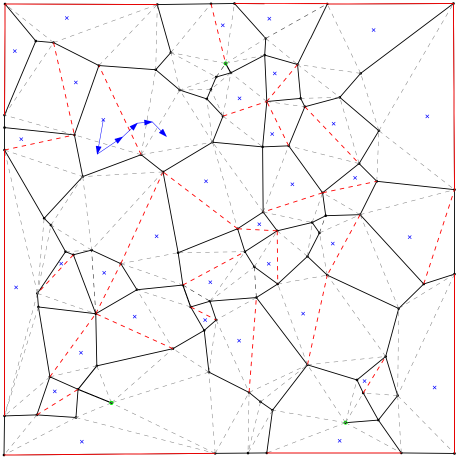


Algorithm has three main phases

1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase

Traversal phase

- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh

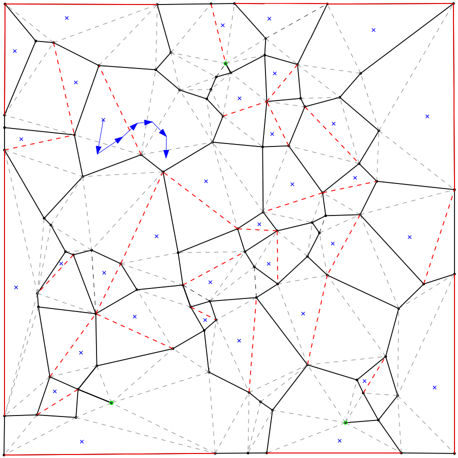


Algorithm has three main phases

1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase

Traversal phase

- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh



Algorithm has three main phases

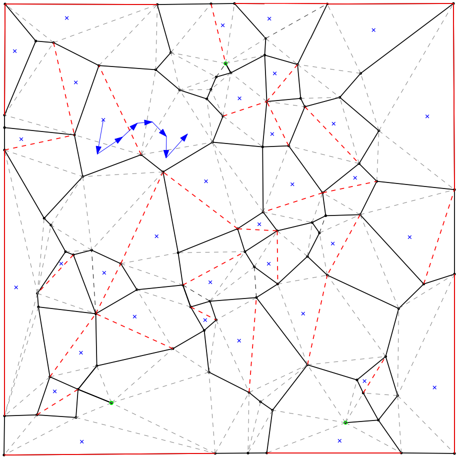
1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase

Traversal phase

- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh

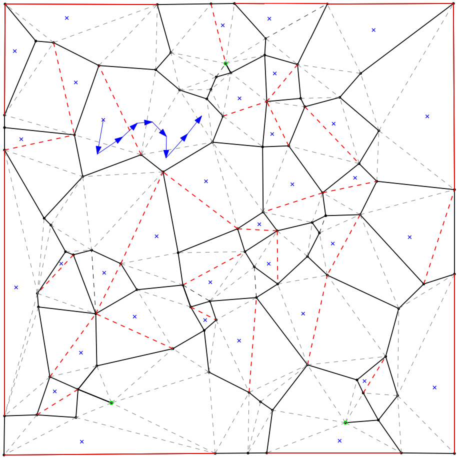
Algorithm has three main phases

1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase



Traversal phase

- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh

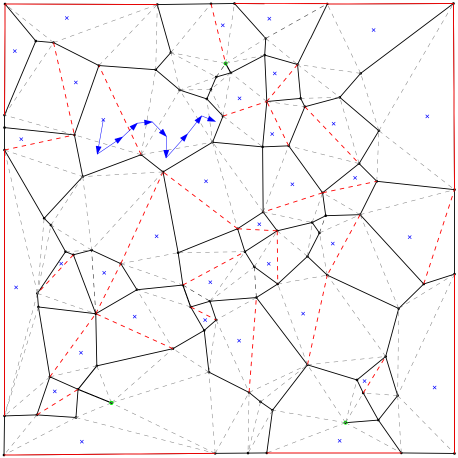


Algorithm has three main phases

1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase

Traversal phase

- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh

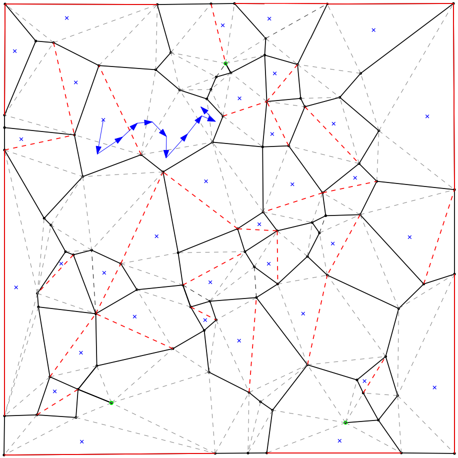


Algorithm has three main phases

1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase

Traversal phase

- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh

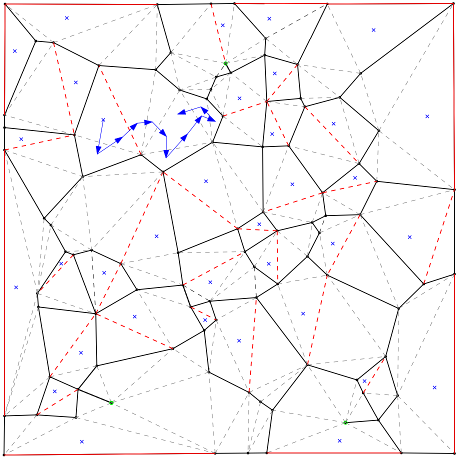


Algorithm has three main phases

1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase

Traversal phase

- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh



Algorithm has three main phases

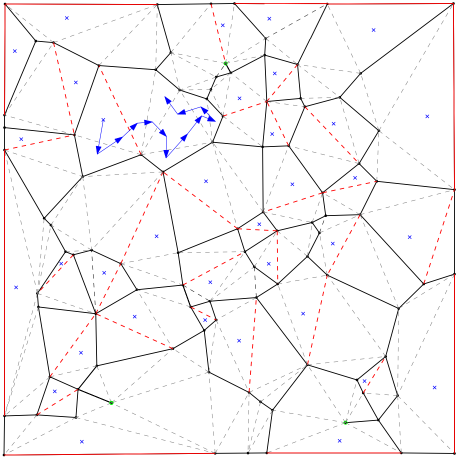
1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase

Traversal phase

- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh

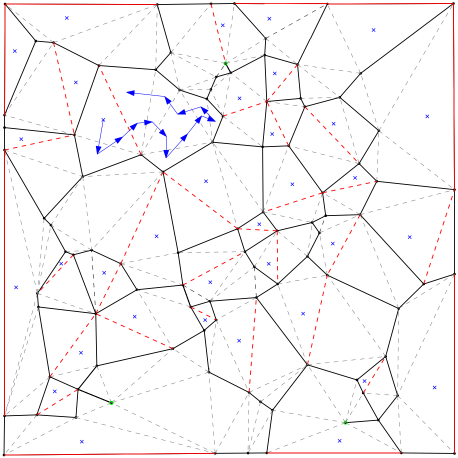
Algorithm has three main phases

1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase



Traversal phase

- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh



Algorithm has three main phases

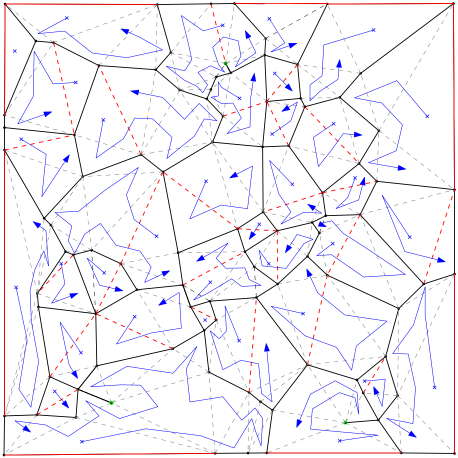
1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase

Traversal phase

- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh

Algorithm has three main phases

1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase

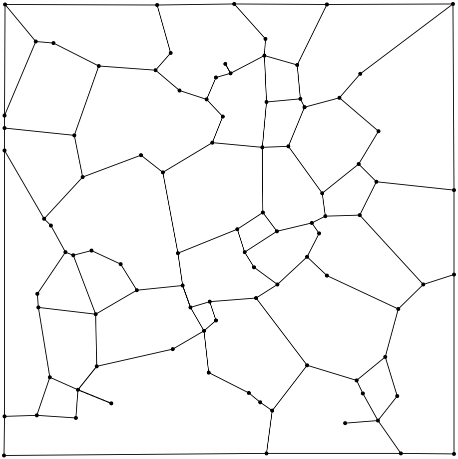


Traversal phase

- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh

Algorithm has three main phases

1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase

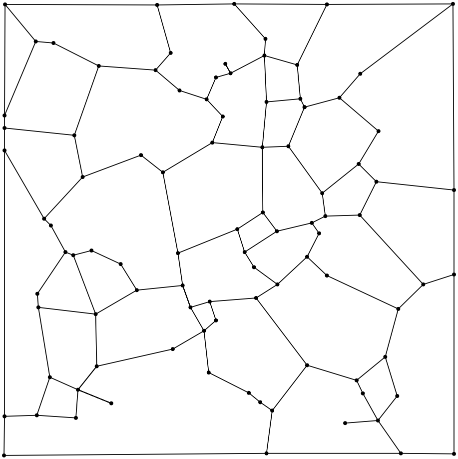


Polylla 2D algorithm

- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh

Algorithm has three main phases

1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase

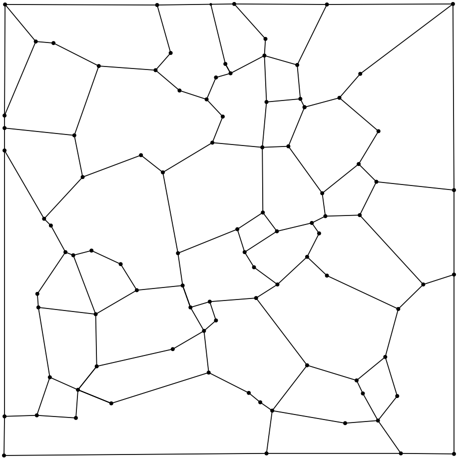


Polylla 2D algorithm

- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh

Algorithm has three main phases

1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase



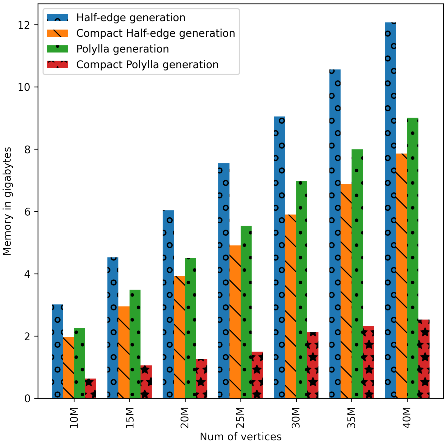
Experiments

Experiments

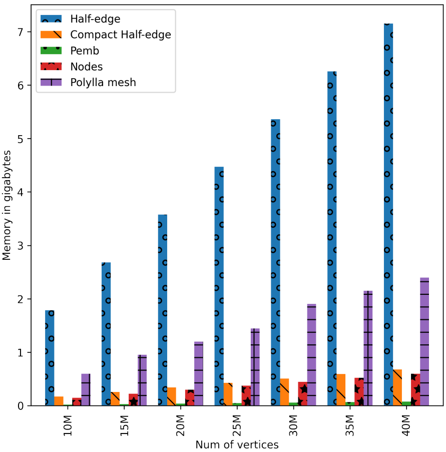
- Polylla was implemented in C++.
- To generate the initital Delaunay triangulation, we used the CGAL library.
- The implementation of Polylla was the same for the two version of the compact half-edge.
- We use the library `malloc_count` to measure the memory usage.

```
Compact polylla source code
https://github.com/ssalinasfe/Compact-Polylla-Mesh
Pemb source code
https://github.com/jfuentess/sdsl-lite
```

Memory usage Experiments



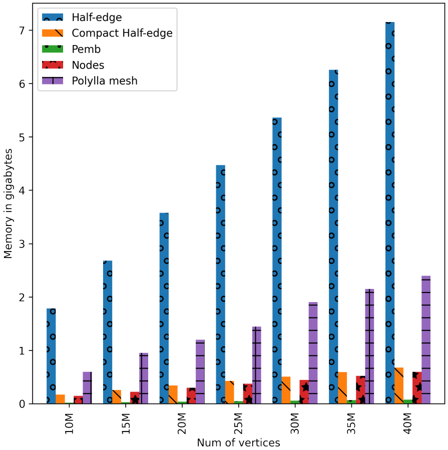
Memory peak during the generation of the data structure and Polylla



Memory usage after the generation of the data structure and Polylla

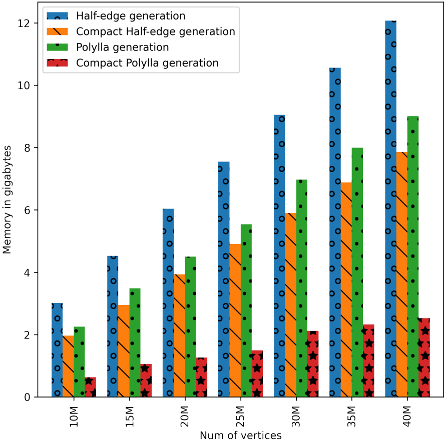
Memory usage results

- Topological information (without coords) of a polygonal mesh can be **compacted by a 99%** using compact half-edge.
- The compact mesh uses a 9.0% of memory of the non-compact version.
- The memory to store the compact mesh is distributed in 88.67% for the point coordinates and 11.33% for the half-edge data structure.



Memory usage after the generation of the data structure and Polylla.

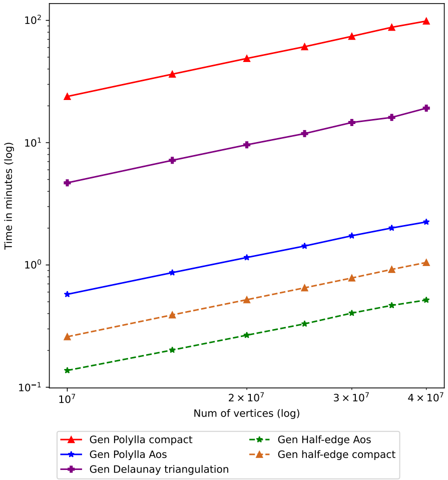
Memory peak results



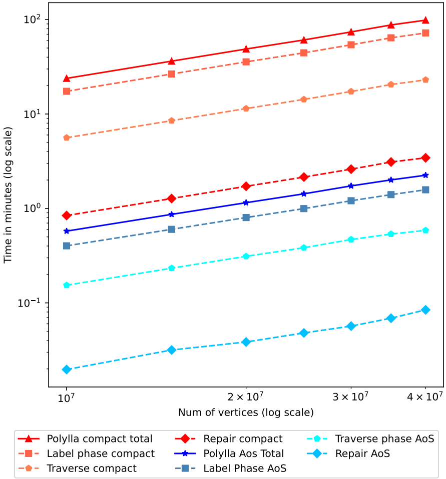
Memory peak during the generation of the data structure and Polylla

- Generate a polylla mesh takes 3.49x more memory using the AoS version than the compact version.

Time results

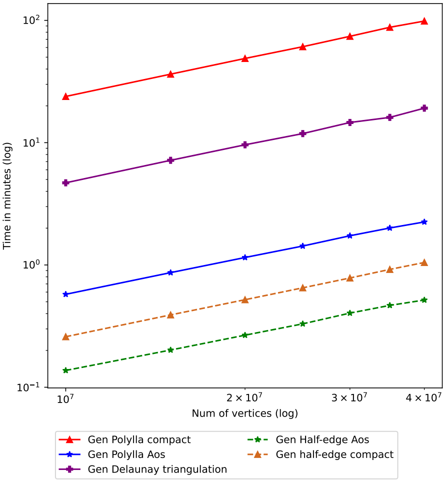


Time to generate the half-edge data structure and Polylla meshes



Polylla mesh phases times

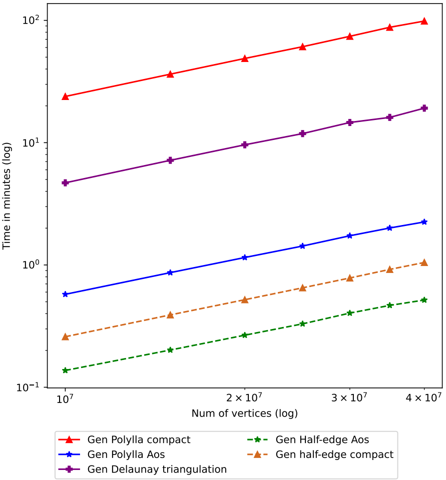
Time results



Time to generate the half-edge data structure and Polylla meshes

- The construction of AoS half-edge data structure is 1.95x faster than the construction of compact half-edge.
- Generation of Polylla using AoS half-edge is 42.7x faster than the generation using the compact half-edge.

Time results

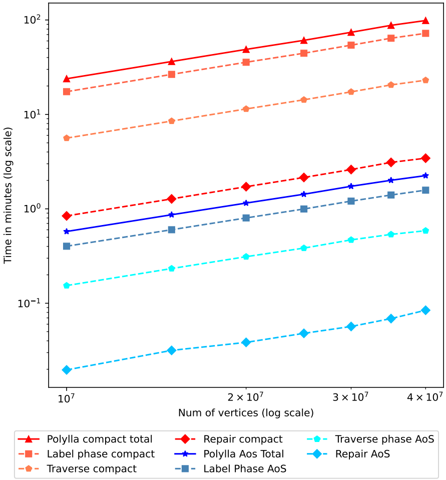


Time to generate the half-edge data structure and Polylla meshes

- The construction of AoS half-edge data structure is 1.95x faster than the construction of compact half-edge.
- Generation of Polylla using AoS half-edge is 42.7x faster than the generation using the compact half-edge.

Time results

- The complexity is $O(n)$ using both data structures.
- The growth of all the phases is the same for both data structures.



Polylla mesh phases times

Conclusions and future work

Conclusions and future work

- We encapsulate a compact data structure as the well-known half-edge data structure. This is the first practical use of pemb.
- We develop new queries for pemb to support the generation of polygonal meshes.
- We can get a compaction of the 99% of the topological information of a polygonal mesh.
- We expect to expand pemb to support surface meshes.
- We expect to add edge flip and vertex insertion in the future.
- We expect to test this compact data structure in parallel.

Conclusions and future work

- We encapsulate a compact data structure as the well-known half-edge data structure. This is the first practical use of pemb.
- We develop new queries for pemb to support the generation of polygonal meshes.
- We can get a compaction of the 99% of the topological information of a polygonal mesh.
- We expect to expand pemb to support surface meshes.
- We expect to add edge flip and vertex insertion in the future.
- We expect to test this compact data structure in parallel.

Conclusions and future work

- We encapsulate a compact data structure as the well-known half-edge data structure. This is the first practical use of pemb.
- We develop new queries for pemb to support the generation of polygonal meshes.
- We can get a compaction of the 99% of the topological information of a polygonal mesh.
- We expect to expand pemb to support surface meshes.
- We expect to add edge flip and vertex insertion in the future.
- We expect to test this compact data structure in parallel.

Conclusions and future work

- We encapsulate a compact data structure as the well-known half-edge data structure. This is the first practical use of pemb.
- We develop new queries for pemb to support the generation of polygonal meshes.
- We can get a compaction of the 99% of the topological information of a polygonal mesh.
- We expect to expand pemb to support surface meshes.
- We expect to add edge flip and vertex insertion in the future.
- We expect to test this compact data structure in parallel.

Conclusions and future work

- We encapsulate a compact data structure as the well-known half-edge data structure. This is the first practical use of pemb.
- We develop new queries for pemb to support the generation of polygonal meshes.
- We can get a compaction of the 99% of the topological information of a polygonal mesh.
- We expect to expand pemb to support surface meshes.
- We expect to add edge flip and vertex insertion in the future.
- We expect to test this compact data structure in parallel.

Conclusions and future work

- We encapsulate a compact data structure as the well-known half-edge data structure. This is the first practical use of pemb.
- We develop new queries for pemb to support the generation of polygonal meshes.
- We can get a compaction of the 99% of the topological information of a polygonal mesh.
- We expect to expand pemb to support surface meshes.
- We expect to add edge flip and vertex insertion in the future.
- We expect to test this compact data structure in parallel.

Thanks you!

I'm looking for postdoctoral position or jobs!

Dropped frames

Background

Compression vs Compaction

Compression and Compaction are different concepts.

- Compression: Studies the minimum possible space necessary to store data.
  - Require decompression of the data to be used.
  - Does not useful for real-time applications.
- Compaction: Studies the minimum possible space necessary to store data **but allowing access to the information.**
  - Allows us to fit and efficiently query, navigate, and manipulate much larger datasets in main memory.
  - Uses more memory than compression.

.

Compression vs Compaction

Compression and Compaction are different concepts.

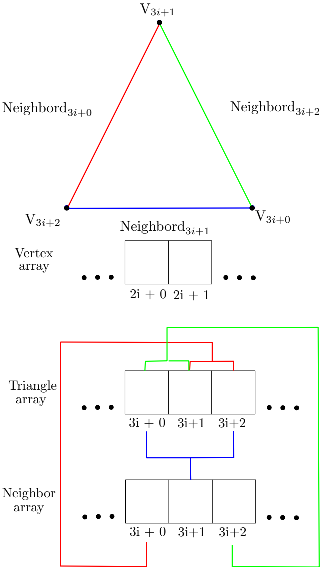
- Compression: Studies the minimum possible space necessary to store data.
  - Require decompression of the data to be used.
  - Does not useful for real-time applications.
- Compaction: Studies the minimum possible space necessary to store data **but allowing access to the information.**
  - Allows us to fit and efficiently query, navigate, and manipulate much larger datasets in main memory.
  - Uses more memory than compression.

The minimum space suffice to represent a polygonal mesh with m edges is $3.58m$ bits.

Kind of compact mesh data structures

- Classic data structures
  - Face based 13 rpv
  - Half-edge 19 rpv
- Compact data structures
  - Catalog representation 7.64 rpv
  - Star vertex representation 7 rpv
  - Schnyder wood representation 4m bits
  - Pemb 4m bits

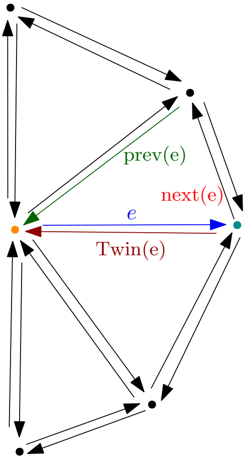
Rpv = references per vertex



Kind of compact mesh data structures

- Classic data structures
  - Face based 13 rpv
  - Half-edge 19 rpv
- Compact data structures
  - Catalog representation 7.64 rpv
  - Star vertex representation 7 rpv
  - Schnyder wood representation 4m bits
  - Pemb 4m bits

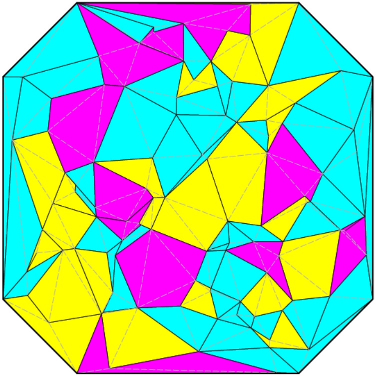
Rpv = references per vertex



Kind of compact mesh data structures

- Classic data structures
  - Face based 13 rpv
  - Half-edge 19 rpv
- Compact data structures
  - Catalog representation 7.64 rpv
  - Star vertex representation 7 rpv
  - Schnyder wood representation 4m bits
  - Pemb 4m bits

Rpv = references per vertex

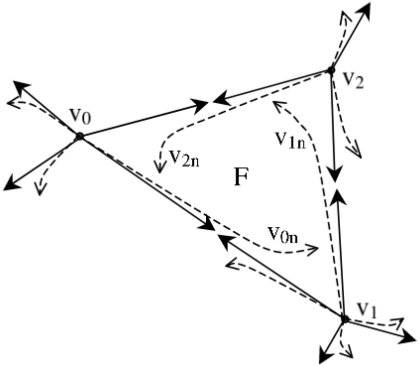


Source: ALEARDI, L. C., DEVILLERS, O., & MEBARKI, A. (2011). CATALOG-BASED REPRESENTATION OF 2D TRIANGULATIONS. International Journal of Computational Geometry & Applications, 21(04), 393–402.

Kind of compact mesh data structures

- Classic data structures
  - Face based 13 rpv
  - Half-edge 19 rpv
- Compact data structures
  - Catalog representation 7.64 rpv
  - Star vertex representation 7 rpv
  - Schnyder wood representation 4m bits
  - Pemb 4m bits

Rpv = references per vertex



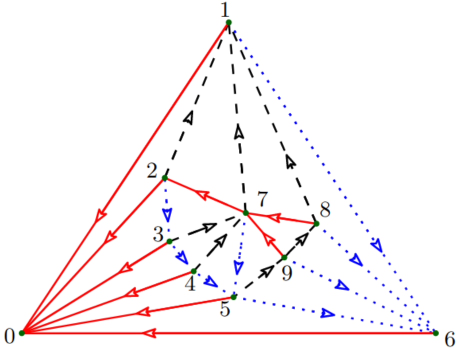
Source: Kallmann, M., & Thalmann, D. (2001). Star-Vertices: A Compact Representation for Planar Meshes with Adjacency Information. Journal of Graphics Tools, 6(1), 7–18. https://doi.org/10.1080/10867651.2001.1048753

Kind of compact mesh data structures

- Classic data structures
  - Face based 13 rpv
  - Half-edge 19 rpv
- Compact data structures
  - Catalog representation 7.64 rpv
  - Star vertex representation 7 rpv
  - Schnyder wood representation 4m bits
  - Pemb 4m bits

Rpv = references per vertex

m = edges of the mesh



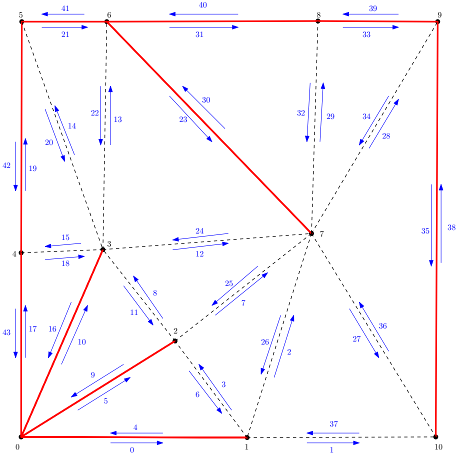
Source: Aleardi, L. C., & Devillers, O. (2018). Array-based compact data structures for triangulations: Practical solutions with theoretical guarantees. Journal of Computational Geometry, 9(1), Article 1. https://doi.org/10.20382/jocg.v9i1a8

Kind of compact mesh data structures

- Classic data structures
  - Face based 13 rpv
  - Half-edge 19 rpv
- Compact data structures
  - Catalog representation 7.64 rpv
  - Star vertex representation 7 rpv
  - Schnyder wood representation 4m bits
  - Pemb 4m bits

Rpv = references per vertex

m = edges of the mesh



$A[0..48]$ = 0110001100011000001101010100000011010100111111110

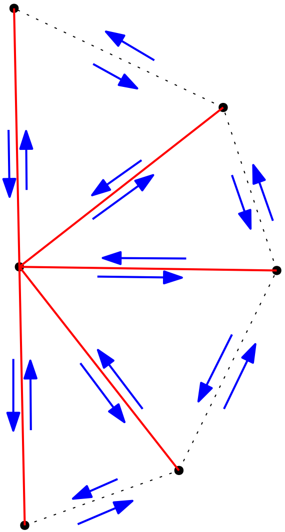
$B[0..22]$ = 00101010000100011111111

$B^*[0..26]$ = 000010000000111111100011111

Pemb

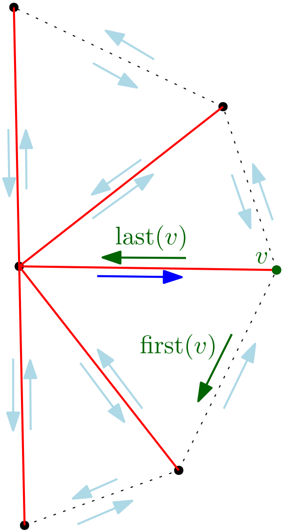
Pemb queries

- `first(v)/last(v)`: the first/last half-edge whose source is the node v;
- `mate(i)`: the other half-edge corresponding to the same edge of i;
- `next(i)/prev(i)`: the next/previous half-edge with the same source of i, in ccw order of the neighbors of v.
- `vertex(i)`: the index of the node that is the source of half-edge i;



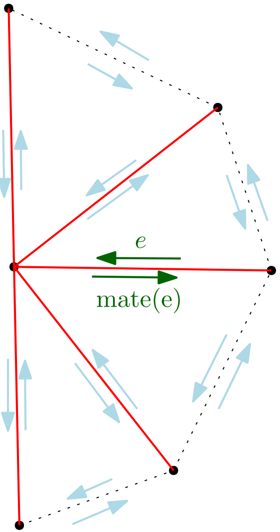
Pemb queries

- `first(v)/last(v)`: the first/last half-edge whose source is the node v;
- `mate(i)`: the other half-edge corresponding to the same edge of i;
- `next(i)/prev(i)`: the next/previous half-edge with the same source of i, in ccw order of the neighbors of v.
- `vertex(i)`: the index of the node that is the source of half-edge i;



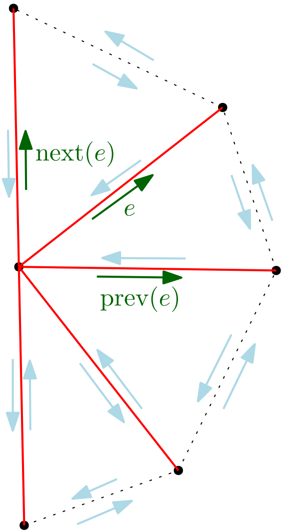
Pemb queries

- first(v)/last(v): the first/last half-edge whose source is the node v;
- mate(i): the other half-edge corresponding to the same edge of i;
- next(i)/prev(i): the next/previous half-edge with the same source of i, in ccw order of the neighbors of v.
- vertex(i): the index of the node that is the source of half-edge i;



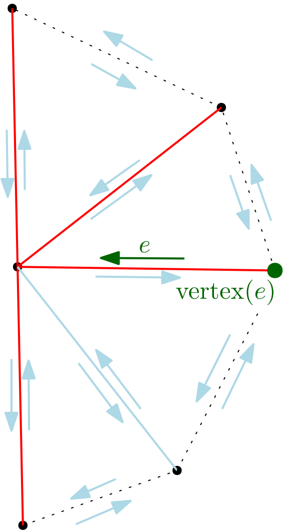
Pemb queries

- `first(v)/last(v)`: the first/last half-edge whose source is the node v;
- `mate(i)`: the other half-edge corresponding to the same edge of i;
- `next(i)/prev(i)`: the next/previous half-edge with the same source of i, in ccw order of the neighbors of v.
- `vertex(i)`: the index of the node that is the source of half-edge i;



Pemb queries

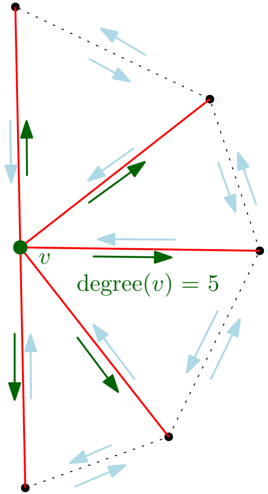
- `first(v)/last(v)`: the first/last half-edge whose source is the node v;
- `mate(i)`: the other half-edge corresponding to the same edge of i;
- `next(i)/prev(i)`: the next/previous half-edge with the same source of i, in ccw order of the neighbors of v.
- `vertex(i)`: the index of the node that is the source of half-edge i;



Pemb extra queries

We implemented extra queries to simulate the behavior of the half-edge data structure.

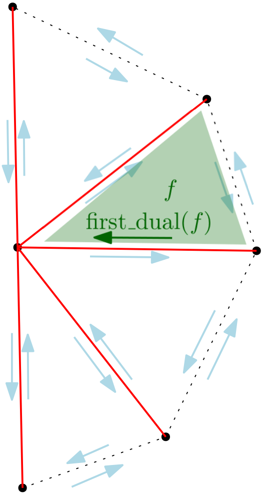
- **degree(v)**: return the number of edges incident to vertex v.
- **first_dual(f)**: return the position of the first visited edge incident to face f during the traversal of T.
- **get_face(e)**: return the id of the face incident to edge e.



Pemb extra queries

We implemented extra queries to simulate the behavior of the half-edge data structure.

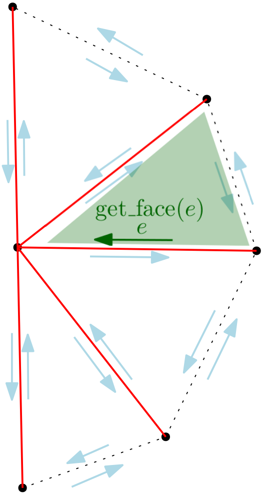
- **degree(v)**: return the number of edges incident to vertex v.
- **first_dual(f)**: return the position of the first visited edge incident to face f during the traversal of T.
- **get_face(e)**: return the id of the face incident to edge e.



Pemb extra queries

We implemented extra queries to simulate the behavior of the half-edge data structure.

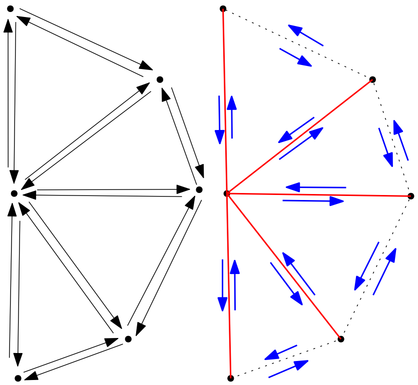
- **degree(v)**: return the number of edges incident to vertex v.
- **first_dual(f)**: return the position of the first visited edge incident to face f during the traversal of T.
- **get_face(e)**: return the id of the face incident to edge e.



Half-edge queries as pemb queries

Given plannar graph G. Each edge $e \in E$ is split into two half-edges h_1 and h_2 with opposite orientation in both data structures.

- Edge h_i has an orientation
- Edge h_i has a twin edge with opposite orientation
- Edge h_i is accessible by random access
- All edges of face f_j have the same orientation

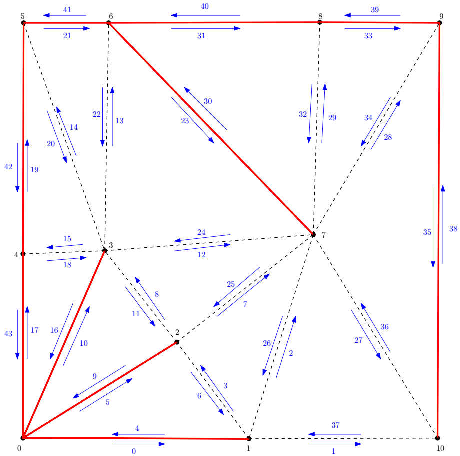


Compact data structure encapsulation as Half-edge data structure

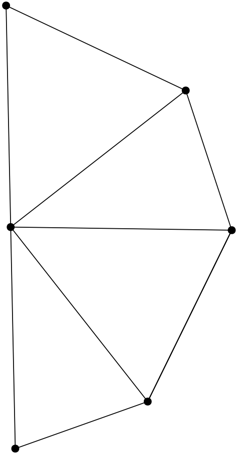
Half-edge queries as pemb queries

Given plannar graph G. Each edge $e \in E$ is split into two half-edges h_1 and h_2 with opposite orientation in both data structures.

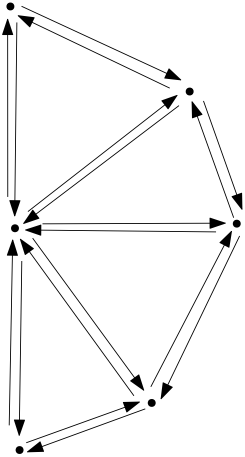
- Edge h_i has an orientation
- Edge h_i has a twin edge with opposite orientation
- Edge h_i is accessible by random access
- All edges of face f_j have the same orientation



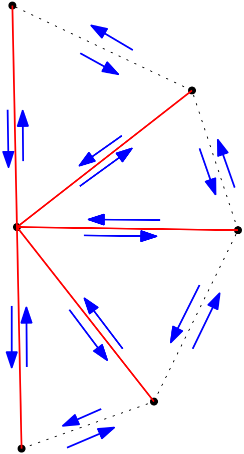
Pemb encapsulation as Half-edge



Triangular Mesh



Half-edge representation

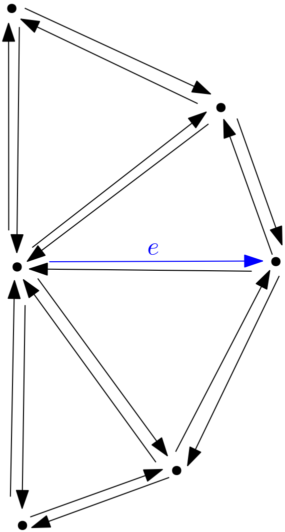


Pemb representation

Pemb encapsulation as Half-edge: Classic queries

We encapsulates the basic queries of the Half-edge data structure as:

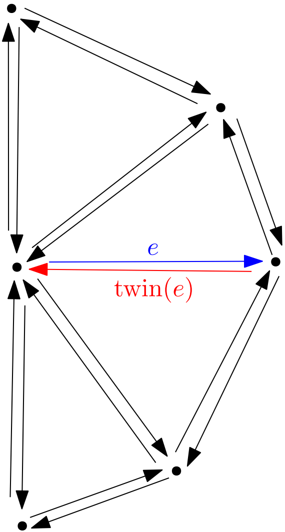
- twin(e):
- next(e):
- prev(e):
- origin(e):
- target(e):
- face(e):



Pemb encapsulation as Half-edge: Classic queries

We encapsulates the basic queries of the Half-edge data structure as:

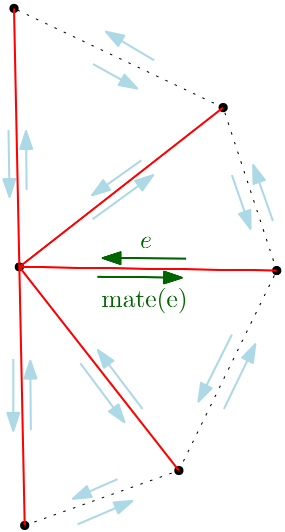
- twin(e):
- next(e):
- prev(e):
- origin(e):
- target(e):
- face(e):



Pemb encapsulation as Half-edge: Classic queries

We encapsulates the basic queries of the Half-edge data structure as:

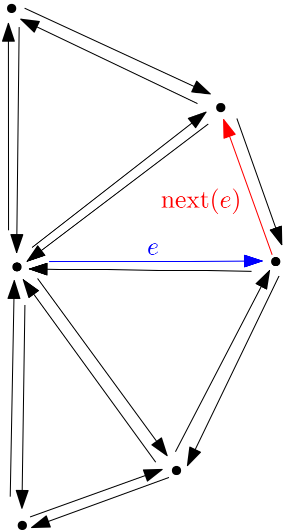
- twin(e): pemb::mate(e)
- next(e):
- prev(e):
- origin(e):
- target(e):
- face(e):



Pemb encapsulation as Half-edge: Classic queries

We encapsulates the basic queries of the Half-edge data structure as:

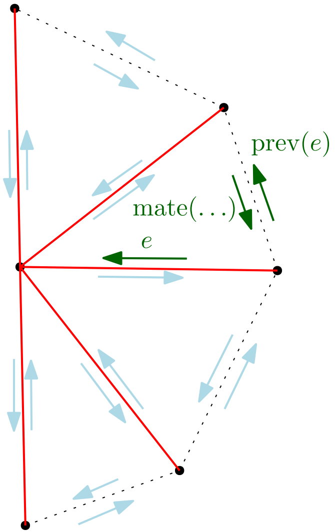
- twin(e): pemb::mate(e)
- next(e):
- prev(e):
- origin(e):
- target(e):
- face(e):



Pemb encapsulation as Half-edge: Classic queries

We encapsulates the basic queries of the Half-edge data structure as:

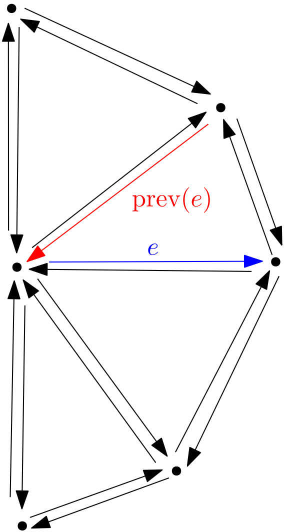
- twin(e): pemb::mate(e)
- next(e): pemb::mate(pemb::prev(e))
- prev(e):
- origin(e):
- target(e):
- face(e):



Pemb encapsulation as Half-edge: Classic queries

We encapsulates the basic queries of the Half-edge data structure as:

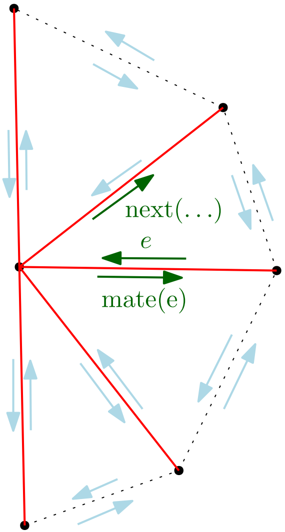
- twin(e): pemb::mate(e)
- next(e): pemb::mate(pemb::prev(e))
- prev(e):
- origin(e):
- target(e):
- face(e):



Pemb encapsulation as Half-edge: Classic queries

We encapsulates the basic queries of the Half-edge data structure as:

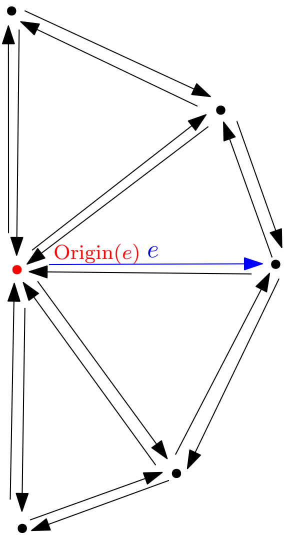
- twin(e): pemb::mate(e)
- next(e): pemb::mate(pemb::prev(e))
- prev(e): pemb::next(pemb::mate(e))
- origin(e):
- target(e):
- face(e):



Pemb encapsulation as Half-edge: Classic queries

We encapsulates the basic queries of the Half-edge data structure as:

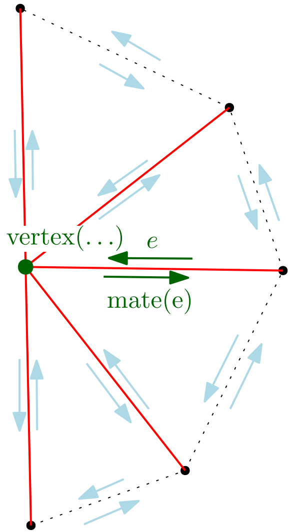
- twin(e): pemb::mate(e)
- next(e): pemb::mate(pemb::prev(e))
- prev(e): pemb::next(pemb::mate(e))
- origin(e):
- target(e):
- face(e):



Pemb encapsulation as Half-edge: Classic queries

We encapsulates the basic queries of the Half-edge data structure as:

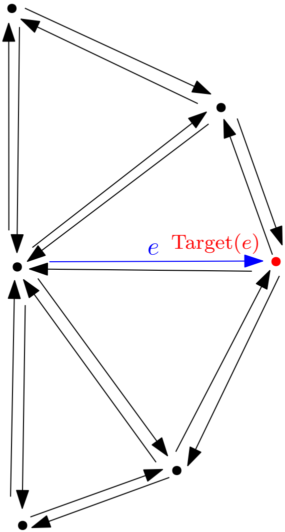
- twin(e): pemb::mate(e)
- next(e): pemb::mate(pemb::prev(e))
- prev(e): pemb::next(pemb::mate(e))
- origin(e): pemb::vertex(pemb::mate(e))
- target(e):
- face(e):



Pemb encapsulation as Half-edge: Classic queries

We encapsulates the basic queries of the Half-edge data structure as:

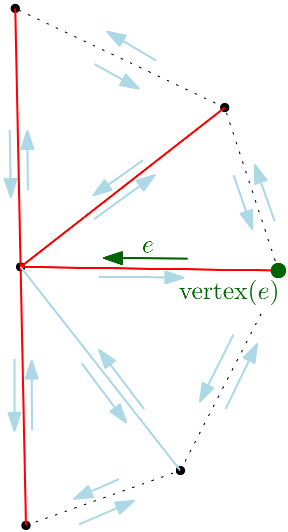
- twin(e): pemb::mate(e)
- next(e): pemb::mate(pemb::prev(e))
- prev(e): pemb::next(pemb::mate(e))
- origin(e): pemb::vertex(pemb::mate(e))
- target(e):
- face(e):



Pemb encapsulation as Half-edge: Classic queries

We encapsulates the basic queries of the Half-edge data structure as:

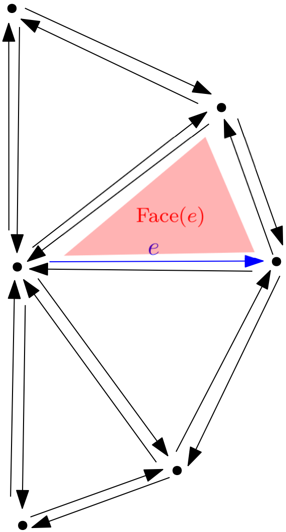
- twin(e): pemb::mate(e)
- next(e): pemb::mate(pemb::prev(e))
- prev(e): pemb::next(pemb::mate(e))
- origin(e): pemb::vertex(pemb::mate(e))
- target(e): pemb::vertex(e)
- face(e):



Pemb encapsulation as Half-edge: Classic queries

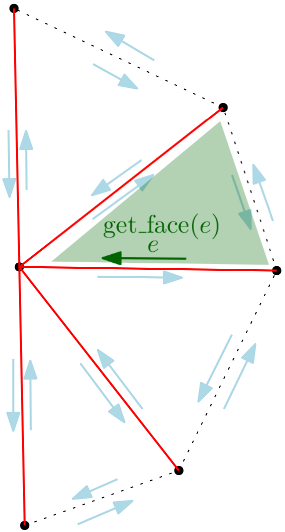
We encapsulates the basic queries of the Half-edge data structure as:

- twin(e): pemb::mate(e)
- next(e): pemb::mate(pemb::prev(e))
- prev(e): pemb::next(pemb::mate(e))
- origin(e): pemb::vertex(pemb::mate(e))
- target(e): pemb::vertex(e)
- face(e):



We encapsulates the basic queries of the Half-edge data structure as:

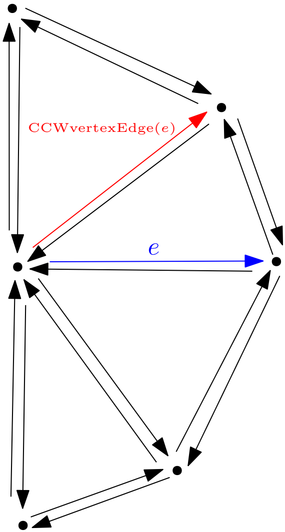
- twin(e): pemb::mate(e)
- next(e): pemb::mate(pemb::prev(e))
- prev(e): pemb::next(pemb::mate(e))
- origin(e): pemb::vertex(pemb::mate(e))
- target(e): pemb::vertex(e)
- face(e): pemb::get_face(e)



Pemb encapsulation as Half-edge: Extra queries

We encapsulates the additional queries of the Half-edge data structure as:

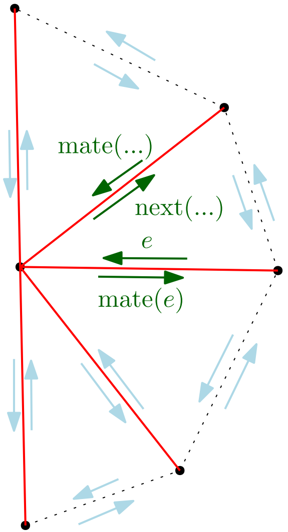
- CCWvertexEdge(e):
- CWvertexEdge(e):
- isBorder(e):
- incidentHalfedge(f):
- edgeOfVertex(v):
- degree(v):



Pemb encapsulation as Half-edge: Extra queries

We encapsulates the additional queries of the Half-edge data structure as:

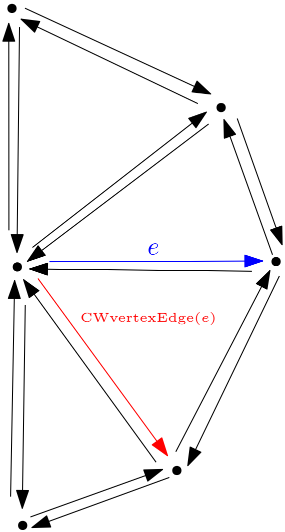
- CCWvertexEdge(e): pemb::mate(pemb::next(pemb:mate(e)))
- CWvertexEdge(e):
- isBorder(e):
- incidentHalfedge(f):
- edgeOfVertex(v):
- degree(v):



Pemb encapsulation as Half-edge: Extra queries

We encapsulates the additional queries of the Half-edge data structure as:

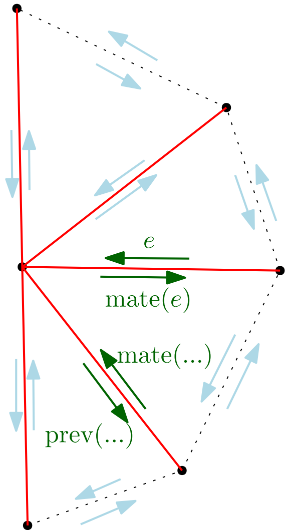
- CCWvertexEdge(e): pemb::mate(pemb::next(pemb:mate(e)))
- CWvertexEdge(e):
- isBorder(e):
- incidentHalfedge(f):
- edgeOfVertex(v):
- degree(v):



Pemb encapsulation as Half-edge: Extra queries

We encapsulates the additional queries of the Half-edge data structure as:

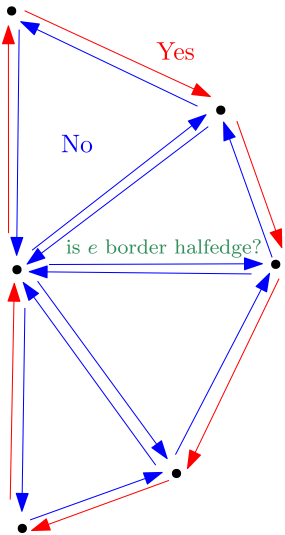
- CCWvertexEdge(e): pemb::mate(pemb::next(pemb:mate(e)))
- CWvertexEdge(e): pemb::mate(pemb::prev(pemb:mate(e)))
- isBorder(e):
- incidentHalfedge(f):
- edgeOfVertex(v):
- degree(v):



Pemb encapsulation as Half-edge: Extra queries

We encapsulates the additional queries of the Half-edge data structure as:

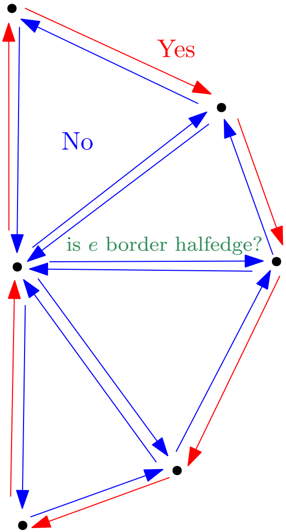
- CCWvertexEdge(e): pemb::mate(pemb::next(pemb:mate(e)))
- CWvertexEdge(e): pemb::mate(pemb::prev(pemb:mate(e)))
- isBorder(e):
- incidentHalfedge(f):
- edgeOfVertex(v):
- degree(v):



Pemb encapsulation as Half-edge: Extra queries

We encapsulates the additional queries of the Half-edge data structure as:

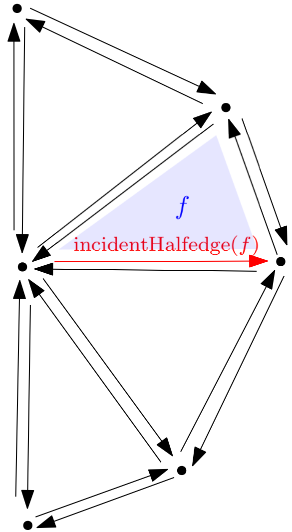
- CCWvertexEdge(e): pemb::mate(pemb::next(pemb:mate(e)))
- CWvertexEdge(e): pemb::mate(pemb::prev(pemb:mate(e)))
- isBorder(e): return true if pemb::get_face(e) returns the id of the outer face. Otherwise, return false
- incidentHalfedge(f):
- edgeOfVertex(v):
- degree(v):



Pemb encapsulation as Half-edge: Extra queries

We encapsulates the additional queries of the Half-edge data structure as:

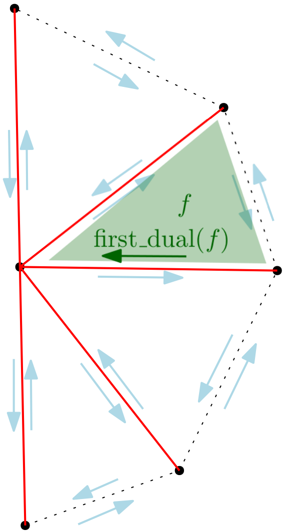
- CCWvertexEdge(e): pemb::mate(pemb::next(pemb:mate(e)))
- CWvertexEdge(e): pemb::mate(pemb::prev(pemb:mate(e)))
- isBorder(e): return true if pemb::get_face(e) returns the id of the outer face. Otherwise, return false
- incidentHalfedge(f):
- edgeOfVertex(v):
- degree(v):



Pemb encapsulation as Half-edge: Extra queries

We encapsulates the additional queries of the Half-edge data structure as:

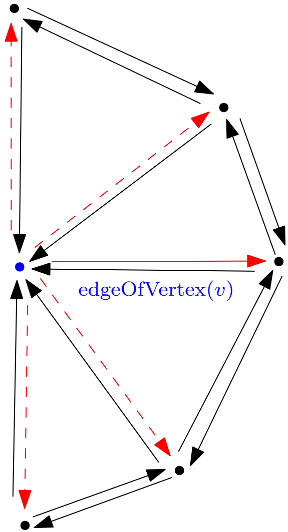
- CCWvertexEdge(e): pemb::mate(pemb::next(pemb:mate(e)))
- CWvertexEdge(e): pemb::mate(pemb::prev(pemb:mate(e)))
- isBorder(e): return true if pemb::get_face(e) returns the id of the outer face. Otherwise, return false
- incidentHalfedge(f): pemb::first_dual(f)
- edgeOfVertex(v):
- degree(v):



Pemb encapsulation as Half-edge: Extra queries

We encapsulates the additional queries of the Half-edge data structure as:

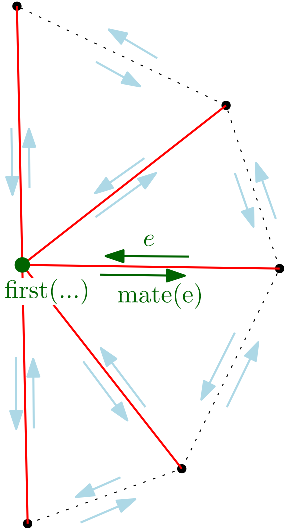
- CCWvertexEdge(e): pemb::mate(pemb::next(pemb:mate(e)))
- CWvertexEdge(e): pemb::mate(pemb::prev(pemb:mate(e)))
- isBorder(e): return true if pemb::get_face(e) returns the id of the outer face. Otherwise, return false
- incidentHalfedge(f): pemb::first_dual(f)
- edgeOfVertex(v):
- degree(v):



Pemb encapsulation as Half-edge: Extra queries

We encapsulates the additional queries of the Half-edge data structure as:

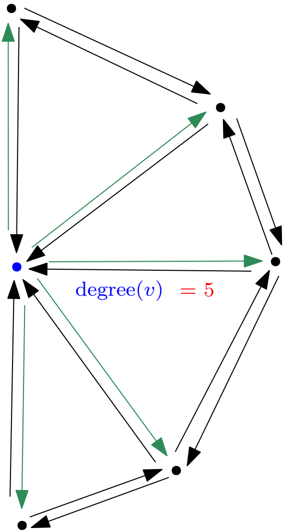
- CCWvertexEdge(e): pemb::mate(pemb::next(pemb:mate(e)))
- CWvertexEdge(e): pemb::mate(pemb::prev(pemb:mate(e)))
- isBorder(e): return true if pemb::get_face(e) returns the id of the outer face. Otherwise, return false
- incidentHalfedge(f): pemb::first_dual(f)
- edgeOfVertex(v): pemb::mate(pemb::first(v))
- degree(v):



Pemb encapsulation as Half-edge: Extra queries

We encapsulates the additional queries of the Half-edge data structure as:

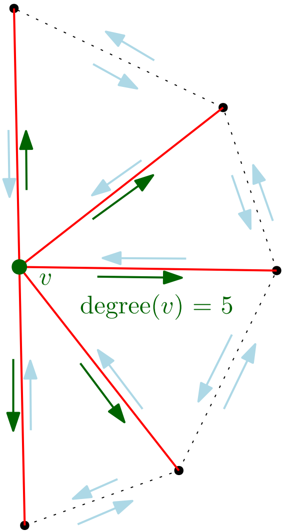
- CCWvertexEdge(e): pemb::mate(pemb::next(pemb:mate(e)))
- CWvertexEdge(e): pemb::mate(pemb::prev(pemb:mate(e)))
- isBorder(e): return true if pemb::get_face(e) returns the id of the outer face. Otherwise, return false
- incidentHalfedge(f): pemb::first_dual(f)
- edgeOfVertex(v): pemb::mate(pemb::first(v))
- degree(v):



Pemb encapsulation as Half-edge: Extra queries

We encapsulates the additional queries of the Half-edge data structure as:

- CCWvertexEdge(e): pemb::mate(pemb::next(pemb:mate(e)))
- CWvertexEdge(e): pemb::mate(pemb::prev(pemb:mate(e)))
- isBorder(e): return true if pemb::get_face(e) returns the id of the outer face. Otherwise, return false
- incidentHalfedge(f): pemb::first_dual(f)
- edgeOfVertex(v): pemb::mate(pemb::first(v))
- degree(v): pemb::degree(v)



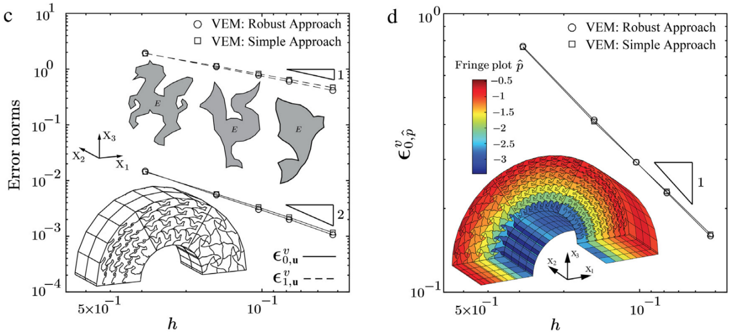
Polylla

For 1M of vertices Polylla takes:

- 1.3 seconds in the face based DS.
- 4.6 seconds in the half-edge DS.
- 137.242 seconds in the Pemb DS.

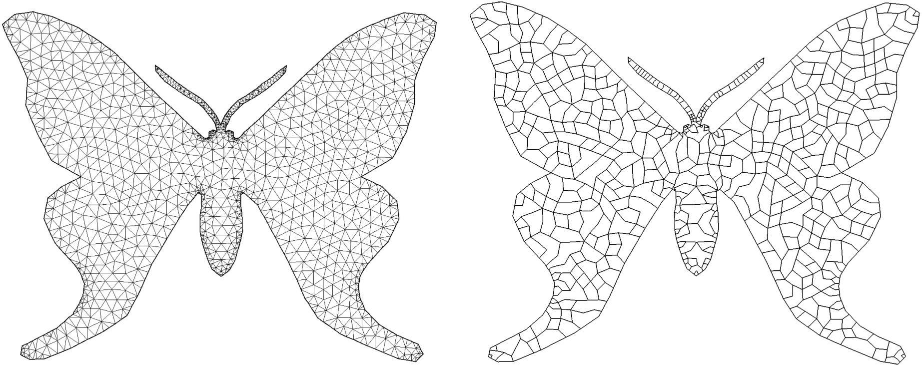
New numerical methods

There are new numerical methods that uses polygons/polyhedrons of arbitrary shape as the Virtual Element method.



Source: H. Chi, L. Beirão da Veiga, & G.H. Paulino (2017). Some basic formulations of the virtual element method (VEM) for finite deformations. Computer Methods in Applied Mechanics and Engineering, 318, 148-192.

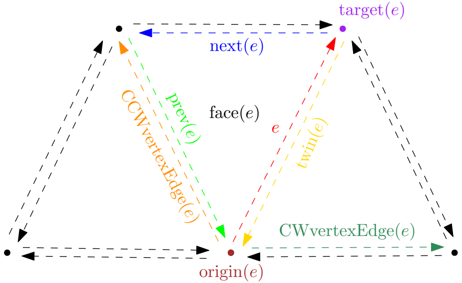
Polylla mesh generator



Source: Salinas-Fernández, S., Hitschfeld-Kahler, N., Ortiz-Bernardin, A., & Si, H. (2022). POLYLLA: Polygonal meshing algorithm based on terminal-edge regions. Engineering with Computers, 38(5), 4545–4567. https://doi.org/10.1007/s00366-022-01643-4

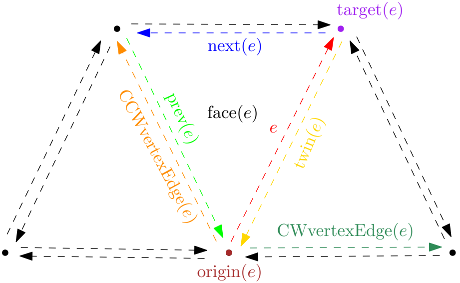
Half-edge data queries

- Next()
- Prev()
- Twin()
- Origin()
- Target()
- Face()



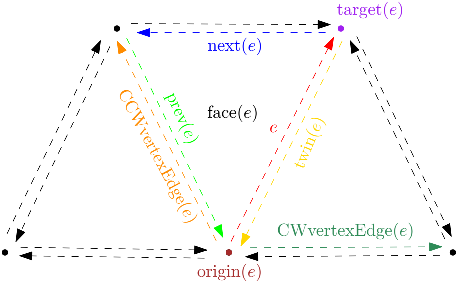
Half-edge data queries

- `Next()`
- `Prev()`
- `Twin()`
- `Origin()`
- `Target()`
- `Face()`



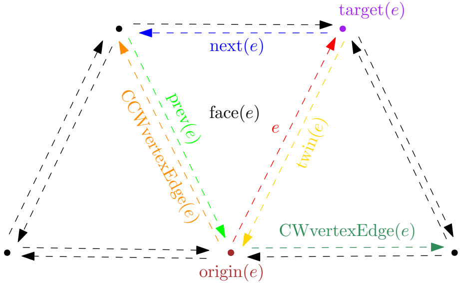
Half-edge data queries

- Next()
- Prev()
- Twin()
- Origin()
- Target()
- Face()



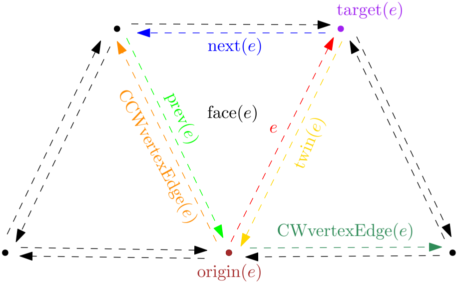
Half-edge data queries

- `Next()`
- `Prev()`
- `Twin()`
- `Origin()`
- `Target()`
- `Face()`



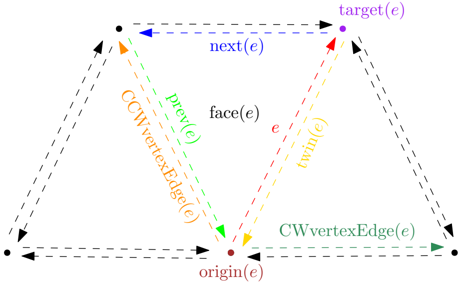
Half-edge data queries

- `Next()`
- `Prev()`
- `Twin()`
- `Origin()`
- `Target()`
- `Face()`

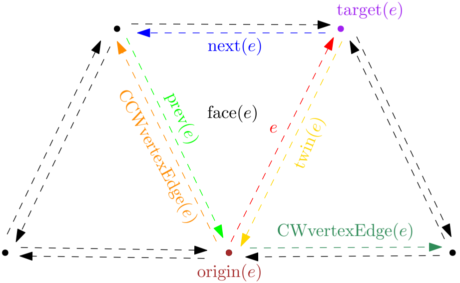


Half-edge data queries

- `Next()`
- `Prev()`
- `Twin()`
- `Origin()`
- `Target()`
- `Face()`



- CCWvertexEdge(e)
- CWvertexEdge(e)
- edgeOfVertex(v)
- incidentHalfEdge(f)
- isBorder(e)
- degree(v)



Half-edge AoS extended Queries

- CCWvertexEdge(e): twin(next(e)).
- CWvertexEdge(e): twin(prev(e)).
- incidentHalfEdge(f): Half-edge at index $3f$ in the array of half-edges.
- length(e): Euclidean distance of the coordinates of origin(e) and target(e).
- degree(e): Using the query CCWvertexEdge(e), iterate over the neighbors of origin(e) until reaching e.

Half-edge AoS extended Queries

- CCWvertexEdge(e): twin(next(e)).
- CWvertexEdge(e): twin(prev(e)).
- incidentHalfEdge(f): Half-edge at index $3f$ in the array of half-edges.
- length(e): Euclidean distance of the coordinates of origin(e) and target(e).
- degree(e): Using the query CCWvertexEdge(e), iterate over the neighbors of origin(e) until reaching e.

Half-edge AoS extended Queries

- CCWvertexEdge(e): twin(next(e)).
- CWvertexEdge(e): twin(prev(e)).
- incidentHalfEdge(f): Half-edge at index $3f$ in the array of half-edges.
- length(e): Euclidean distance of the coordinates of origin(e) and target(e).
- degree(e): Using the query CCWvertexEdge(e), iterate over the neighbors of origin(e) until reaching e.

Half-edge AoS extended Queries

- CCWvertexEdge(e): twin(next(e)).
- CWvertexEdge(e): twin(prev(e)).
- incidentHalfEdge(f): Half-edge at index $3f$ in the array of half-edges.
- length(e): Euclidean distance of the coordinates of origin(e) and target(e).
- degree(e): Using the query CCWvertexEdge(e), iterate over the neighbors of origin(e) until reaching e.

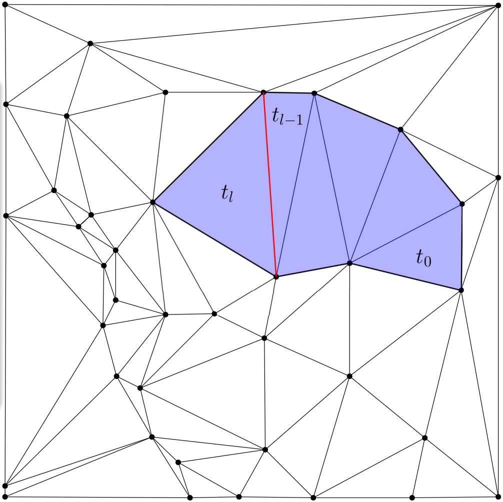
Half-edge AoS extended Queries

- CCWvertexEdge(e): twin(next(e)).
- CWvertexEdge(e): twin(prev(e)).
- incidentHalfEdge(f): Half-edge at index $3f$ in the array of half-edges.
- length(e): Euclidean distance of the coordinates of origin(e) and target(e).
- degree(e): Using the query CCWvertexEdge(e), iterate over the neighbors of origin(e) until reaching e.

Longest-edge propagation path (Lepp) 2D

Longest Edge Propagation Path 2D †

The lepp of a triangle t_0 is the ordered list of all adjacent triangles $t_0, t_1, ..., t_l$, such that t_i is the longest edge neighbor triangle of t_{i-1} by the longest-edge of t_{i-1}, for $i = 1, 2, ..., l$.

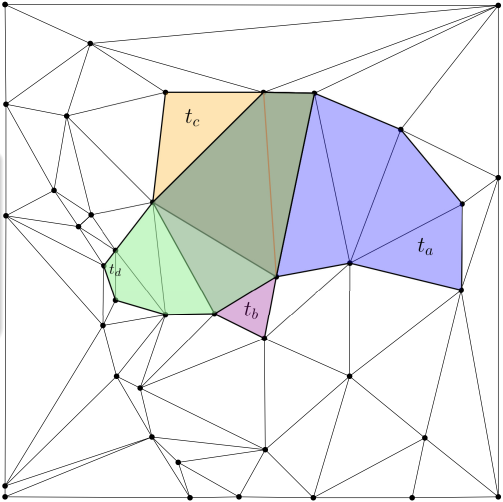


† **Source** Maria-Cecilia Rivara. New longest-edge algorithms for the refinement and/or improvement of unstructured triangulations. International Journal for Numerical Methods in Engineering, 40(18):3313-3324, 1997.

Terminal-edge regions

Terminal-edge region ‡

A *terminal-edge region R* is a region formed by the union of all triangles t_i such that Lepp(t_i) has the same terminal-edge.

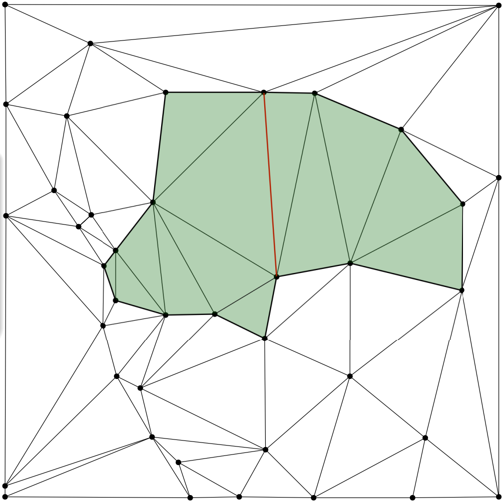


‡R. Alonso, J. Ojeda, N. Hitschfeld, C. Hervías, and L.E. Campusano. Delaunay based algorithm for finding polygonal voids in planar point sets. Astronomy and Computing, 22:48 - 62, 2018. 20

Terminal-edge regions

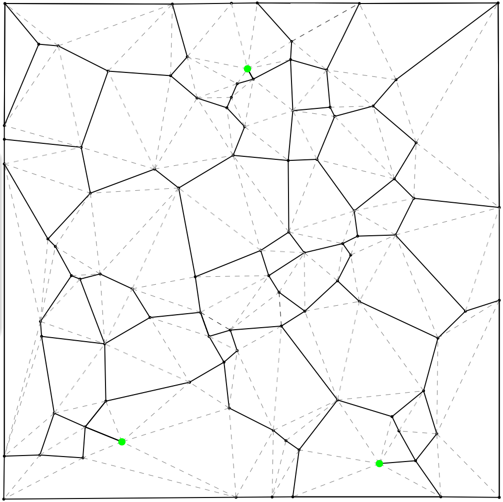
Terminal-edge region ‡

A *terminal-edge region R* is a region formed by the union of all triangles t_i such that Lepp(t_i) has the same terminal-edge.



‡R. Alonso, J. Ojeda, N. Hitschfeld, C. Hervías, and L.E. Campusano. Delaunay based algorithm for finding polygonal voids in planar point sets. Astronomy and Computing, 22:48 - 62, 2018. 20

Frontier-edges

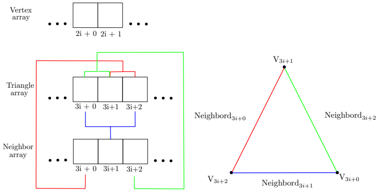


Frontier-edges ‡

A frontier-edge is an edge that is shared by two triangles, each one belonging to a different terminal-edge region.

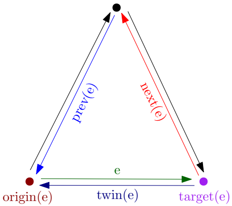
‡R. Alonso, J. Ojeda, N. Hitschfeld, C. Hervías, and L.E. Campusano. Delaunay based algorithm for finding polygonal voids in planar point sets. Astronomy and Computing, 22:48 - 62, 2018. 20

Old Polylla



- Triangle based data structure
- Hard to calculate edge adjacencies
- No edge iteration

New Polylla



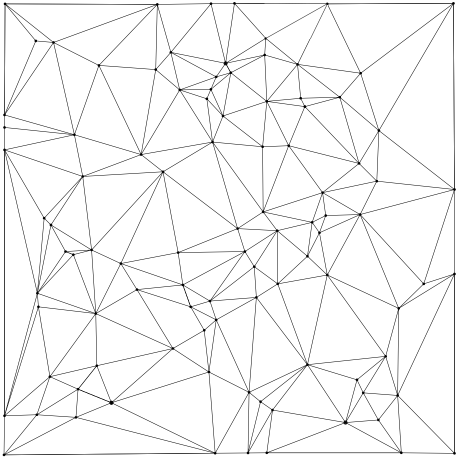
- Edge based data structure
- Easy navigation inside mesh
- Edge and Triangle navigation
- Easy to read

Polylla 2D algorithm

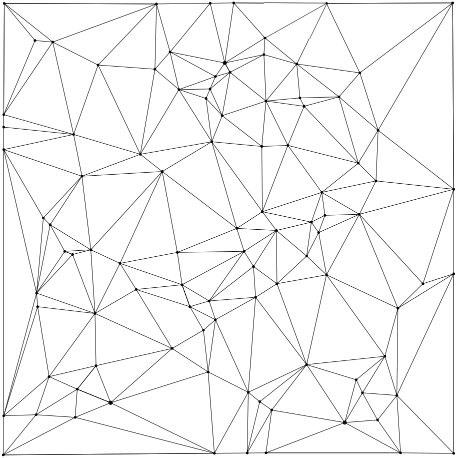
- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh

Algorithm has three main phases

1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase



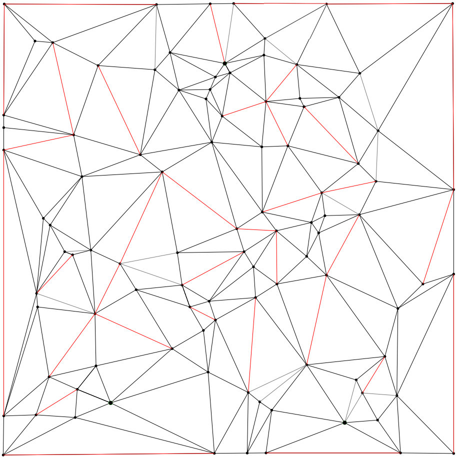
Label Phase



Algorithm Label phase

Require: Half-edge data structure `HalfEdge`
Ensure: Bitvectors `frontie-edge` and `max-edge`, and vector `seed-list`

Label Phase

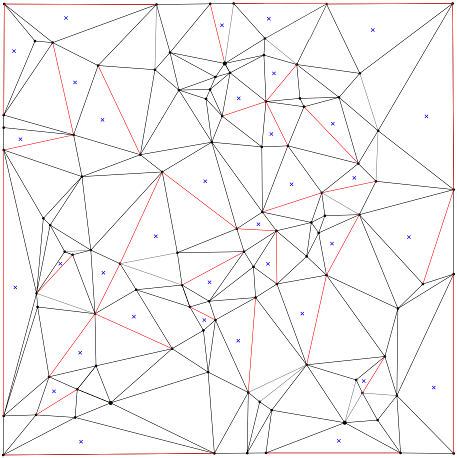


Algorithm Label phase

Require: Half-edge data structure `HalfEdge`
Ensure: Bitvectors `frontie-edge` and `max-edge`, and vector `seed-list`

  for all triangle t **in** `HalfEdge` **do**
    e = incidentHalfedge(t_i)
    Label the max edge between e,next(e), prev(e)
  end for

Label Phase

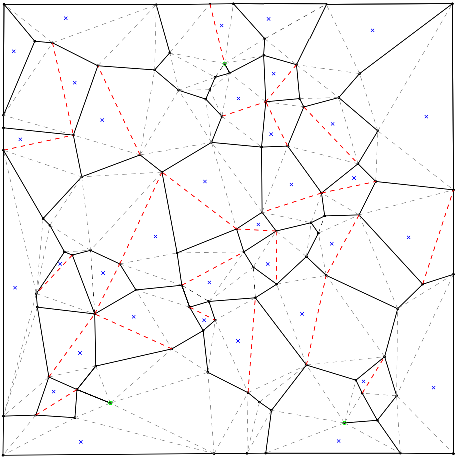


Algorithm Label phase

Require: Half-edge data structure `HalfEdge`
Ensure: Bitvectors `frontie-edge` and `max-edge`, and vector `seed-list`

for all triangle t in `HalfEdge` **do**
  e = incidentHalfedge(t_i)
  Label the max edge between e,next(e), prev(e)
end for
for all half-edge e in `HalfEdges` **do**
  if e is terminal-edge or border terminal-edge **then**
    Store the id of e or twin(e) in the `seed list`
  end if
end for

Label Phase



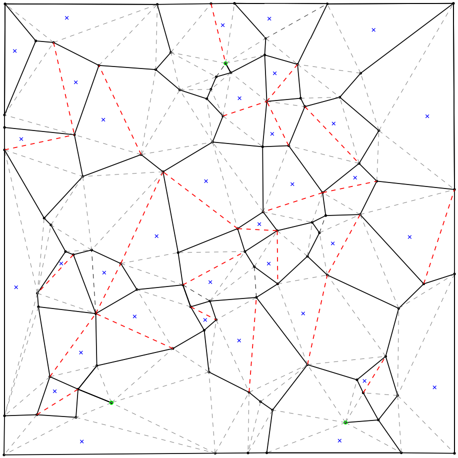
Algorithm Label phase

Require: Half-edge data structure `HalfEdge`
Ensure: Bitvectors `frontie-edge` and `max-edge`, and vector `seed-list`

for all triangle t in `HalfEdge` **do**
  e = incidentHalfedge(t_i)
  Label the max edge between e,next(e), prev(e)
end for
for all half-edge e in `HalfEdges` **do**
  if e is terminal-edge or border terminal-edge **then**
    Store the id of e or twin(e) in the `seed list`
  end if
  if e and twin(e) are not in `max-edge` **then**
    Mark e in `frontier-edge`
  else if e or twin(e) are border edges **then**
    Mark e in `frontier-edge`
  end if
end for

Polylla 2D algorithm

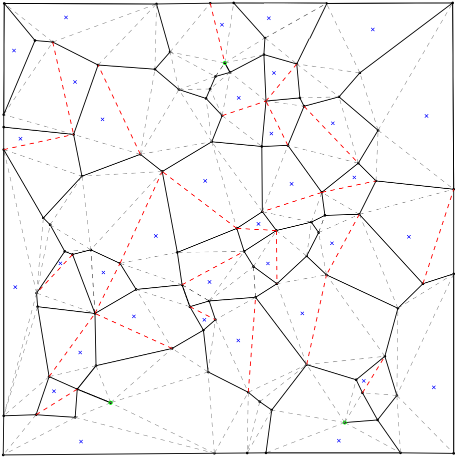
- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh



Algorithm has three main phases

1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase

Traversal Phase

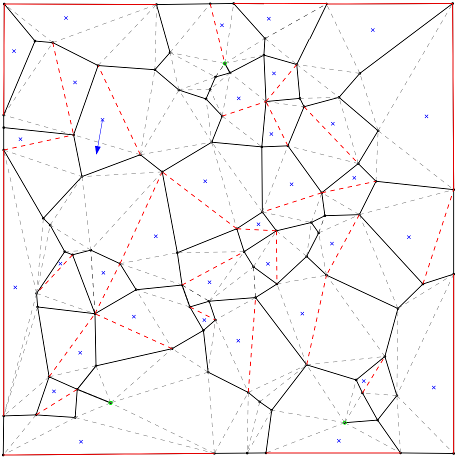


Algorithm Polygon construction

Require: Seed edge e of a terminal-edge region
Ensure: Arbitrary shape polygon P

- $P \leftarrow \emptyset$
- **while** e is not a frontier-edge **do**
  - $e \leftarrow$ CWvertexEdge(e)
- **end while**
- $e_{init} \leftarrow e$
- $e_{curr} \leftarrow$ next(e)
- $P \leftarrow P \cup$ origin(e)
- **while** $e_{init} \neq e_{curr}$ **do**
  - **while** e_{curr} is not a frontier-edge **do**
    - $e_{curr} \leftarrow$ CWvertexEdge(e)
  - **end while**
  - $e_{curr} \leftarrow$ next(e_{curr})
  - $P \leftarrow P \cup$ origin(e_{curr})
- **end while**
- **return** P

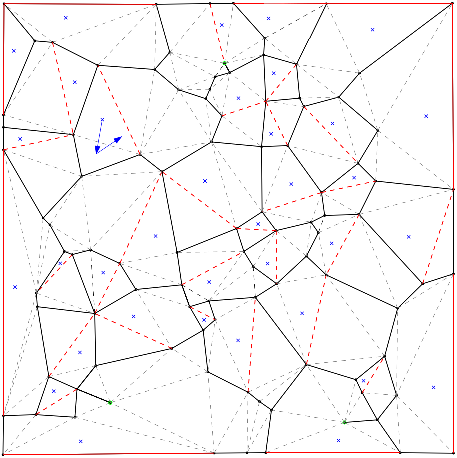
Traversal Phase



Algorithm Polygon construction

Require: Seed edge e of a terminal-edge region
Ensure: Arbitrary shape polygon P

```
P ← ∅
while e is not a frontier-edge do
    e ← CWvertexEdge(e)
end while
e_init ← e
e_curr ← next(e)
P ← P ∪ origin(e)
while e_init ≠ e_curr do
    while e_curr is not a frontier-edge do
        e_curr ← CWvertexEdge(e)
    end while
    e_curr ← next(e_curr)
    P ← P ∪ origin(e_curr)
end while
return P
```

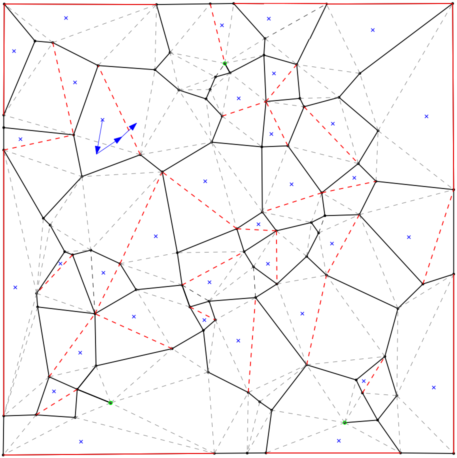


Algorithm Polygon construction

Require: Seed edge e of a terminal-edge region
Ensure: Arbitrary shape polygon P

```
P ← ∅
while e is not a frontier-edge do
    e ← CWvertexEdge(e)
end while
e_init ← e
e_curr ← next(e)
P ← P ∪ origin(e)
while e_init ≠ e_curr do
    while e_curr is not a frontier-edge do
        e_curr ← CWvertexEdge(e)
    end while
    e_curr ← next(e_curr)
    P ← P ∪ origin(e_curr)
end while
return P
```

Traversal Phase

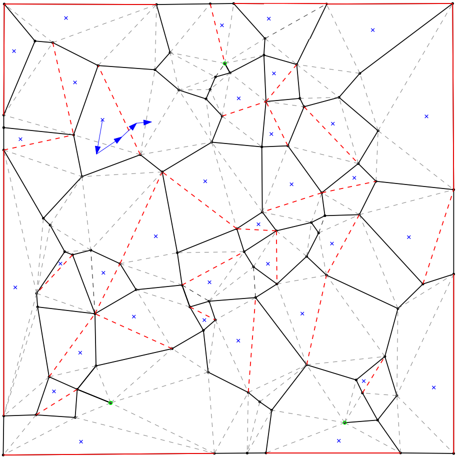


Algorithm Polygon construction

Require: Seed edge e of a terminal-edge region
Ensure: Arbitrary shape polygon P

```
P ← ∅
while e is not a frontier-edge do
    e ← CWvertexEdge(e)
end while
e_init ← e
e_curr ← next(e)
P ← P ∪ origin(e)
while e_init ≠ e_curr do
    while e_curr is not a frontier-edge do
        e_curr ← CWvertexEdge(e)
    end while
    e_curr ← next(e_curr)
    P ← P ∪ origin(e_curr)
end while
return P
```

Traversal Phase

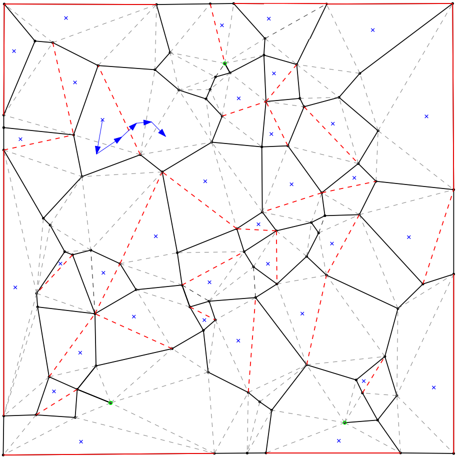


Algorithm Polygon construction

Require: Seed edge e of a terminal-edge region
Ensure: Arbitrary shape polygon P

```
P ← ∅
while e is not a frontier-edge do
    e ← CWvertexEdge(e)
end while
e_init ← e
e_curr ← next(e)
P ← P ∪ origin(e)
while e_init ≠ e_curr do
    while e_curr is not a frontier-edge do
        e_curr ← CWvertexEdge(e)
    end while
    e_curr ← next(e_curr)
    P ← P ∪ origin(e_curr)
end while
return P
```

Traversal Phase

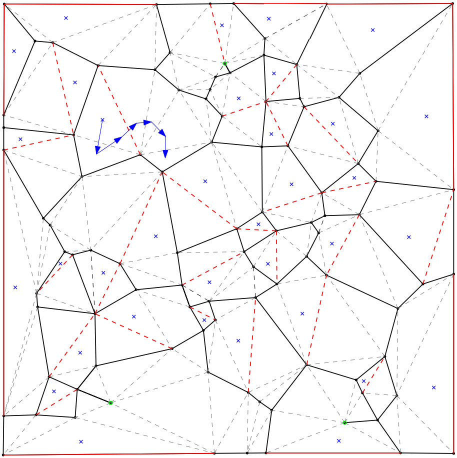


Algorithm Polygon construction

Require: Seed edge e of a terminal-edge region
Ensure: Arbitrary shape polygon P

```
P ← ∅
while e is not a frontier-edge do
    e ← CWvertexEdge(e)
end while
e_init ← e
e_curr ← next(e)
P ← P ∪ origin(e)
while e_init ≠ e_curr do
    while e_curr is not a frontier-edge do
        e_curr ← CWvertexEdge(e)
    end while
    e_curr ← next(e_curr)
    P ← P ∪ origin(e_curr)
end while
return P
```

Traversal Phase

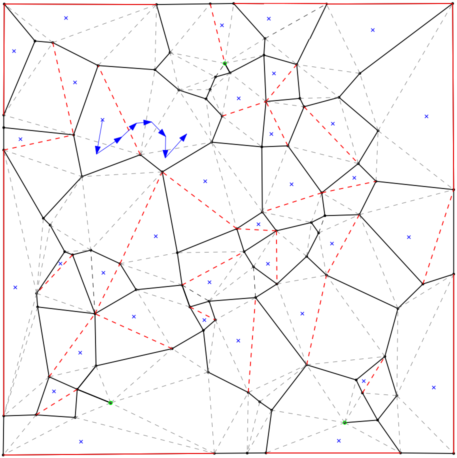


Algorithm Polygon construction

Require: Seed edge e of a terminal-edge region
Ensure: Arbitrary shape polygon P

```
P ← ∅
while e is not a frontier-edge do
    e ← CWvertexEdge(e)
end while
e_init ← e
e_curr ← next(e)
P ← P ∪ origin(e)
while e_init ≠ e_curr do
    while e_curr is not a frontier-edge do
        e_curr ← CWvertexEdge(e)
    end while
    e_curr ← next(e_curr)
    P ← P ∪ origin(e_curr)
end while
return P
```

Traversal Phase

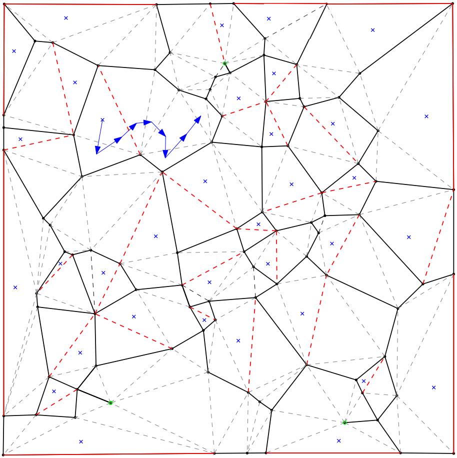


Algorithm Polygon construction

Require: Seed edge e of a terminal-edge region
Ensure: Arbitrary shape polygon P

```
P ← ∅
while e is not a frontier-edge do
    e ← CWvertexEdge(e)
end while
e_init ← e
e_curr ← next(e)
P ← P ∪ origin(e)
while e_init ≠ e_curr do
    while e_curr is not a frontier-edge do
        e_curr ← CWvertexEdge(e)
    end while
    e_curr ← next(e_curr)
    P ← P ∪ origin(e_curr)
end while
return P
```

Traversal Phase

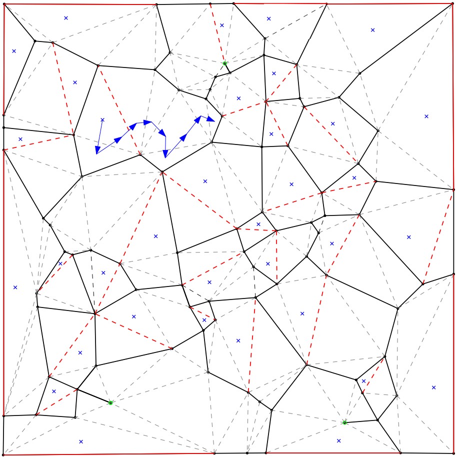


Algorithm Polygon construction

Require: Seed edge e of a terminal-edge region
Ensure: Arbitrary shape polygon P

$P \leftarrow \emptyset$
while e is not a frontier-edge **do**
  $e \leftarrow$ CWvertexEdge(e)
end while
$e_{init} \leftarrow e$
$e_{curr} \leftarrow$ next(e)
$P \leftarrow P \cup$ origin(e)
while $e_{init} \neq e_{curr}$ **do**
  while e_{curr} is not a frontier-edge **do**
    $e_{curr} \leftarrow$ CWvertexEdge(e)
  end while
  $e_{curr} \leftarrow$ next(e_{curr})
  $P \leftarrow P \cup$ origin(e_{curr})
end while
return P

Traversal Phase

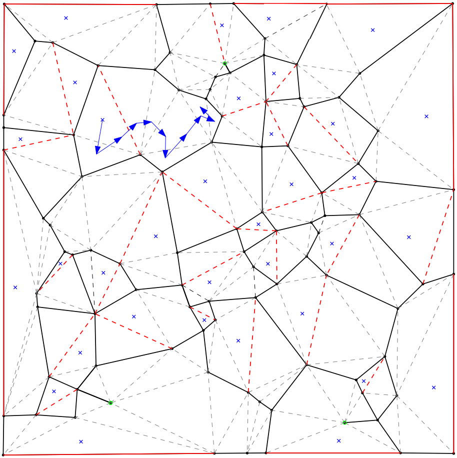


Algorithm Polygon construction

Require: Seed edge e of a terminal-edge region
Ensure: Arbitrary shape polygon P

$P \leftarrow \emptyset$
while e is not a frontier-edge **do**
  $e \leftarrow$ CWvertexEdge(e)
end while
$e_{init} \leftarrow e$
$e_{curr} \leftarrow$ next(e)
$P \leftarrow P \cup$ origin(e)
while $e_{init} \neq e_{curr}$ **do**
  while e_{curr} is not a frontier-edge **do**
    $e_{curr} \leftarrow$ CWvertexEdge(e)
  end while
  $e_{curr} \leftarrow$ next(e_{curr})
  $P \leftarrow P \cup$ origin(e_{curr})
end while
return P

Traversal Phase

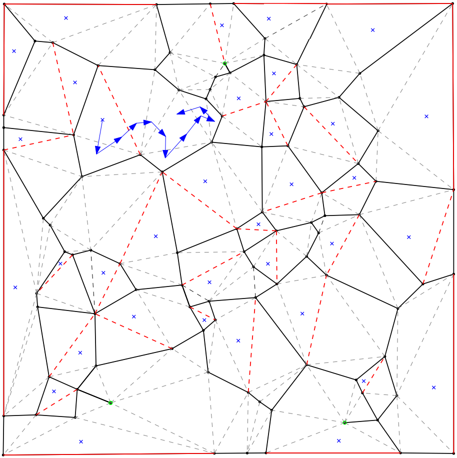


Algorithm Polygon construction

Require: Seed edge e of a terminal-edge region
Ensure: Arbitrary shape polygon P

```
P ← ∅
while e is not a frontier-edge do
    e ← CWvertexEdge(e)
end while
e_init ← e
e_curr ← next(e)
P ← P ∪ origin(e)
while e_init ≠ e_curr do
    while e_curr is not a frontier-edge do
        e_curr ← CWvertexEdge(e)
    end while
    e_curr ← next(e_curr)
    P ← P ∪ origin(e_curr)
end while
return P
```

Traversal Phase

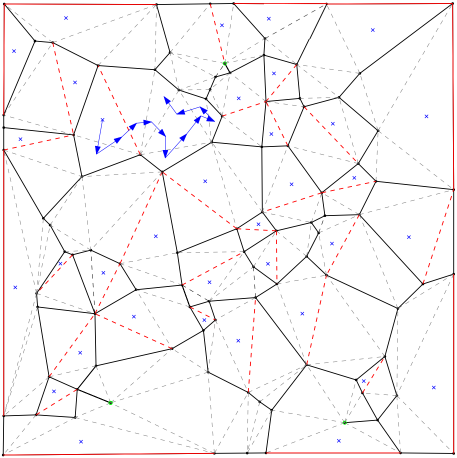


Algorithm Polygon construction

Require: Seed edge e of a terminal-edge region
Ensure: Arbitrary shape polygon P

```
P ← ∅
while e is not a frontier-edge do
    e ← CWvertexEdge(e)
end while
e_init ← e
e_curr ← next(e)
P ← P ∪ origin(e)
while e_init ≠ e_curr do
    while e_curr is not a frontier-edge do
        e_curr ← CWvertexEdge(e)
    end while
    e_curr ← next(e_curr)
    P ← P ∪ origin(e_curr)
end while
return P
```

Traversal Phase

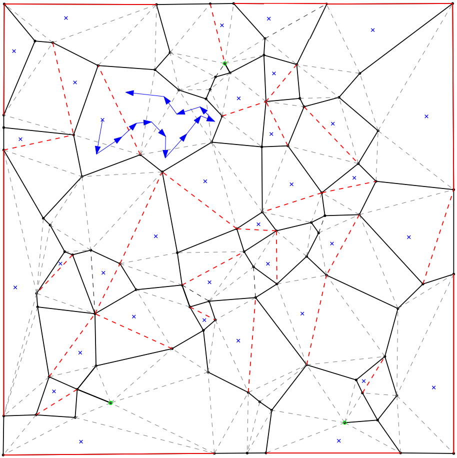


Algorithm Polygon construction

Require: Seed edge e of a terminal-edge region
Ensure: Arbitrary shape polygon P

```
P ← ∅
while e is not a frontier-edge do
    e ← CWvertexEdge(e)
end while
e_init ← e
e_curr ← next(e)
P ← P ∪ origin(e)
while e_init ≠ e_curr do
    while e_curr is not a frontier-edge do
        e_curr ← CWvertexEdge(e)
    end while
    e_curr ← next(e_curr)
    P ← P ∪ origin(e_curr)
end while
return P
```

Traversal Phase

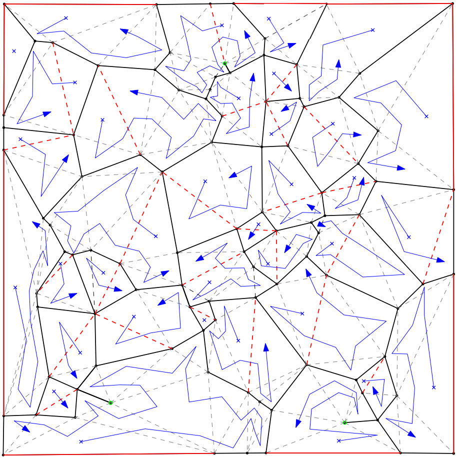


Algorithm Polygon construction

Require: Seed edge e of a terminal-edge region
Ensure: Arbitrary shape polygon P

```
P ← ∅
while e is not a frontier-edge do
    e ← CWvertexEdge(e)
end while
e_init ← e
e_curr ← next(e)
P ← P ∪ origin(e)
while e_init ≠ e_curr do
    while e_curr is not a frontier-edge do
        e_curr ← CWvertexEdge(e)
    end while
    e_curr ← next(e_curr)
    P ← P ∪ origin(e_curr)
end while
return P
```

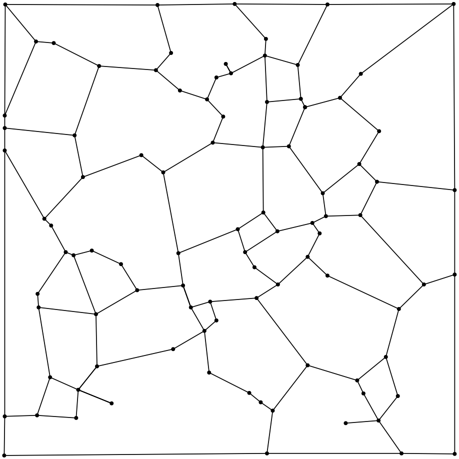
Traversal Phase



Algorithm Polygon construction

Require: Seed edge e of a terminal-edge region
Ensure: Arbitrary shape polygon P

```
P ← ∅
while e is not a frontier-edge do
    e ← CWvertexEdge(e)
end while
e_init ← e
e_curr ← next(e)
P ← P ∪ origin(e)
while e_init ≠ e_curr do
    while e_curr is not a frontier-edge do
        e_curr ← CWvertexEdge(e)
    end while
    e_curr ← next(e_curr)
    P ← P ∪ origin(e_curr)
end while
return P
```



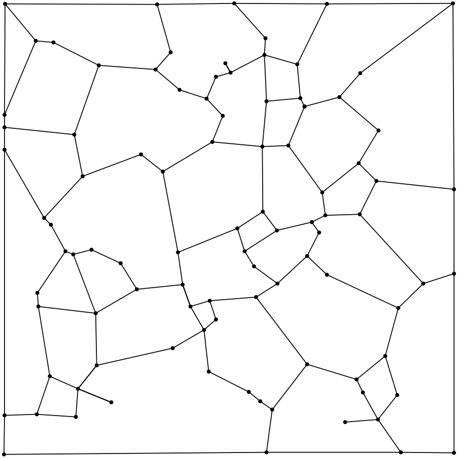
Algorithm Polygon construction

Require: Seed edge e of a terminal-edge region
Ensure: Arbitrary shape polygon P

```
P ← ∅
while e is not a frontier-edge do
    e ← CWvertexEdge(e)
end while
e_init ← e
e_curr ← next(e)
P ← P ∪ origin(e)
while e_init ≠ e_curr do
    while e_curr is not a frontier-edge do
        e_curr ← CWvertexEdge(e)
    end while
    e_curr ← next(e_curr)
    P ← P ∪ origin(e_curr)
end while
return P
```

Polylla 2D algorithm

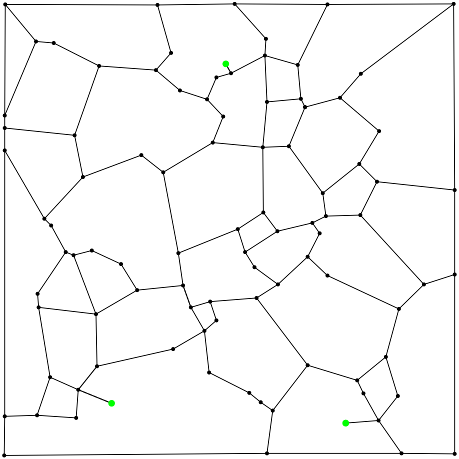
- Input: Triangulation $T(\Omega)$.
- Output: Polygon mesh



Algorithm has three main phases

1. Label Phase
2. Traversal Phase
3. Polygon reparation Phase

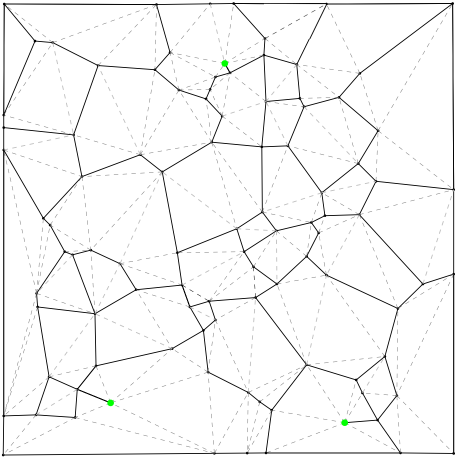
Repair Phase



Algorithm Non-simple polygon reparation

Require: Non-simple polygon P
Ensure: Set of simple polygons S
  `subseed list` **as** L_p **and** `usage bitarray` **as** A
  $S \leftarrow \emptyset$
  for all barrier-edge tip b **in** P **do**
  end for

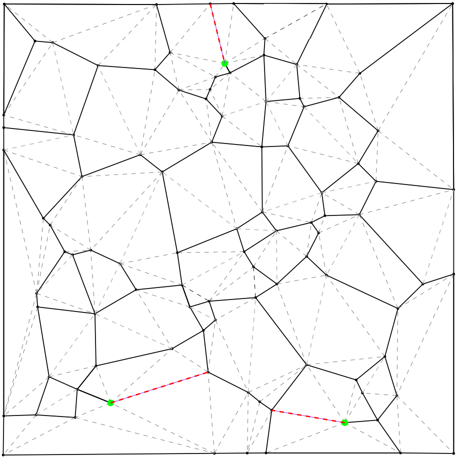
Repair Phase



Algorithm Non-simple polygon reparation

Require: Non-simple polygon P
Ensure: Set of simple polygons S
  `subseed list` **as** L_p **and** `usage bitarray` **as** A
  $S \leftarrow \emptyset$
  for all barrier-edge tip b **in** P **do**
  end for

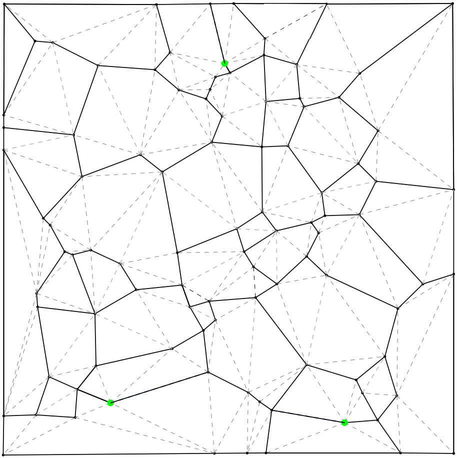
Repair Phase



Algorithm Non-simple polygon reparation

Require: Non-simple polygon P
Ensure: Set of simple polygons S

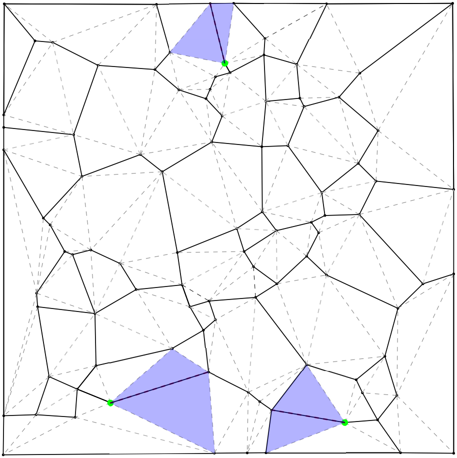
```
subseed list as L_p and usage bitarray as A
S ← ∅
for all barrier-edge tip b in P do
    e ← edgeOfVertex(b)
    while e is not a frontier-edge do
        e ← CWvertexEdge(e)
    end while
    for 0 to (textscdegree(b) - 1)/2 do
        e ← CWvertexEdge(e)
    end for
end for
```



Algorithm Non-simple polygon reparation

Require: Non-simple polygon P
Ensure: Set of simple polygons S

```
subseed list as L_p and usage bitarray as A
S ← ∅
for all barrier-edge tip b in P do
    e ← edgeOfVertex(b)
    while e is not a frontier-edge do
        e ← CWvertexEdge(e)
    end while
    for 0 to (textscdegree(b) - 1)/2 do
        e ← CWvertexEdge(e)
    end for
    Label e as frontier-edge
    Save half-edges h_1 and h_2 of e in L_p
    A[h_1] ← True, A[h_2] ← True
end for
```

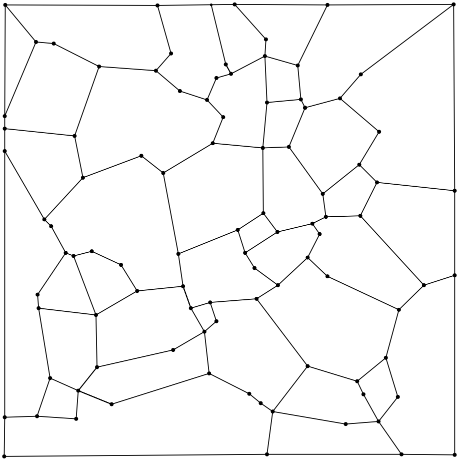


Algorithm Non-simple polygon reparation

Require: Non-simple polygon P
Ensure: Set of simple polygons S

```
subseed list as L_p and usage bitarray as A
S ← ∅
for all barrier-edge tip b in P do
    e ← edgeOfVertex(b)
    while e is not a frontier-edge do
        e ← CWvertexEdge(e)
    end while
    for 0 to (textscdegree(b) - 1)/2 do
        e ← CWvertexEdge(e)
    end for
    Label e as frontier-edge
    Save half-edges h_1 and h_2 of e in L_p
    A[h_1] ← True, A[h_2] ← True
end for
for all half-edge h in L_p do
    if A[h] is True then
        A[h] ← False
        Generate new polygon P′ starting from h repeating the Traversal phase.
        Set as False all indices of half-edges in A used to generate P′
    end if
    S ← S ∪ P′
end for
return S
```

Repair Phase



Algorithm Non-simple polygon reparation

Require: Non-simple polygon P
Ensure: Set of simple polygons S

subseed list **as** L_p **and** usage bitarray **as** A
$S \leftarrow \emptyset$
for all barrier-edge tip b in P **do**
    $e \leftarrow$ edgeOfVertex(b)
    while e is not a frontier-edge **do**
        $e \leftarrow$ CWvertexEdge(e)
    end while
    for 0 **to** (textscdegree(b) - 1)/2 **do**
        $e \leftarrow$ CWvertexEdge(e)
    end for
    Label e as frontier-edge
    Save half-edges h_1 and h_2 of e in L_p
    $A[h_1] \leftarrow$ True, $A[h_2] \leftarrow$ True
end for
for all half-edge h in L_p **do**
    if $A[h]$ is True **then**
        $A[h] \leftarrow$ False
        Generate new polygon P' starting from h repeating the Traversal phase.
        Set as False all indices of half-edges in A used to generate P'
    end if
    $S \leftarrow S \cup P'$
end for
return S