

Geometry Processing in Python - on the Example of Polygon Aggregation -

PhD colloquium DGK - September 16th, 2025

Alexander Naumann

- ▶ Dealing with geometries in code is paramount for various geoinformation tasks
- ▶ It is important to consider the use of data structures to handle geometric computations efficiently
- ▶ This tutorial is based on Python as a programming language. The general concepts are however directly transferrable to other programming languages
- ▶ Python itself is not a particularly efficient language, but we make use of libraries that are based on C++ implementations (`shapely`, `triangle`)

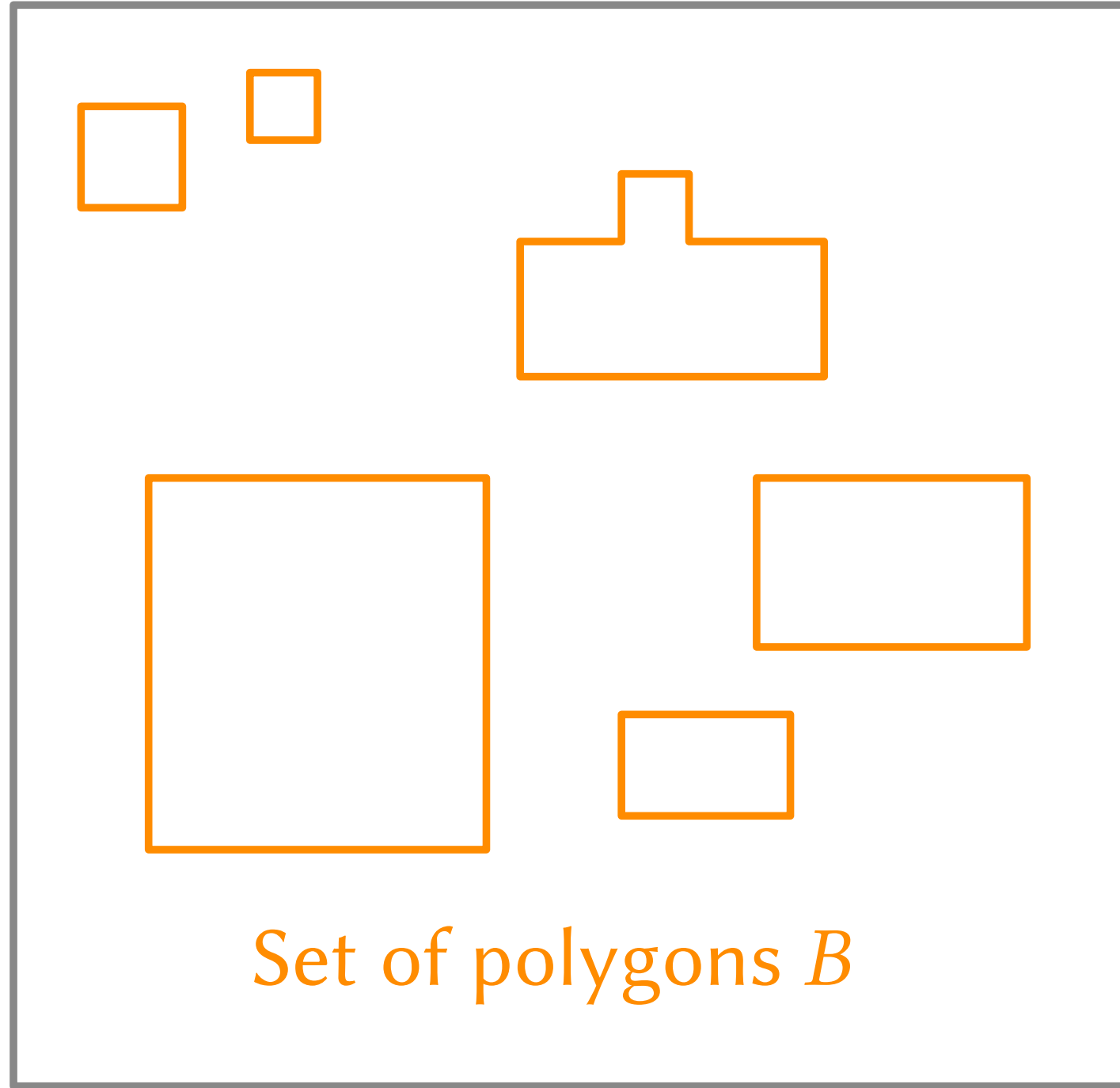
What to expect from this Tutorial

3

- ▶ an example of how to deal with geometric computations in code
- ▶ the use of efficient datastructures to speed up geometric computations
- ▶ we take polygon aggregation as an exemplary use case
- ▶ the techniques applied are directly transferable to other geometric problems

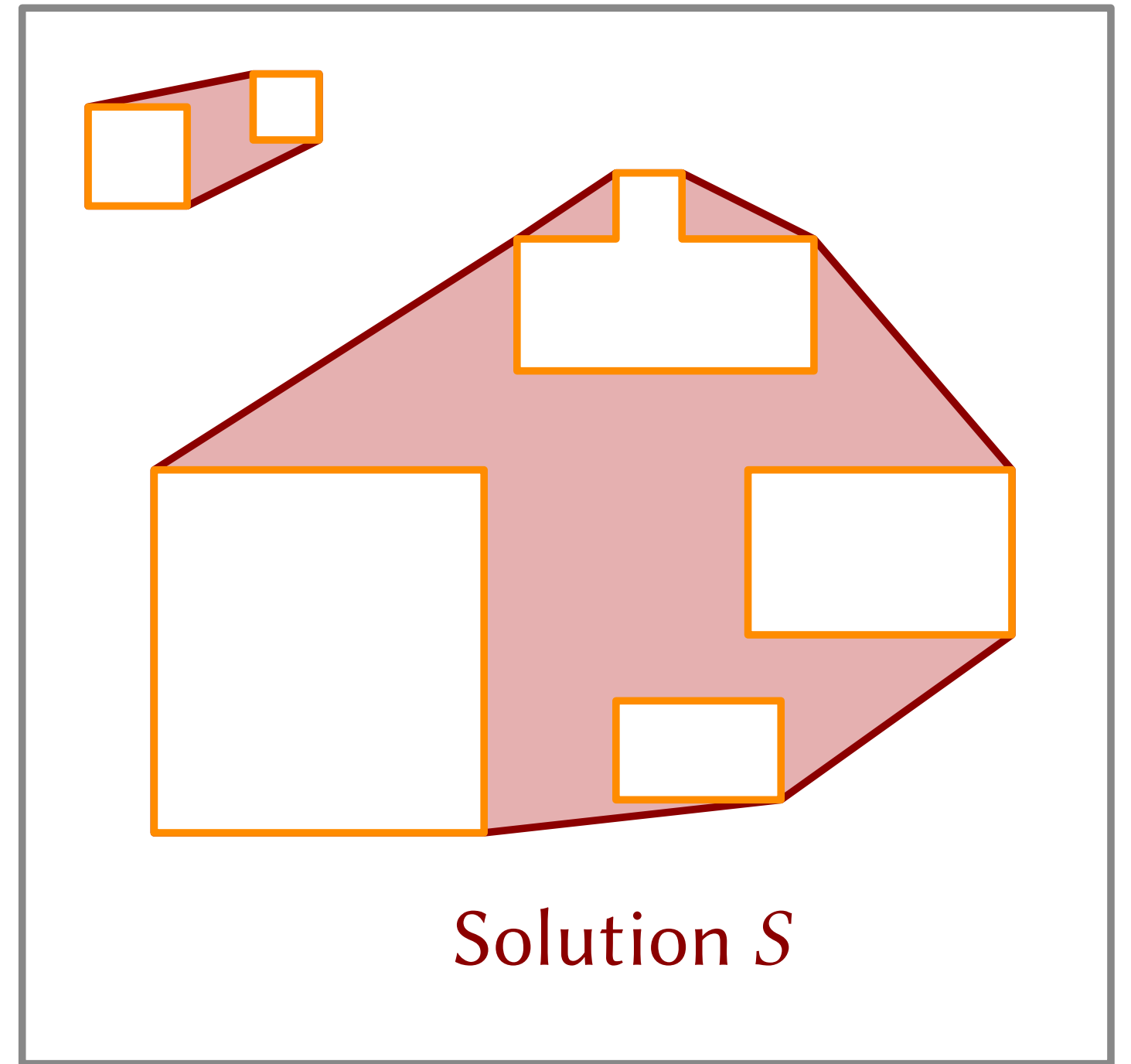
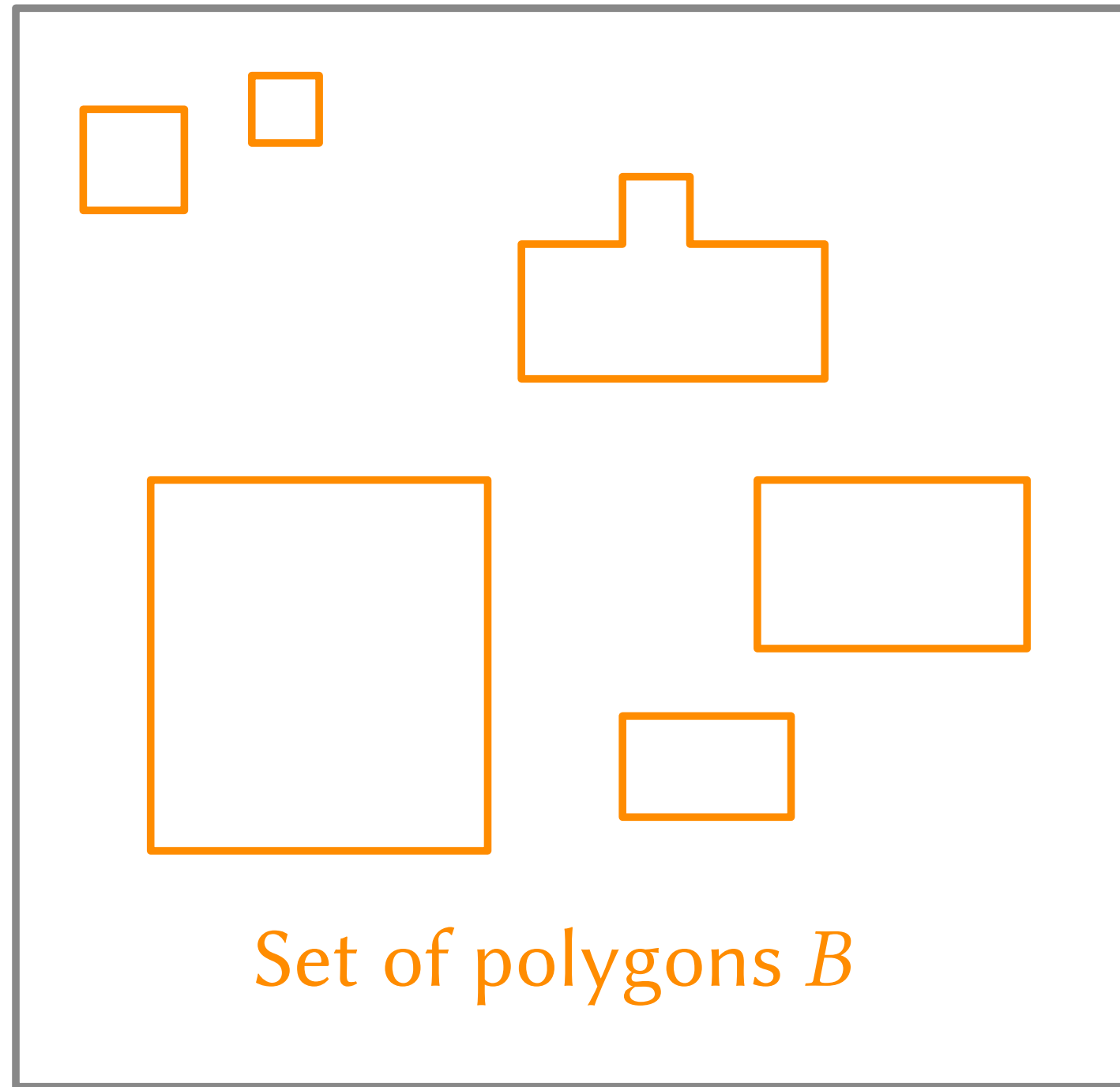
Our use case: Polygon aggregation

4 - 1



Our use case: Polygon aggregation

4 - 2



Application of Polygon Aggregation

5 - 1

- ▶ One exemplary application of polygon aggregation are building footprints in maps.
- ▶ On small scales, building footprints should be aggregated to make the map readable:

Application of Polygon Aggregation

5 - 2

- ▶ One exemplary application of polygon aggregation are building footprints in maps.
- ▶ On small scales, building footprints should be aggregated to make the map readable:



Application of Polygon Aggregation

5 - 3

- ▶ One exemplary application of polygon aggregation are building footprints in maps.
- ▶ On small scales, building footprints should be aggregated to make the map readable:



Source: basemap.de

How to aggregate?

- ▶ multiple strategies exist
- ▶ today we will apply the algorithm proposed by Rottmann et al.¹ - "Bicriteria Shapes"
- ▶ given a fixed subdivision of the plane, where the input polygons B are faces, find a selection of faces including the input polygons that minimizes:

$$S_{\text{OPT}} = \operatorname{argmin}_{S \in \mathcal{S}} A(S) + \alpha \cdot P(S)$$

\mathcal{S} = set of all possible solutions

$A(S)$ = area of S

$P(S)$ = perimeter of S

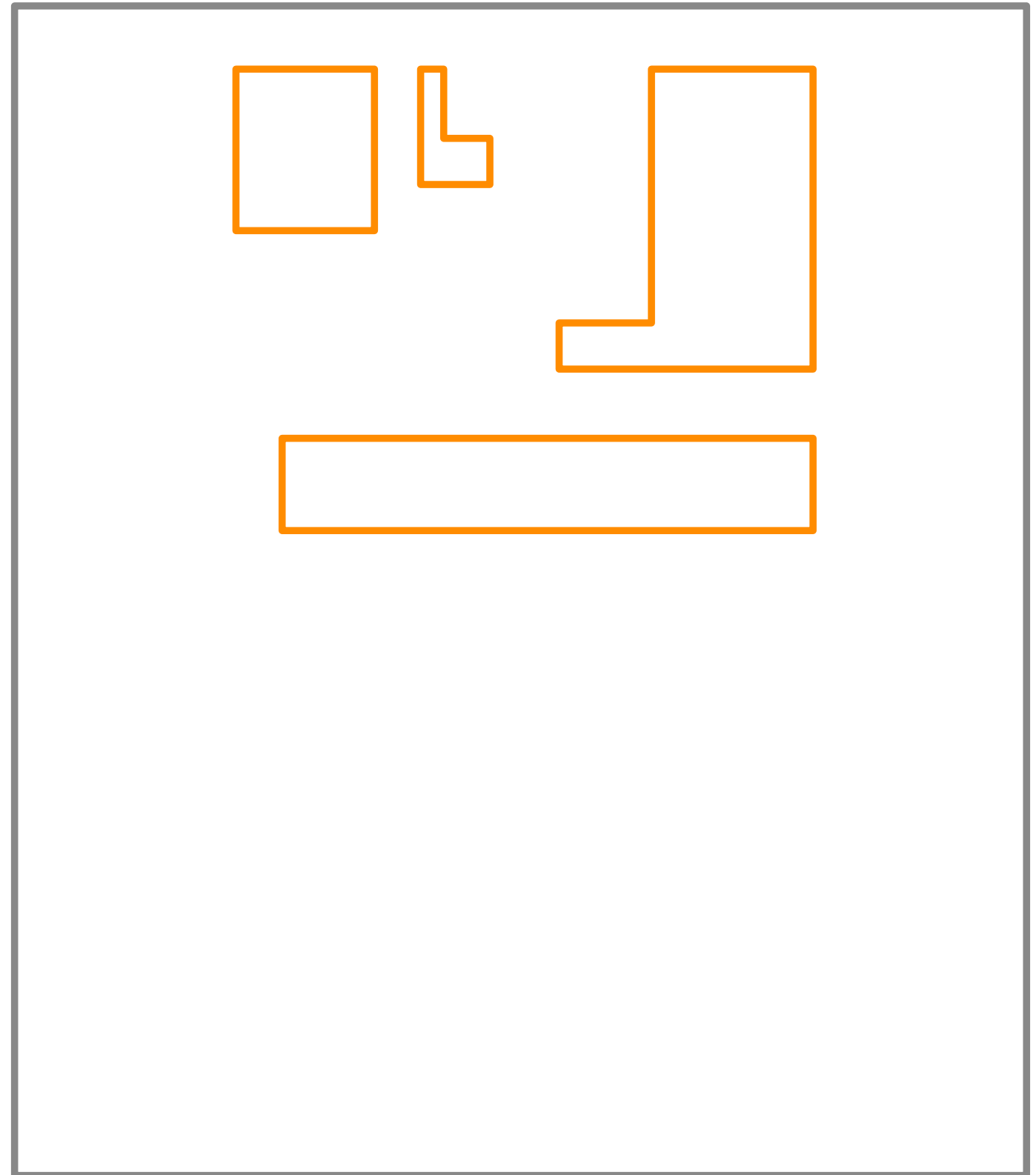
¹ Peter Rottmann, Anne Driemel, Herman Haverkort, Heiko Röglin, and Jan-Henrik Haunert. 2025. Bicriteria Shapes: Hierarchical Grouping and Aggregation of Polygons with an Efficient Graph-Cut Approach. ACM Trans. Spatial Algorithms Syst. 11, 1, Article 3 (March 2025), 23 pages. <https://doi.org/10.1145/3705001>

Getting a feel for the Algorithm

7 - 1

Getting a feel for the Algorithm

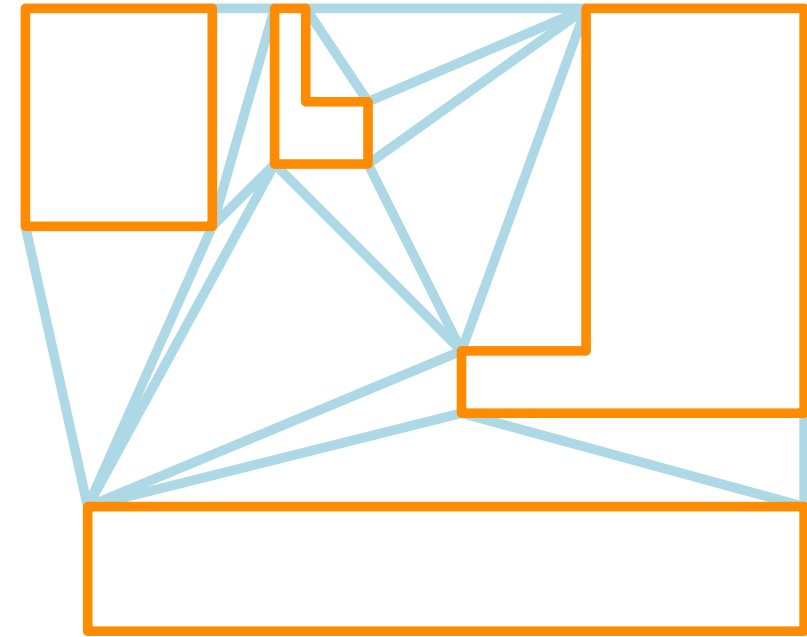
7 - 2



Getting a feel for the Algorithm

7 - 3

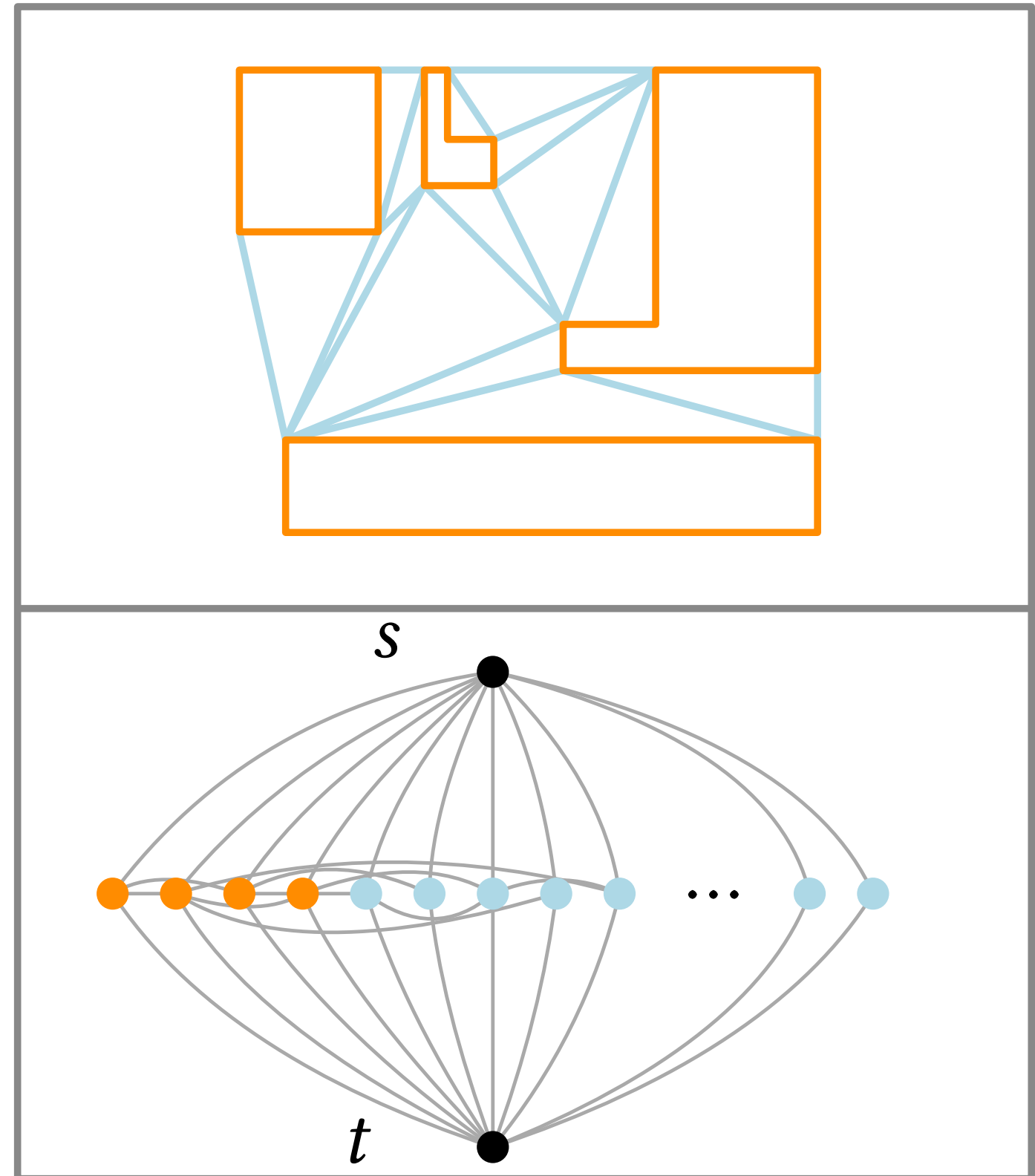
- ▶ compute constrained Delaunay triangulation based on input polygons



Getting a feel for the Algorithm

7 - 4

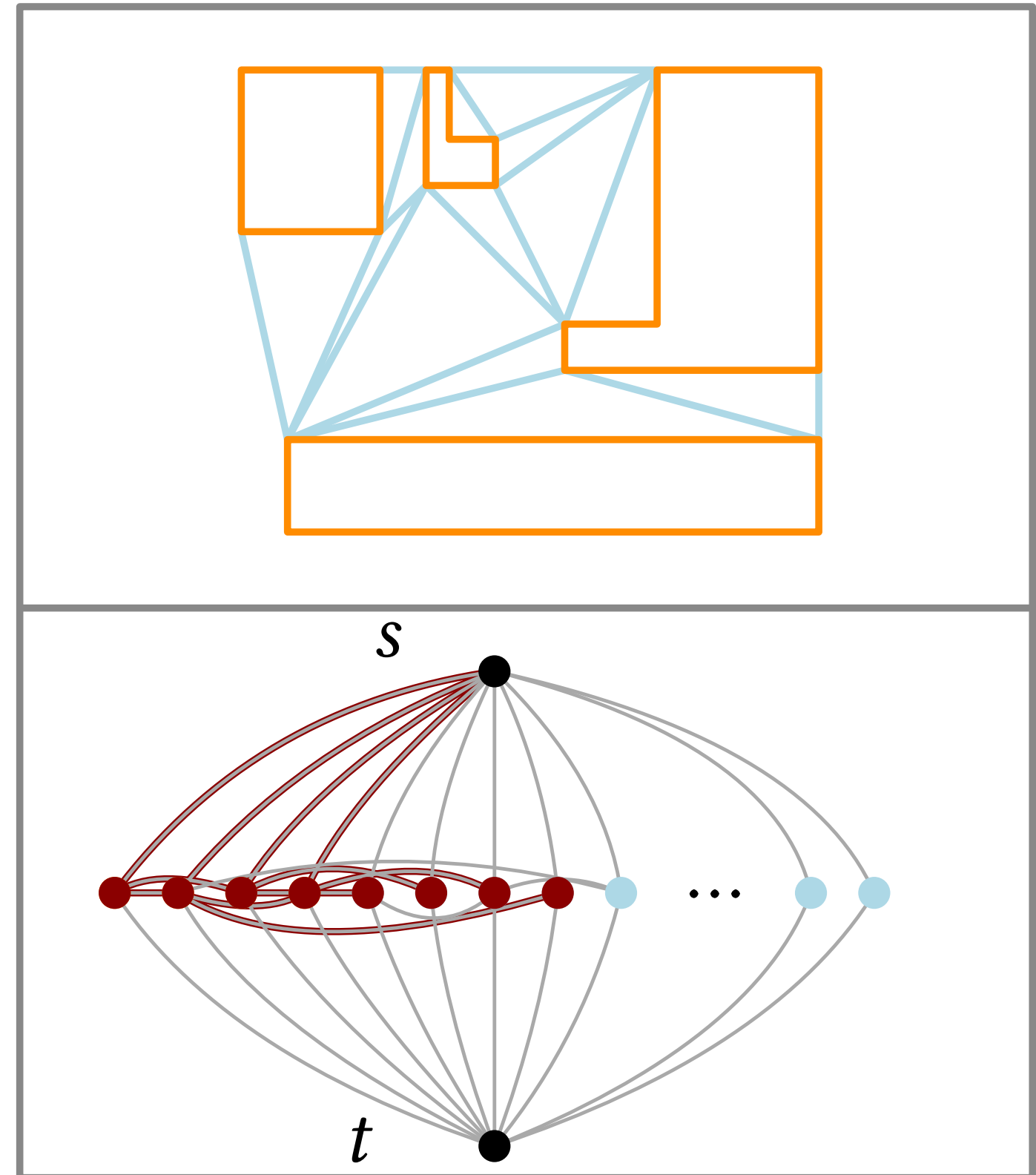
- ▶ compute constrained Delaunay triangulation based on input polygons
- ▶ create graph structure



Getting a feel for the Algorithm

7 - 5

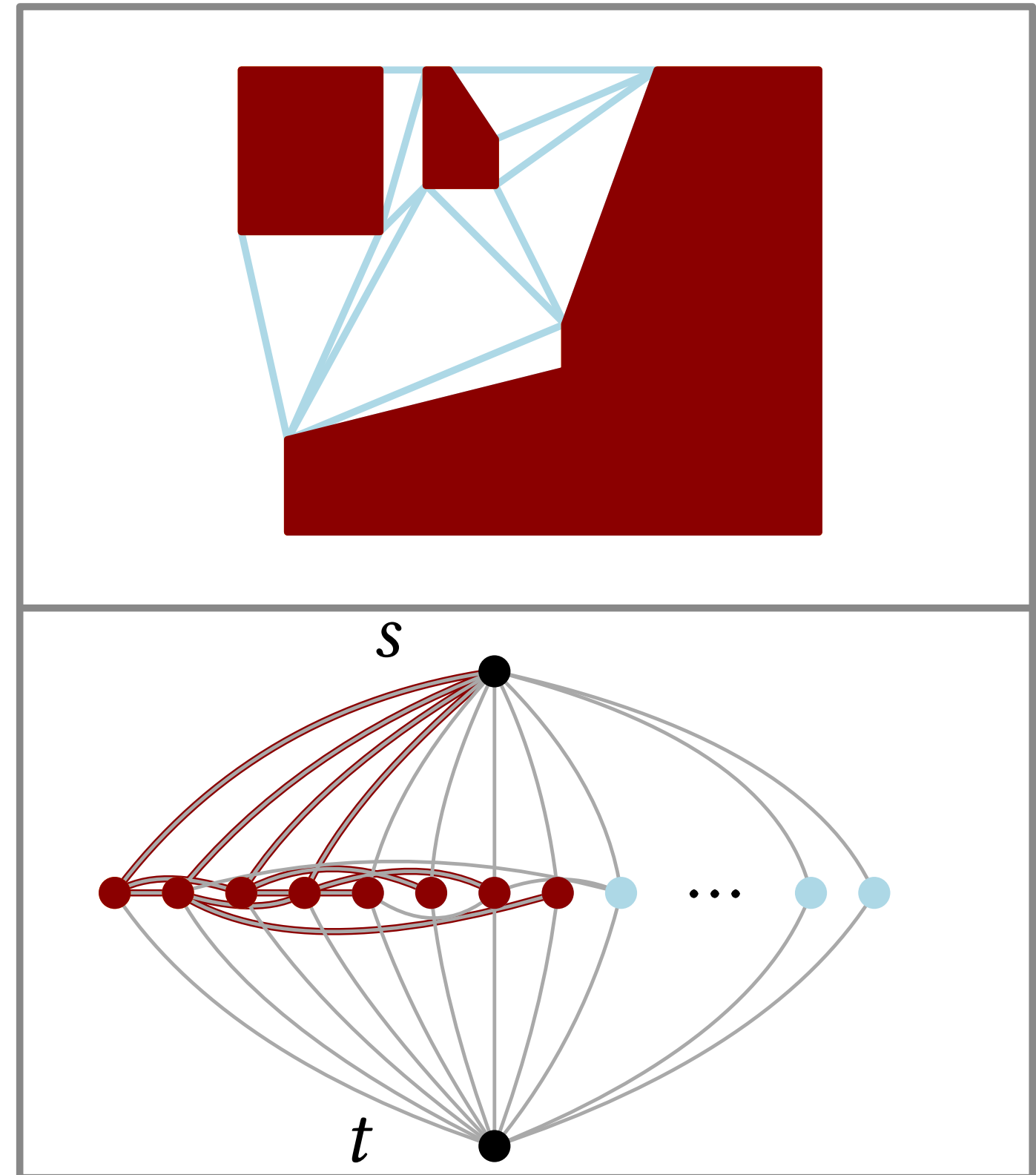
- ▶ compute constrained Delaunay triangulation based on input polygons
- ▶ create graph structure
- ▶ compute minimum s - t -cut and retrieve corresponding geometric solution



Getting a feel for the Algorithm

7 - 6

- ▶ compute constrained Delaunay triangulation based on input polygons
- ▶ create graph structure
- ▶ compute minimum s - t -cut and retrieve corresponding geometric solution



Steps to take a closer look on

8

- ▶ the library we use to compute the triangulation returns the triangles only
- ▶ additionally, it also triangulates the interiors of the input polygon
- ▶ for building the graph, we need to know which triangles actually lie within input polygons



Flagging Triangles

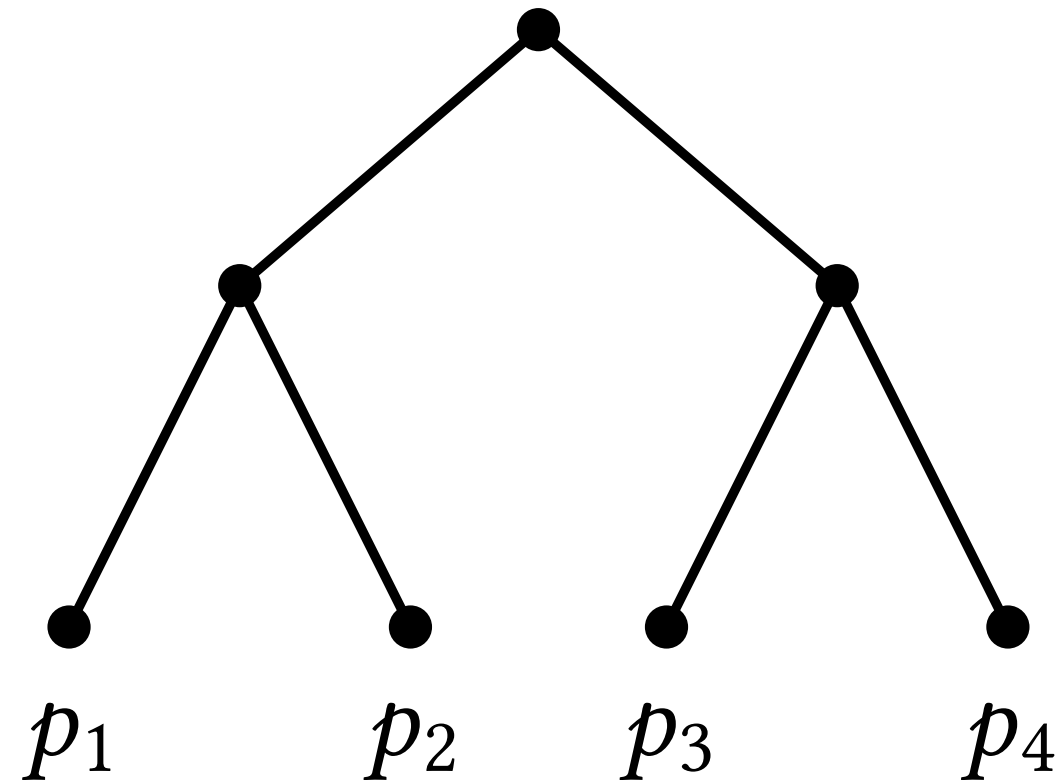
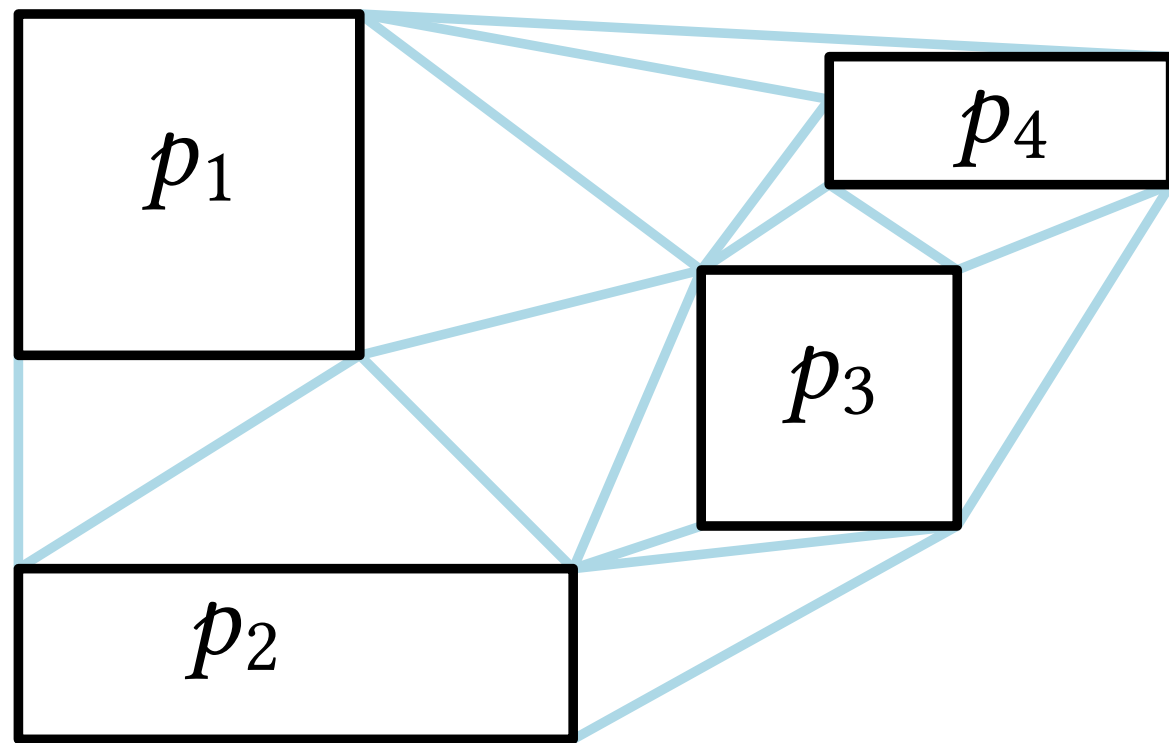
- ▶ naive method (pre-implemented): simply check for every triangle if it is inside a polygon via pairwise comparisons

```
def flag_triangles_naive(triangles, polygons):  
    flagged = []  
    for tri_poly in triangles:  
        if any(tri_poly.intersection(poly).area > 1e-4  
              for poly in polygons):  
            flagged.append(True)  
        else:  
            flagged.append(False)  
    return flagged
```

Flagging Triangles - Improved

10 - 1

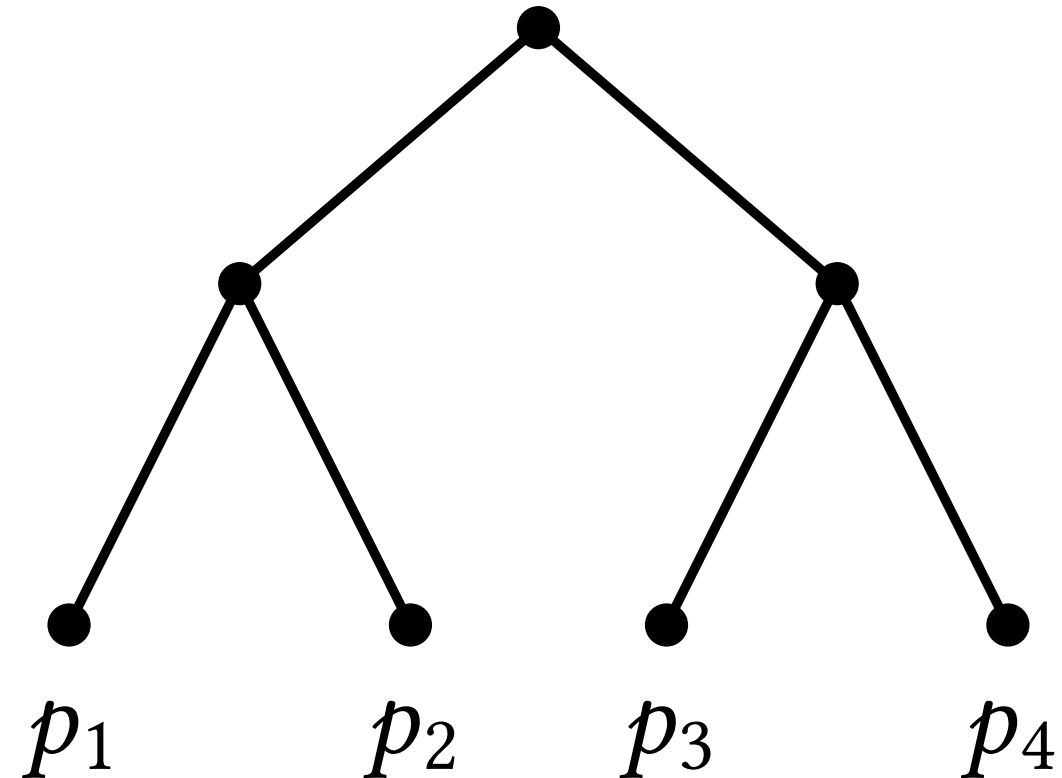
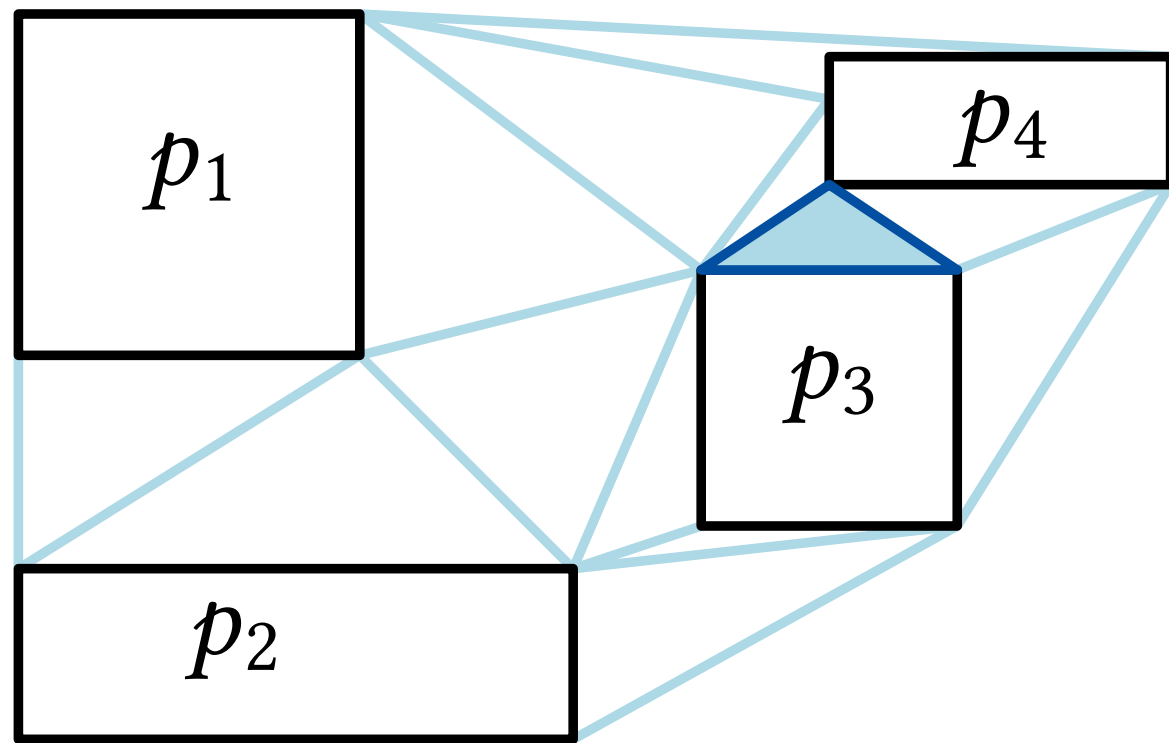
- ▶ we will now use an spatial index datastructure (in this case an R-Tree) to reduce the number of polygons that every triangle is compared to substantially



Flagging Triangles - Improved

10 - 2

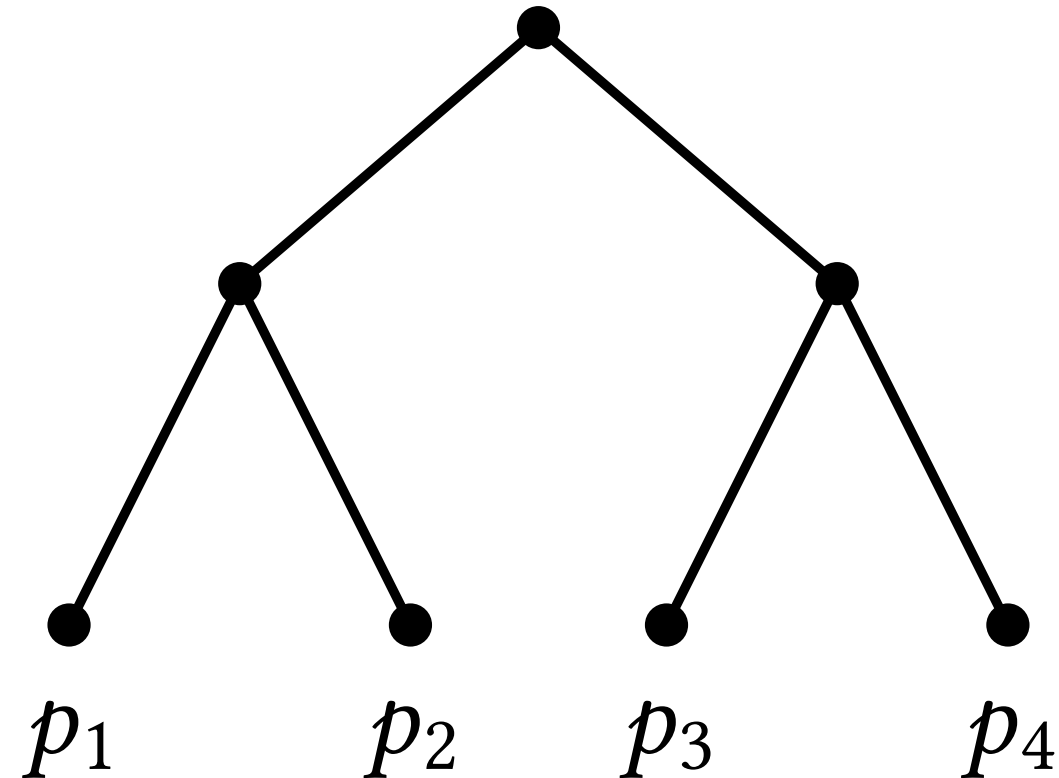
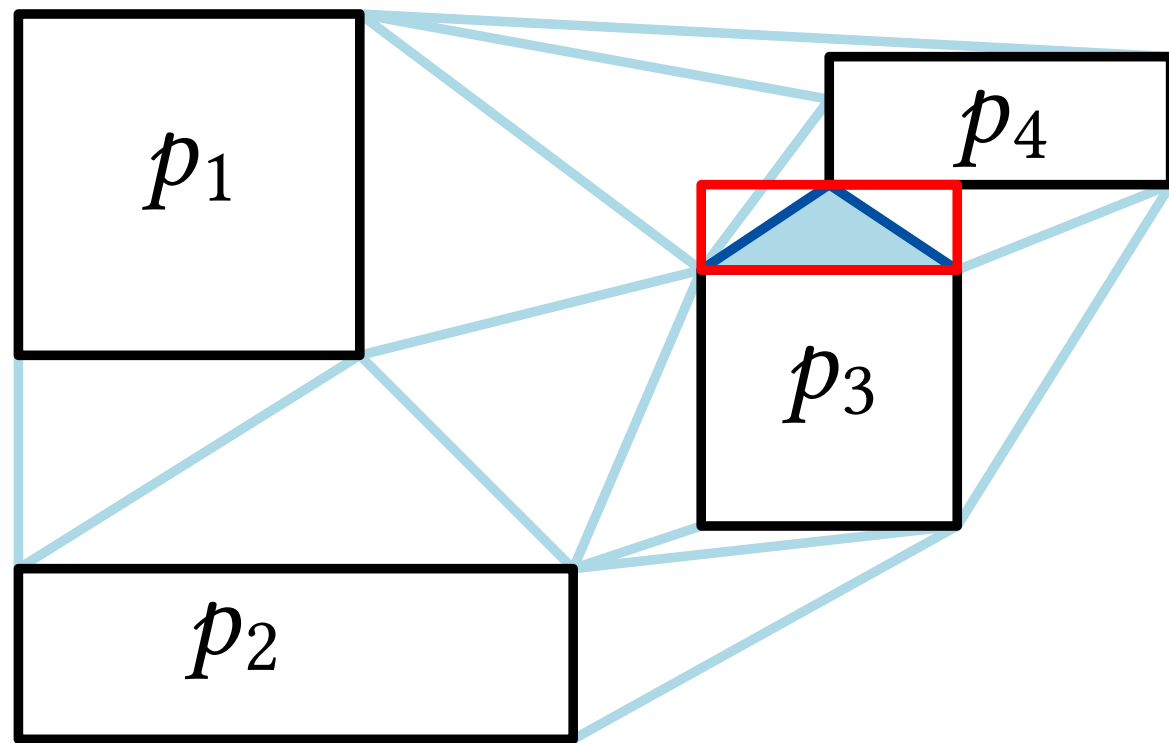
- ▶ we will now use an spatial index datastructure (in this case an R-Tree) to reduce the number of polygons that every triangle is compared to substantially



Flagging Triangles - Improved

10 - 3

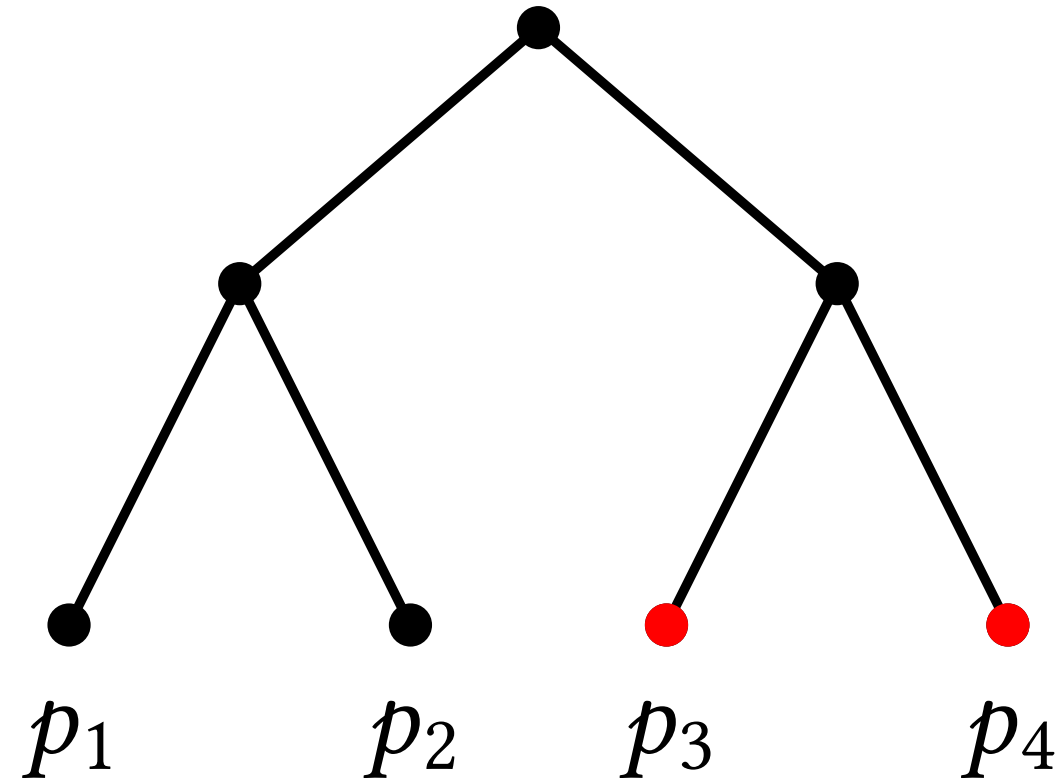
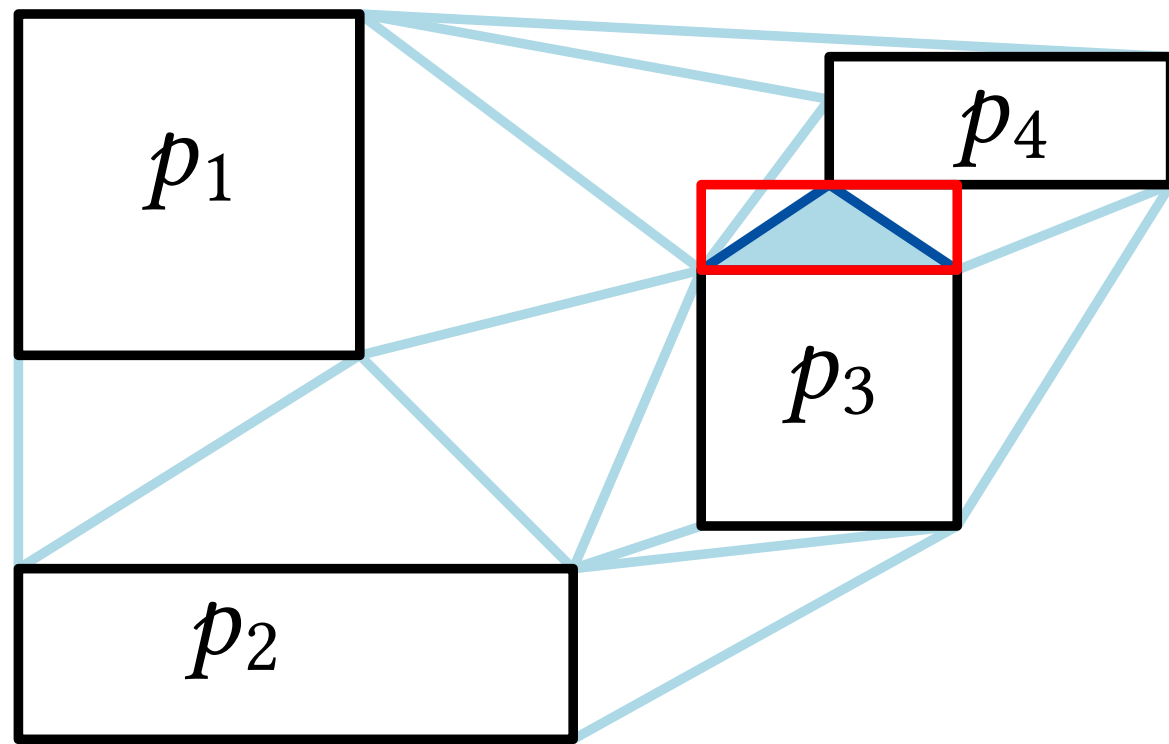
- ▶ we will now use an spatial index datastructure (in this case an R-Tree) to reduce the number of polygons that every triangle is compared to substantially



Flagging Triangles - Improved

10 - 4

- ▶ we will now use an spatial index datastructure (in this case an R-Tree) to reduce the number of polygons that every triangle is compared to substantially



Building the graph

11

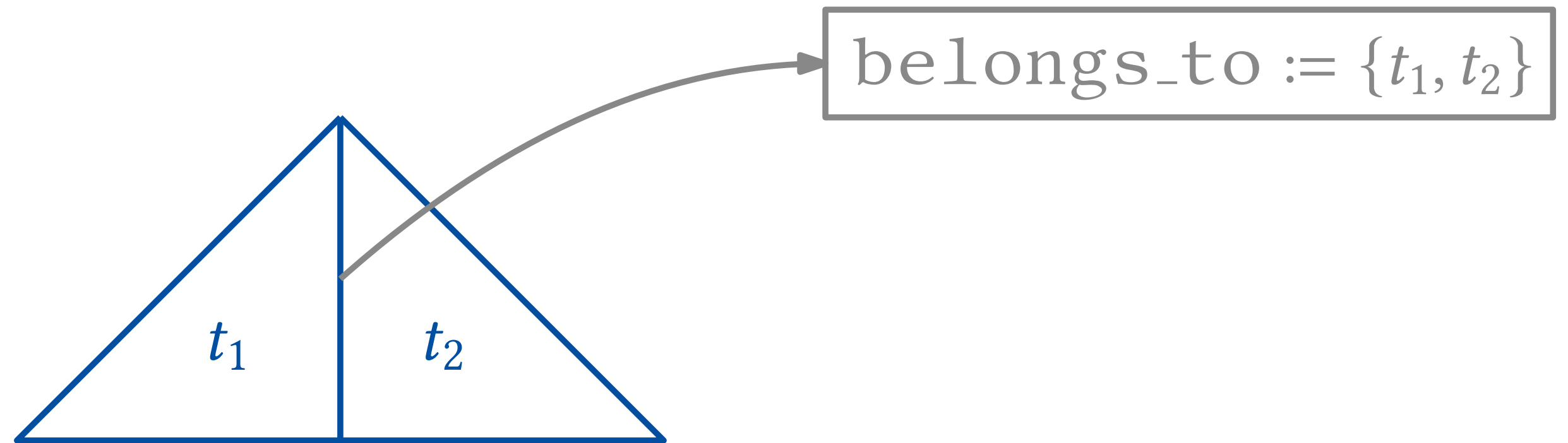
- ▶ to build the graph, we are given a set of triangles T
- ▶ we do not know, which triangles are neighbors
- ▶ in our pre-implemented baseline, we compare all pairwise triangles for an intersection:

```
for i, tri1 in enumerate(triangles):  
    for j, tri2 in enumerate(triangles):  
        if i >= j:  
            continue  
        inter = tri1.intersection(tri2)  
        ...
```

Building the Graph - Improved

12

- ▶ we can instead create a map so we can find out the triangle each edge belongs to in $O(1)$ time



- ▶ geometric computations themselves are computationally expensive and vulnerable to instabilities
- ▶ spatial index structures (such as R-Trees) are paramount to handle localization operations on large geometric datasets
- ▶ numeric instabilities should always be considered in code handling geometric computations
- ▶ if possible, we want to solve geometric problems outside of the spatial domain, e.g., via the use of (hash-) maps or graph data structures