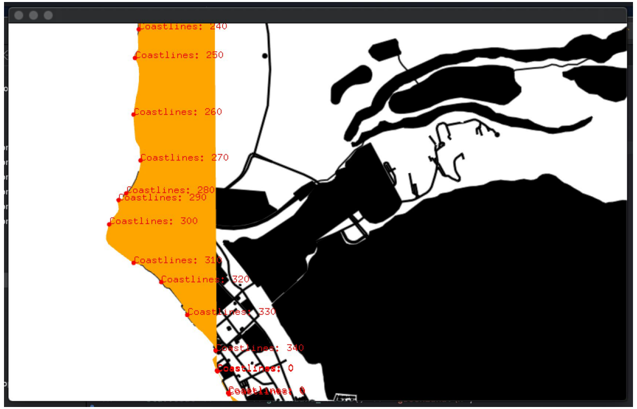
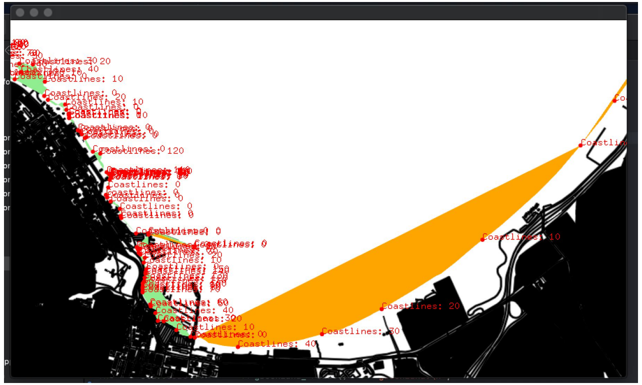
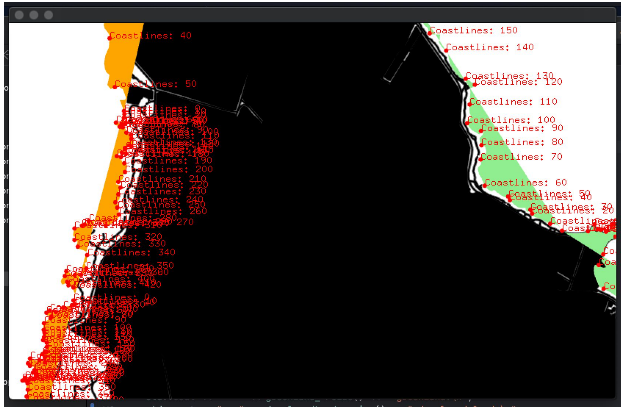


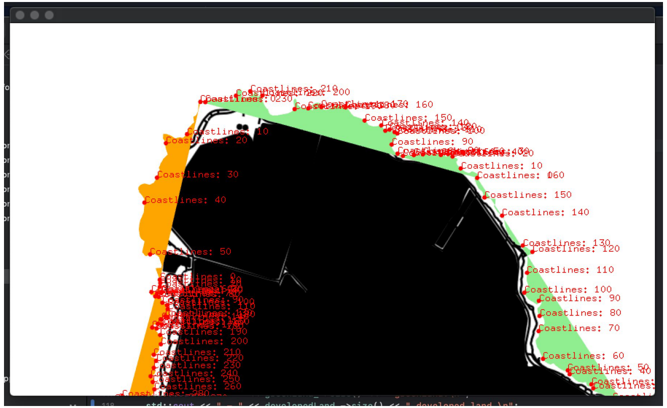
orange: [0] außen, [n-1] außen; [0].x < [n-1].x && [0].y < [n-1].y
grün: -



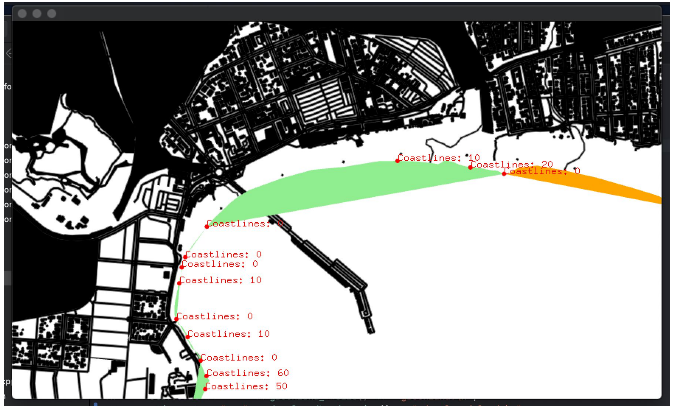
orange: [0] außen, [n-1] innen; [0].x > [n-1].x && [0].y < [n-1].y
grün: [0] innen, [n-1] außen; [0].x > [n-1].x && [0].y > [n-1].y



orange: [0] außen, [n-1] außen; [0].x > [n-1].x && [0].y < [n-1].y
grün: [0] außen, [n-1] außen; [0].x > [n-1].x && [0].y > [n-1].y



orange: [0] innen, [n-1] außen; [0].x > [n-1].x && [0].y < [n-1].y
grün: [0] außen, [n-1] innen; [0].x > [n-1].x && [0].y > [n-1].y

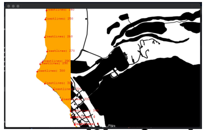


orange: [0] innen, [n-1] außen; [0].x < [n-1].x && [0].y < [n-1].y
grün: [0] außen, [n-1] innen; [0].x < [n-1].x && [0].y > [n-1].y

oben -> unten: Wasser ist links davon

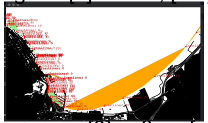
unten -> oben: Wasser ist rechts davon

(0,0) = links oben; (x_max, y_max) = rechts unten

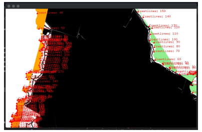


orange: [0] außen, [n-1] außen; [0].x < [n-1].x && [0].y < [n-1].y
grün: -

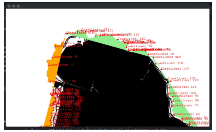
orange: -
grün: [0] außen, [n-1] außen; [0].x < [n-1].x && [0].y > [n-1].y



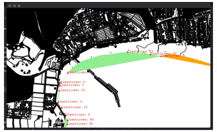
orange: [0] außen, [n-1] innen; [0].x > [n-1].x && [0].y < [n-1].y
grün: [0] innen, [n-1] außen; [0].x > [n-1].x && [0].y > [n-1].y



orange: [0] außen, [n-1] außen; [0].x > [n-1].x && [0].y < [n-1].y
grün: [0] außen, [n-1] außen; [0].x > [n-1].x && [0].y > [n-1].y



orange: [0] innen, [n-1] außen; [0].x > [n-1].x && [0].y < [n-1].y
grün: [0] außen, [n-1] innen; [0].x > [n-1].x && [0].y > [n-1].y



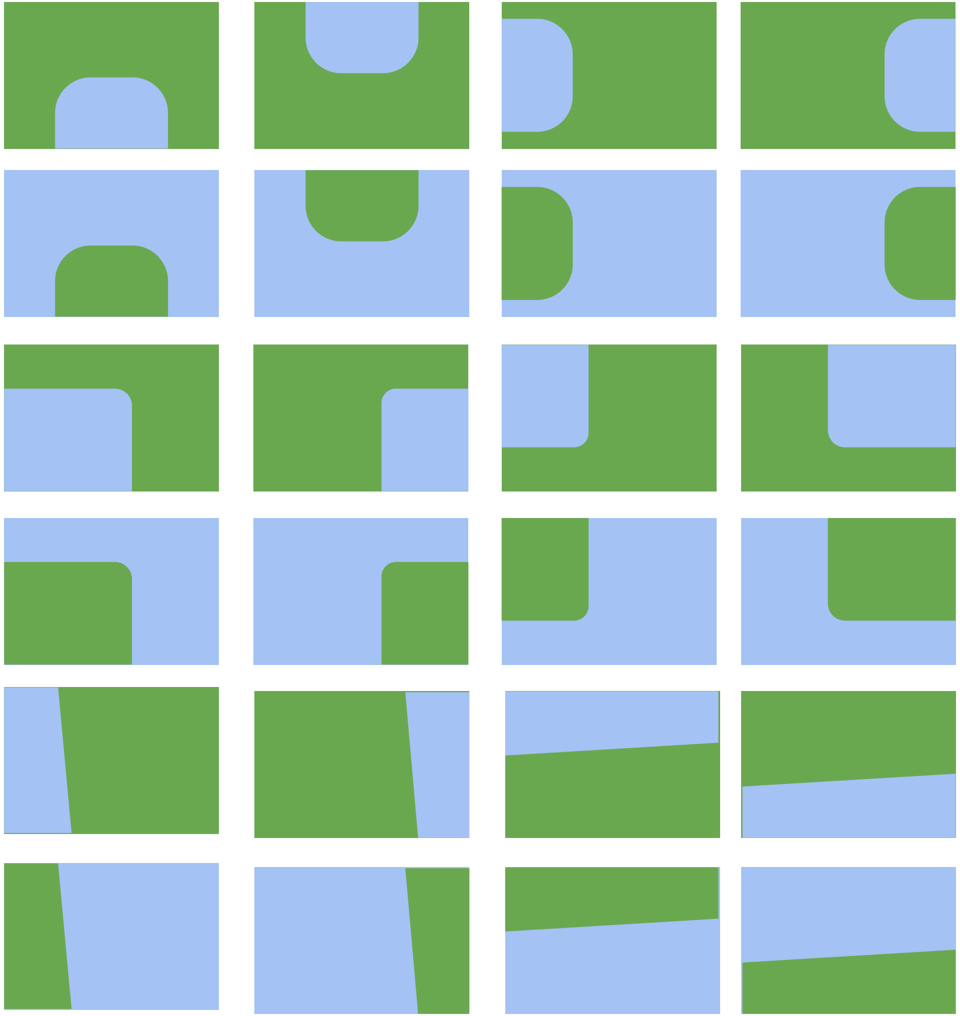
orange: [0] innen, [n-1] außen; [0].x < [n-1].x && [0].y < [n-1].y
grün: [0] außen, [n-1] innen; [0].x < [n-1].x && [0].y > [n-1].y

	[0] außen	**[0] innen**	**[n-1] außen**	**[n-1] innen**	**[0].x < [n-1].x**	**[0].x > [n-1].x**	**[0].y < [n-1].y**	**[0].y > [n-1].y**	**polygon corners clockwise from top-left**
	x		x		x		x		(0,0)-([0].xy)-([n-1].xy)-(0,max_y)
	x		x		x			x	([n-1].xy)-(max_x,0)-(max_x,max_y)-([0].xy)
	x			x		x	x		([n-1].x,0)-(max_x,0)-([0].xy)-([n-1.xy])
		x	x			x		x	(0,0)-([0].x,0)-([0].xy)-([n-1].xy)
	x		x			x	x		(0,0)-([0].xy)-([n-1].xy)-(0,max_y)
	x		x			x		x	([n-1].xy)-(max_x,0)-(max_x,max_y)-([0].xy)
		x	x			x	x		(0,0)-([0].x,0)-([0].xy)-([n-1].xy)-(0,max_y) 5 PUNKTE !!!
	x			x		x		x	([n-1].x,0)-(max_x,0)-(max_x,max_y)-([0].x,max_y)-([n-1].xy) 5 PUNKTE !!!!
		x	x		x		x		([0].xy)-([n-1].xy)-(max_x,max_y)-([0].x,max_y)
	x			x	x			x	from bottom left: ([0].xy)-([n-1].xy)-([n-1].x,max_y)

oben -> unten: Wasser ist links davon

unten -> oben: Wasser ist rechts davon

(0,0) = links oben; (x_max, y_max) = rechts unten



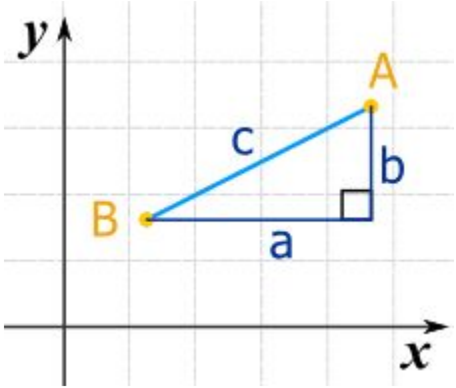
Land is always to the left of coastline ways, water is to the right. Coastline ways can never just stop in a dead-end, they always have to be connected to another coastline

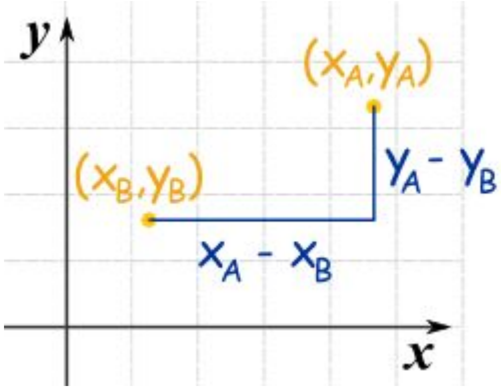
Connecting the Lines.

- Connect lines of which the startpoint and endpoint are not both outside of the bbox (so called “through” lines).
  - Find a line where the startpoint is not equal to any endpoint. This would be the start of the connected line.
- Connect lines where the endpoint is the startpoint of another line.
- If there are still unconnected lines which are not a loop (island) or connected with another line, or is not a “through” line, then:
  - Take the startpoint of this unconnected line and look for a close endpoint in all the other lines.
  - Take the endpoint of this unconnected line and look for a close startpoint in all the other lines.

Determining the Distance between Points.

Pythagoras doesn't work on a round earth but that is ok here, we only need a rough measure anyway.





c = sqrt((x_a - x_b)^2 + (y_a - y_b)^2)

Determining the Distance between Points (in km).

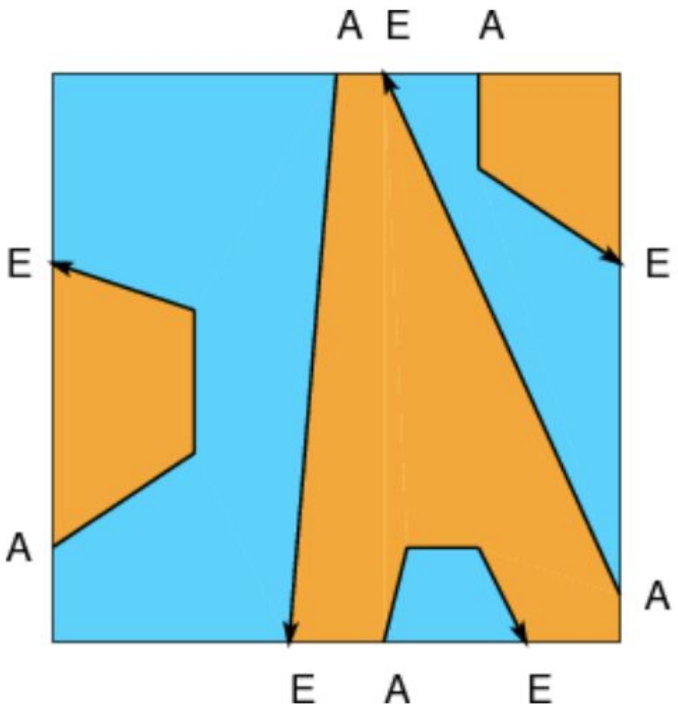
```
#define d2r (M_PI / 180.0)
//calculate haversine distance for linear distance
double haversine_km(double lat1, double long1, double lat2, double long2)
{
    double dlong = (long2 - long1) * d2r;
    double dlat = (lat2 - lat1) * d2r;
    double a = pow(sin(dlat/2.0), 2) + cos(lat1*d2r) * cos(lat2*d2r) * pow(sin(dlong/2.0), 2);
    double c = 2 * atan2(sqrt(a), sqrt(1-a));
    double d = 6367 * c;
    return d;
}
```

You can replace (M_PI / 180.0) with 0.0174532925199433 for better performance.

General information about geo and lines: https://www.movable-type.co.uk/scripts/latlong.html

Finding the Algorithm...

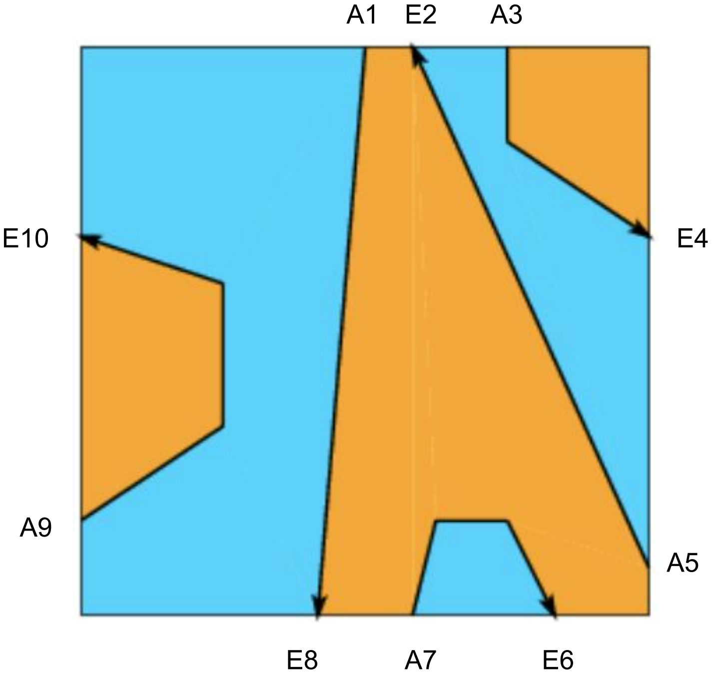
Some inspiration (maybe)




```
sl = []
Für alle offenen Küstenlinien k:
        Berechne Schnittpunkt s(A), s(E) mit der BoundingBox
        sl += (s(A), k, A)
        sl += (s(E), k, E)
m := Mittelpunkt der BoundingBox
Sortiere sl nach Winkel der Strecke [m, s]
i0 := 0
s0, k0, ty0 = sl[i]
Falls ty0 == 'A':
        i += 1
        s0, k0, ty0 = sl[i]
Solange nicht alle Punkte aus sl verarbeitet:
        s1, k1, ty1 = sl[(i+1) mod n]
        Verbinde k0 mit k1 durch die Strecke [s0, s1]
        i := (i + 2) mod n
```

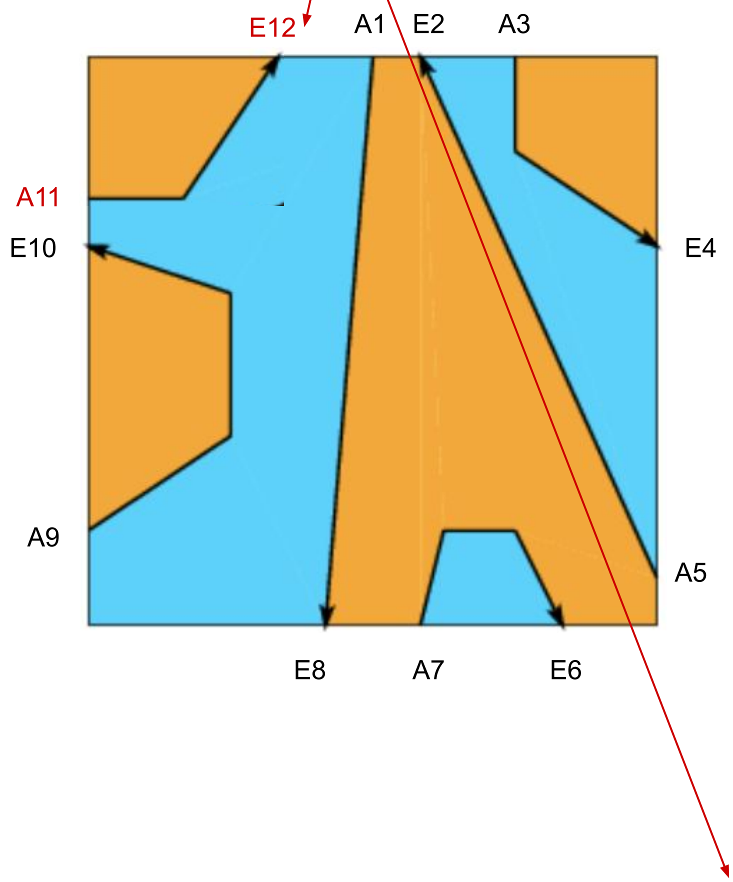
Source: https://monarch.qucosa.de/api/qucosa%3A19708/attachment/ATT-0/

Use a stack and queue, and go through the points clockwise.



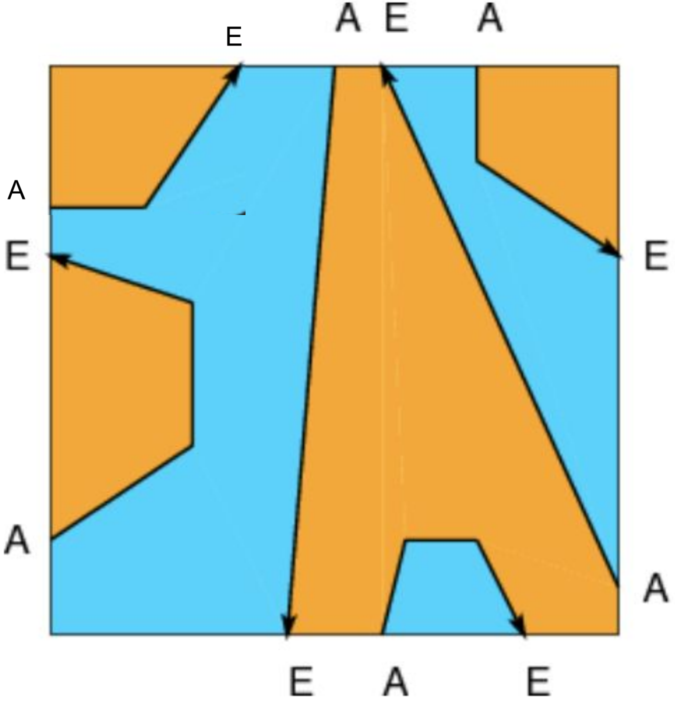
Point	Stack after condition/operation	condition/operation
A1	P1(A1)	A1 not A of tail in top -> push new P1 -> add to top(=P1)
E2	P1(A1,E2)	E2 not E of head in top(=P1) -> add to top
A3	P2(A3),P1(A1,E2)	A3 not A of tail in top(=P1) -> push new P2
E4	P1(A1,E2)	E4 is E of head of top(=P2) -> add to top(=P2) -> close top(=P2) -> pop top(=P2) -> P2=(A3,E4)
A5	P1(A1,E2,A5)	A5 is A of tail of top(=P1) -> add to top(=P1)
E6	P1(A1,E2,A5,E6)	E6 is not E of head in top(=P1) -> add to top(=P1)
A7	P1(A1,E2,A5,E6,A7)	A7 is A of tail of top(=P1) -> add to top(=P1)
E8		E8 is E of head of top(=P1) -> add to top(=P1) -> close top(=P1) -> pop top(=P1) -> P1=(A1,E2,A5,E6,A7,E8)
A9	P3(A9)	A9 is not A of tail in top -> push new P3 -> add to top
E10		E10 is E of head of top(=P3) -> add to top(=P3) -> close top(=P3) -> pop top(=P3) -> P3=(A9,E10)

2) Put in queue clockwise beginning at the top border
If first element is E, then remove from front and add to back



Point	Stack after condition/operation	condition/operation
A1	P1(A1)	A1 not A of tail in top -> push new P1 -> add to top(=P1)
E2	P1(A1,E2)	E2 not E of head in top(=P1) -> add to top(=P1)
A3	P2(A3),P1(A1,E2)	A3 not A of tail in top(=P1) -> push new P2
E4	P1(A1,E2)	E4 is E of head of top(=P2) -> add to top(=P2) -> close top(=P2) -> pop top(=P2) -> P2=(A3,E4)
A5	P1(A1,E2,A5)	A5 is A of tail of top(=P1) -> add to top(=P1)
E6	P1(A1,E2,A5,E6)	E6 is not E of head in top(=P1) -> add to top(=P1)
A7	P1(A1,E2,A5,E6,A7)	A7 is A of tail of top(=P1) -> add to top(=P1)
E8		E8 is E of head of top(=P1) -> add to top(=P1) -> close top(=P1) -> pop top(=P1) -> P1=(A1,E2,A5,E6,A7,E8)
A9	P3(A9)	A9 is not A of tail in top -> push new P3 -> add to top
E10		E10 is E of head of top(=P3) -> add to top(=P3) -> close top(=P3) -> pop top(=P3) -> P3=(A9,E10)
A11	P4(A11)	A11 is not A of tail in top -> push new P4 -> add to top
E12		E12 is E of head of top(=P4) -> close -> pop -> P4=(A11,E12)

The Algorithm.



Observations:

- Between E and A there is only water if they don’t belong to the same coastline; If E and A are from different coastlines, then there is just water in between
- Consecutive A and E belong to the same landmass
- If you go clockwise only close path if E is end of the first A of the path

1) Calculate the border intersection points BIPs (DONE)

2) Put in queue q clockwise beginning at the top border

    If first element is E, then remove from front and add to back

queue q;
for all BIP on top: add BIP to q
for all BIP on right: add BIP to q
for all BIP on bottom: add BIP to q
for all BIP on left: add BIP to q
if q.front is E: remove q.front and add to q.back

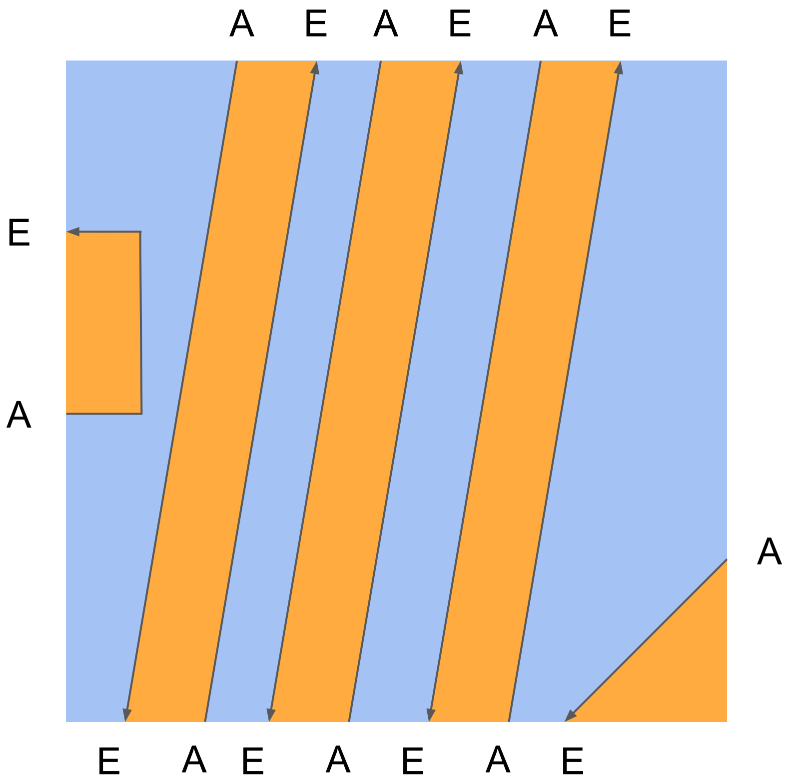
3) Then run the algorithm:

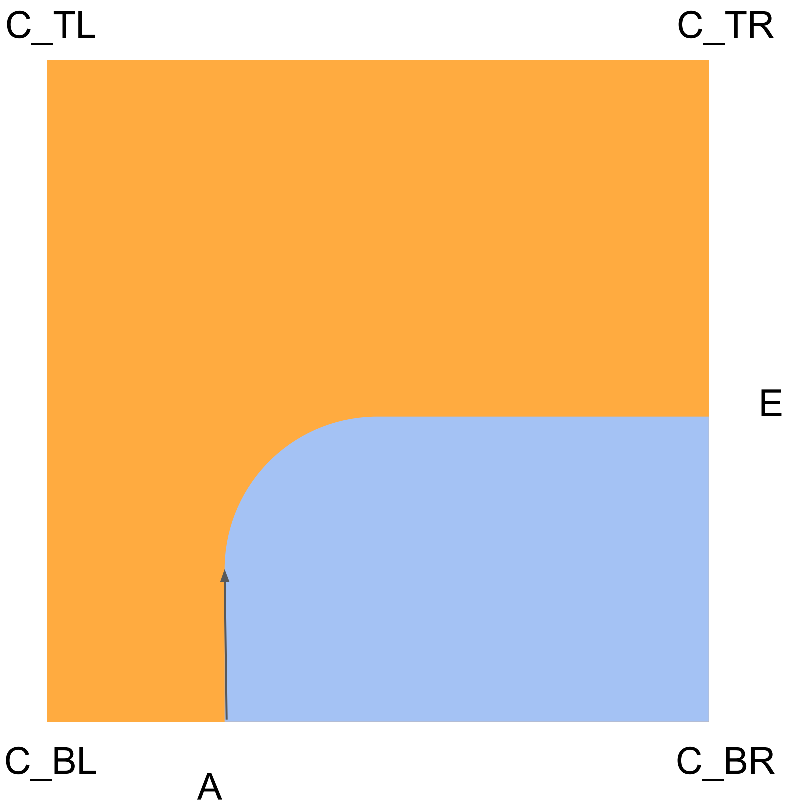
stack<path> s;
while (!q.isempty())
    BIP = q.front
    check if BIP is A of tail of s.top
        no: s.push(new path); s.top.append(BIP)
        yes: s.top.append(BIP)
    check if BIP is E of head of s.top
        no: s.top.append(BIP)
        yes: s.top.append(BIP), close s.top, pop s.top

path is just a vector of BIPs

close-function:

polygon p;
for i=0 to i=path.size
    if i==0: p.add(path[i])
    else if i>0 && path[i]==E:
        if path[i-1] and path[i] different borders, then p.add(corner) && p.add(path[i])
        else: p.add(path[i]);
    else if i>0 && path[i]==A: p.add(all points of way between path[i] and path[i-1]
    else if i==path.size: p.add(all points of way between path[0] and path[i]


A E A E A E
E
A
A
E A E A E A E



Check this only between A and E, because only between A and E (clockwise) there is landmass. Between E and A (clockwise) is water.
And only if A and E are on different borders (no matter which borders).

A C_BL C_TL C_TR E

Add the corner points as long as the E is not between the current and the next corner point.