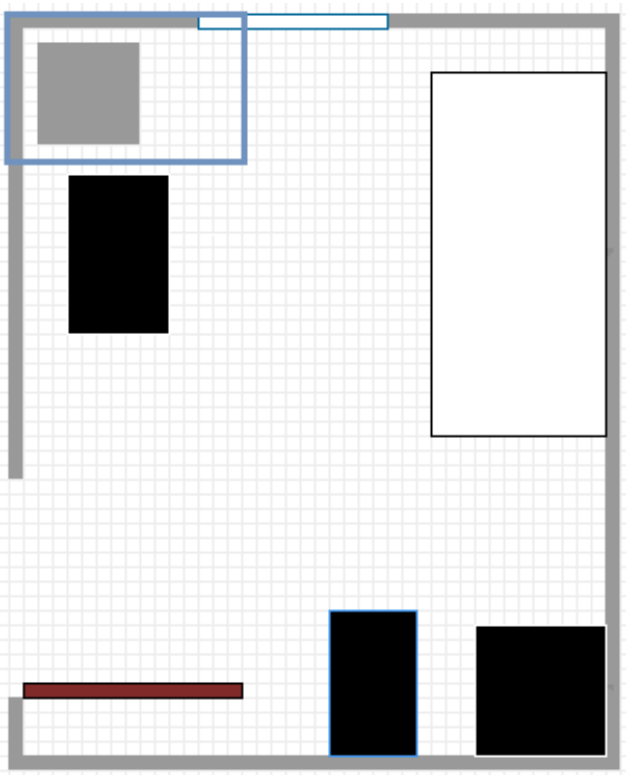
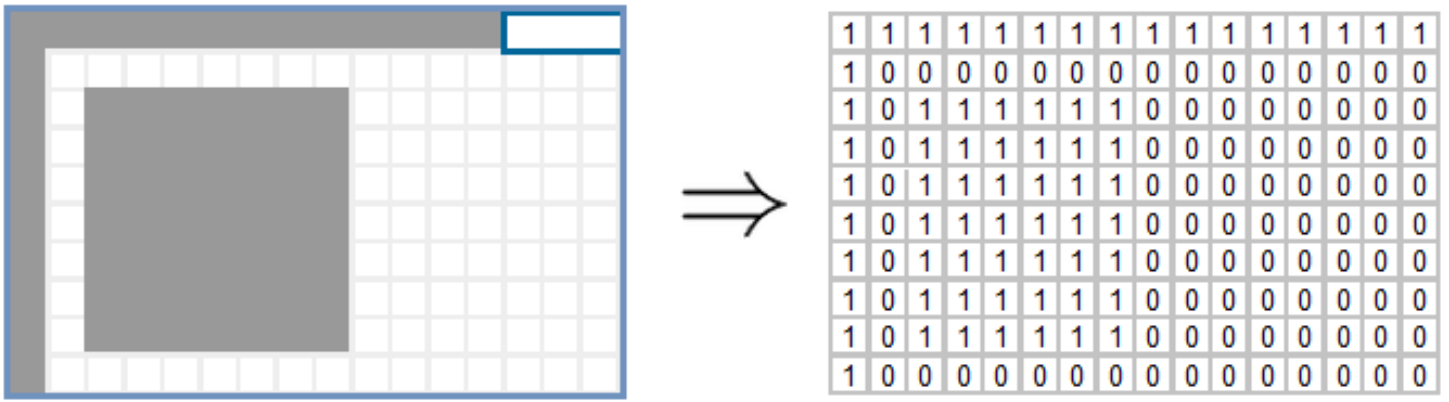
Geomatrix

Motivation

For some applications, we might have large boolean matrices where the true and false values are agrupated in rectangular chuncks.

For example, imagine you are trying to model the space of a room, and you want to use a matrix to represent the positions that are occupied.

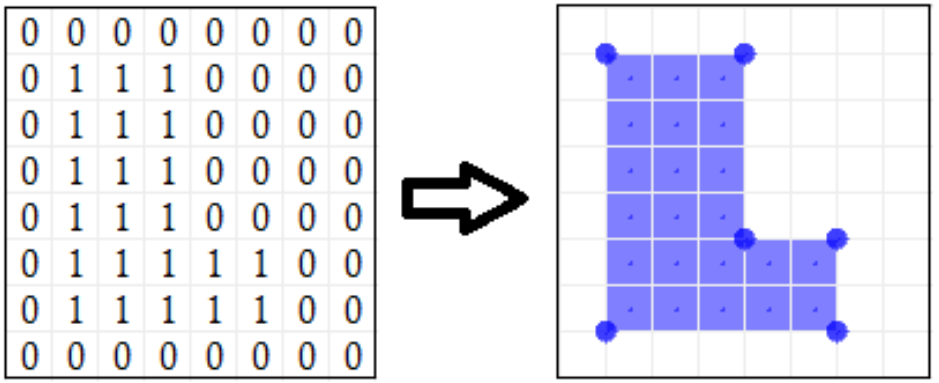




Motivation

Geomatrix is an attempt to exploit the traits of such matrices in order to find a more compact representation.

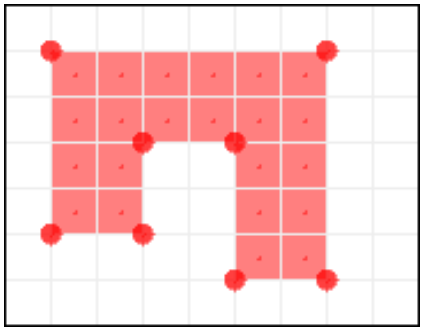
The idea is to see the matrix as a plane. The plane looks like a grid, where each element corresponds to a cell.
The set of cells corresponding to a 'true' value in the matrix form an area which characterizes the original matrix.



Motivation

Advantages of representing matrices as an area:
- We don't need to store each single value, we just have to maintain the representation of the area.
- Since an area is a geometric object, we will be able to exploit this fact in our algorithms.

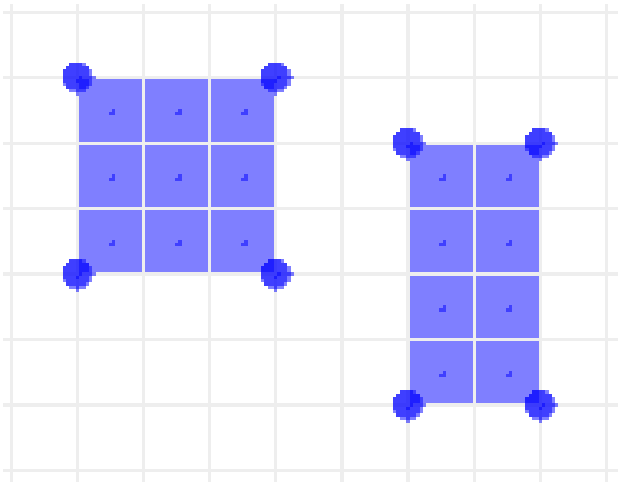
However, some things will get complicated.
For instance, to simply access a value, we will have to check whether it is contained in the area.



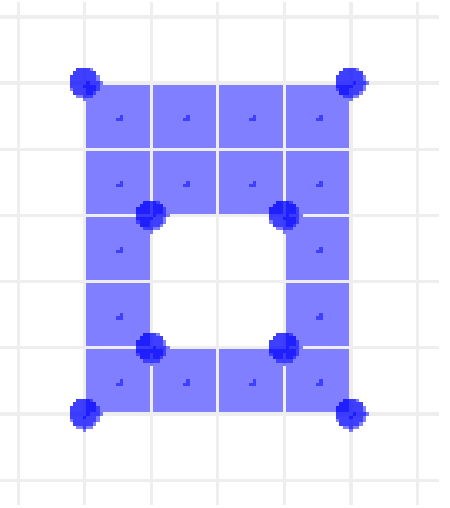
Is the element M[4][3] contained?

Area representation

- Areas can have disjoint parts



- Areas can have holes

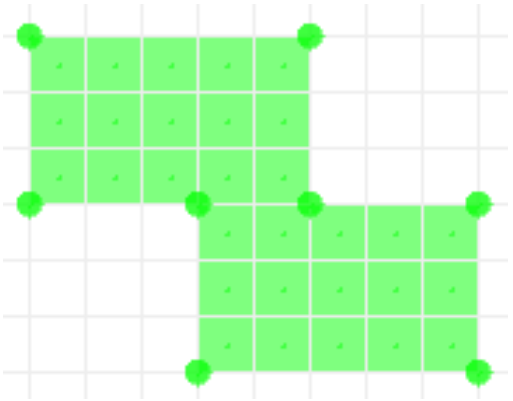


What data structures to use to represent areas?

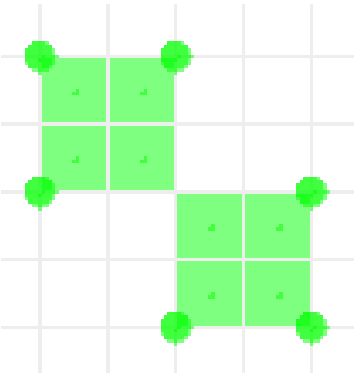
Area representation

Definition: the vertices of an area are the points of the grid plane such that an odd number of adjacent cells are contained in the area.

- In the following picture, the points that are vertices are highlighted



- Notice how the middle point is not a vertex, because it has 2 adjacent contained cells.



Let f be the function that, given an area, returns the set of its vertices.

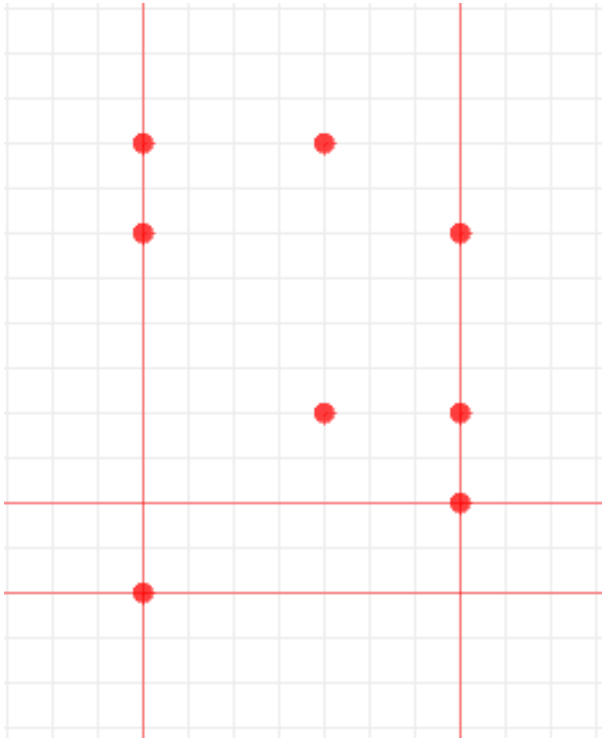
Theorem 1: f is bijective.

Corollary: we can represent any area with just the unordered set of its vertices.

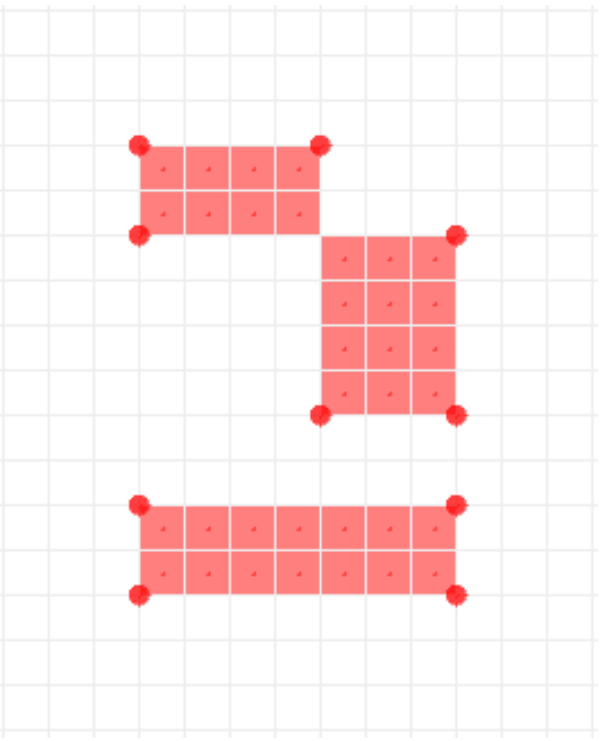
Area representation

Theorem 2: The codomain of f is the set of sets of points such that every line of the grid has an even number of them.

- No area has this set of vertices, because some lines (highlighted) have an odd number of points:



- Once the balance is established again, an area is defined:



Area representation

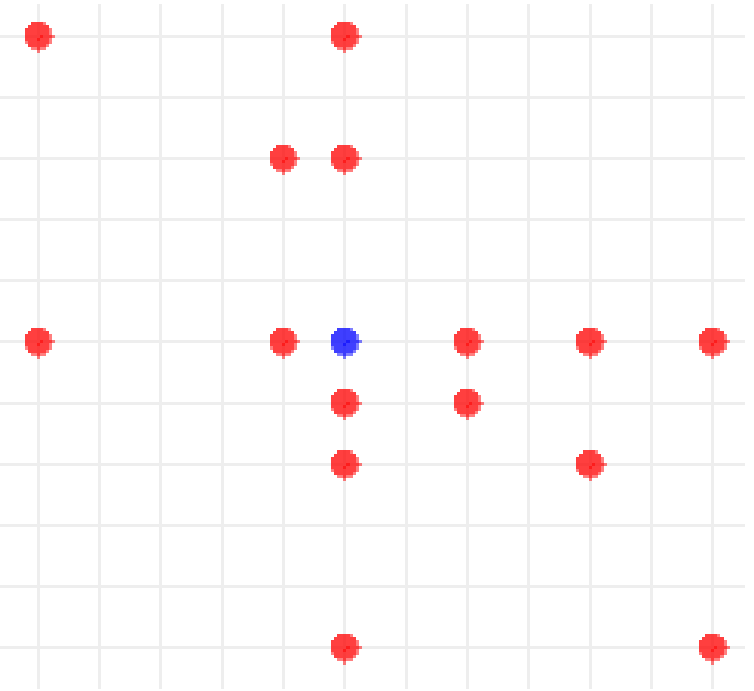
Corollary: Let l be a line with some vertex v of an area a.
Then, if we split l at v, exactly one of the resulting two fragments of l has an odd number of vertices of a.

If we represent an area a with the set of its vertices, we must be able to compute the original area back from the set of vertices.
In particular, we want to find its edges. The following observation, based on the previous collorary, gives a method to compute the set of edges of an area from the set of its vertices.

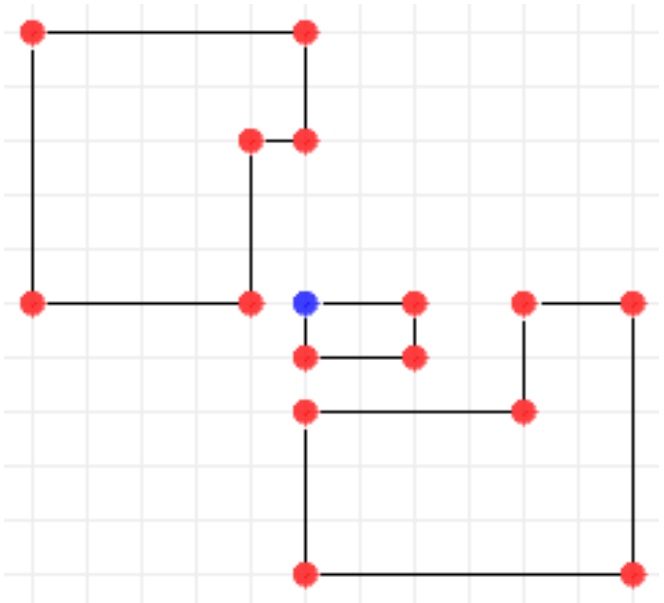
Observation: let v be a vertex of an area a. Then, there is a vertical edge between v and the closest vertex in the same vertical line in the direction in which there is an odd number of vertices.
Similarly, there is a horizontal edge between v and the closest vertex in the same horizontal line in the direction in which there is an odd number of vertices.

Area representation

Example: this set of points define an area, because there is an even number at each line.
But, how to know which?



Consider the blue point. It has 3 vertices to the right and 2 to the left. Therefore, there is an edge between the blue dot and the closest vertex to its right. With the same method, you can find any edge:



Operations

Now that the representation is well defined, what can we do with it?

First of all, we want the operations we would have in an usual matrix: consulting and modifying elements.

Moreover, because an area is a set of cells, we have set operations: union, intersection, difference, symmetric difference, subset check and cardinality.

Geometric operations: translation, rotation, reflection and rectangulation (decomposing an area into rectangles).

Finally, we have some other operations, like returning the contained cells as a list.

Operations

Notation:

a, a_1, a_2 denote areas.

n, n_1, n_2 denote their cardinality (the number of contained cells)

m, m_1, m_2 denote their number of vertices.

Assumption: in general, $n \gg m$. This is just a way to state that the values of the matrix are mostly agrupated in rectangular chuncks (otherwise, we wouldn't be using Geomatrix).

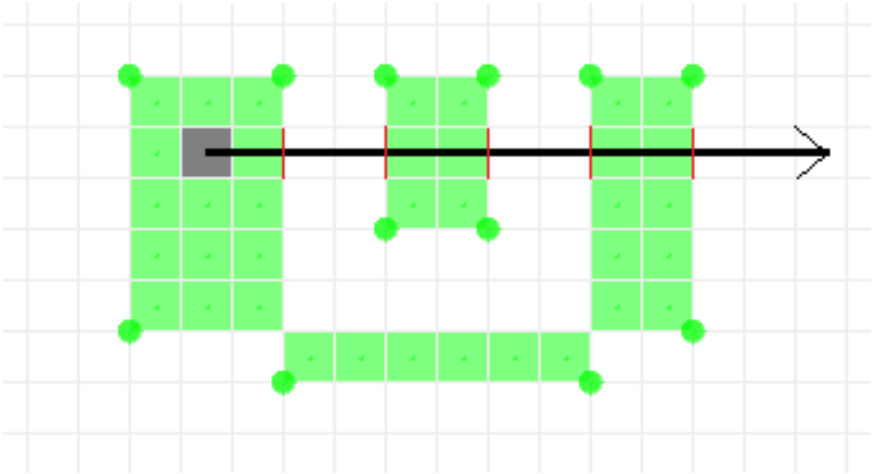
General guideline for implementing operations: the temporal complexity must always depend on m, not n.

Membership check

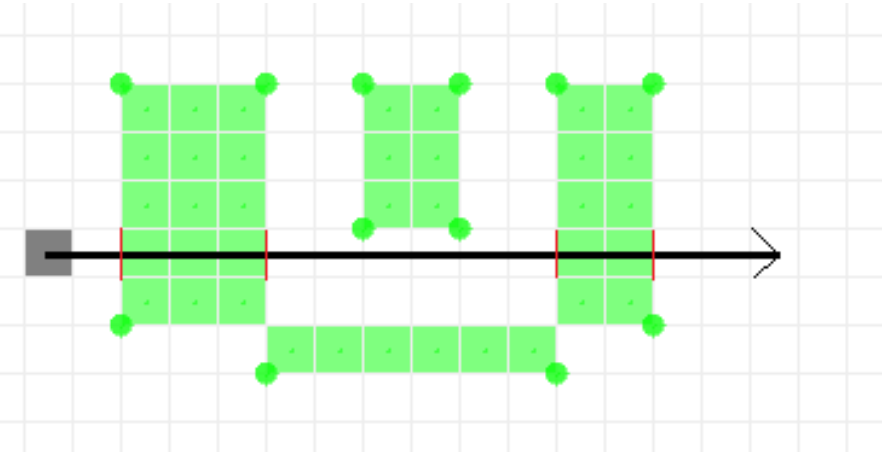
Check if a given cell belongs to an area.

Ray Casting algorithm: propagate a ray from the cell to its right and count how many vertical edges intersect the ray

- If the cell is contained, there will be an **odd** number of intersections:



- If the cell is not contained, there will be an **even** number of intersections:



Membership check

Analysis:

Every edge has 2 vertices as endpoints, and no vertice can be endpoint of 2 vertical or 2 horizontal edges at the same time.
Therefore, number of vertical edges = number of horizontal edges = $\frac{m}{2}$

The cost is proportional to the number of vertical edges: $O(m)$

Optimization:
Keep vertical edges sorted. Check for intersections from right to left, and stop once you are past the origin of the ray.

Note: cost with a traditional matrix: $O(1)$ *(ouch!)*

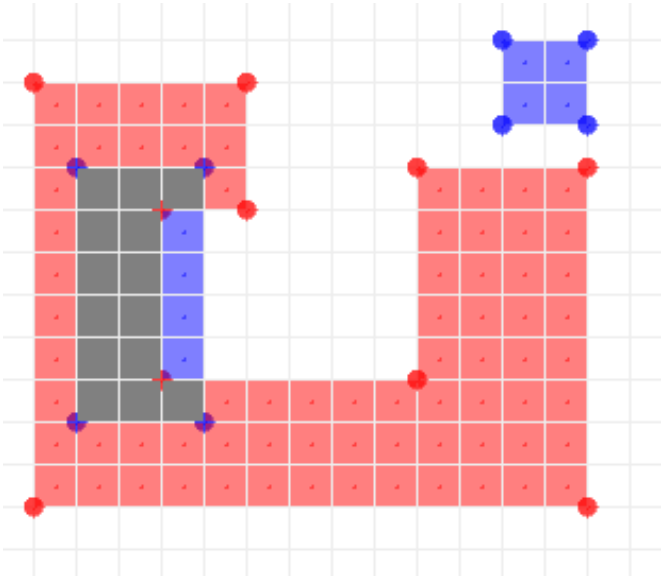
Subset problem

Check if an area a_2 is contained in another area a_1

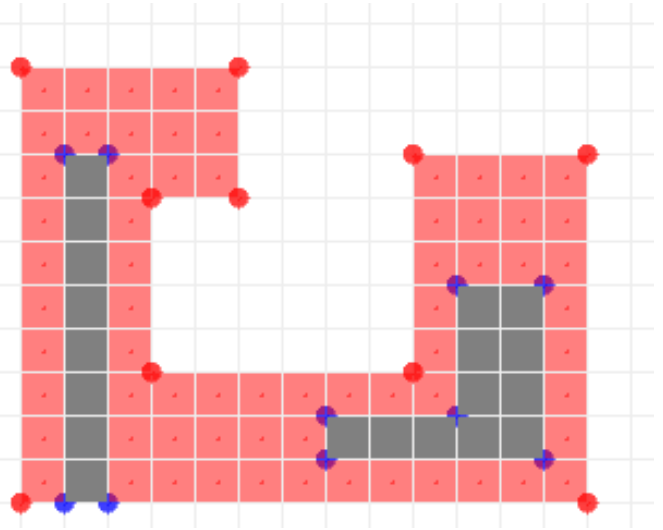
Naive algorithm:

1. Check that no edges intersect
2. Check that all vertices of a_2 are contained in a_1
3. Check that no vertex from a_1 is contained in a_2

- The blue area is not contained in the red area:



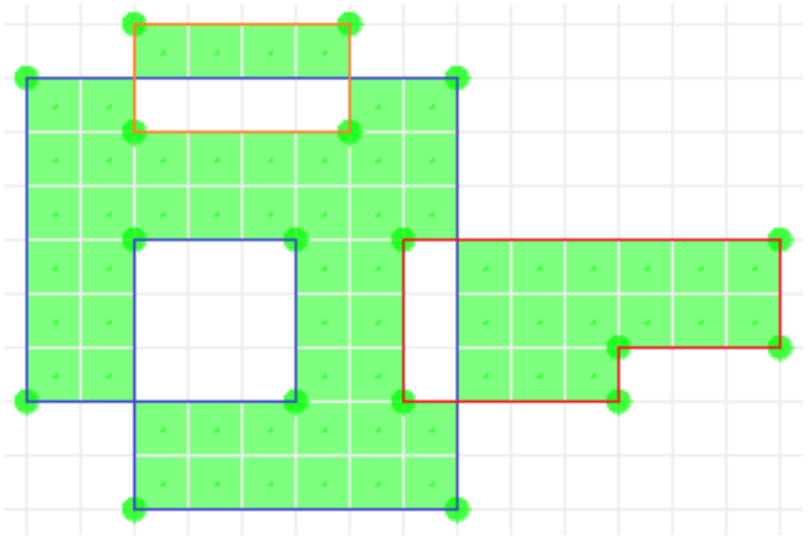
- The blue area is contained in the red area:



Subset problem

To improve the algorithm, we have to introduce the notion of cycles of edges.

Any vertex is the endpoint of exactly one vertical edge and one horizontal edge. If we start at some vertex, and traverse the edges as if we were doing a depth first search in a graph, we end up in the same vertice.



An area with 3 cycles of edges.

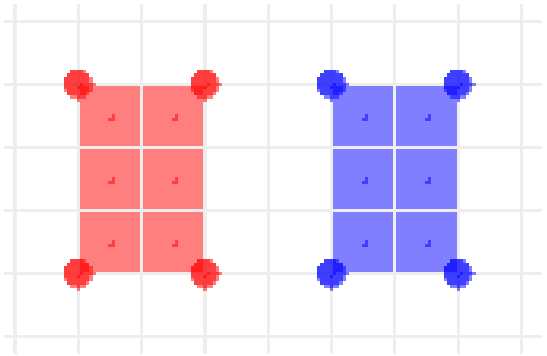
If a vertex v of a_1 is contained in a_2 and the edges of a_1 and a_2 don't intersect, then all the vertices in the same cycle as v are contained in a_2 too.

Subset problem

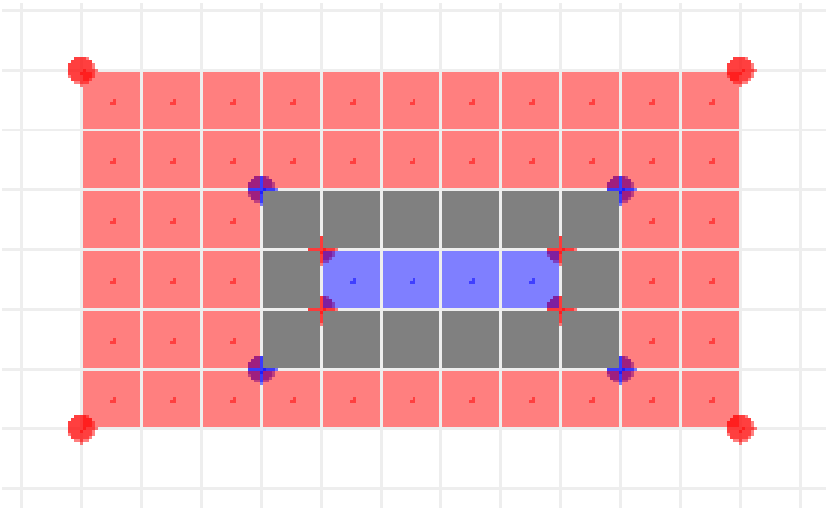
Improved algorithm:

1. Check that no edges intersect
2. Check that **at least one** vertex of a_2 is contained in a_1
3. Check that, for each cycle of edges of a_1, **at least one** vertex is not contained in a_2

- Case 2 is necessary in case a_1 and a_2 are disjoint:



- Case 3 is necessary in cases where a_1 has a hole. a_2 may not be contained even though there are no edge intersections:



Subset problem

Analysis:

1. Check that no edges intersect:

We have to check each vertical edge of a1 with each horizontal edge of a2 and each horizontal edge of a1.
Total: $O(m_1 \times m_2)$

2. Check that at least one vertex of a_2 is contained in a_1:

We have to do one membership check in a_1: $O(m_1)$

3. Check that, for each cycle of edges of a_1, at least one vertex is not contained in a_2:

A cycle of edges has at least 4 vertices. Therefore, we have to do at most $m_1/4$ membership checks:

$$\frac{m_1}{4} \times O(m_2) = O(m_1 \times m_2)$$

Total cost: $O(m_1 \times m_2)$

Subset problem

Optimization:

Supose m_2 is large, and we have a short edge in a_1. Instead of checking every edge of a_2, we can check only the edges along one of the few lines that intersect with it.

To do this, we can maintain a hash table that returns, for a given line, the list of edges along it.
When trying to see if an edge of a_1 intersects with some edge of a_2, use the hash table to access only the relevant edges.

Use this optimization **only** when the length of the edge is small relative to m_2.

Union

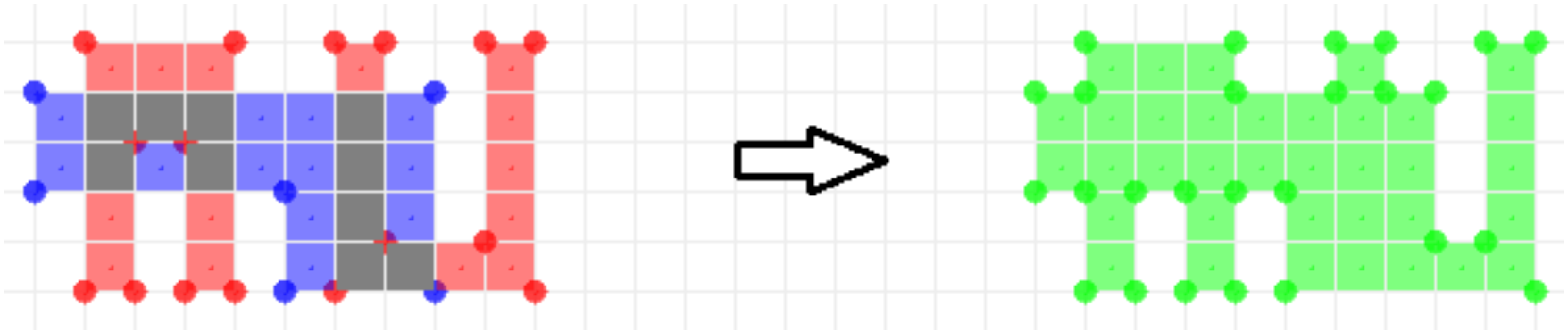
Find the union of a_1 and a_2

Since an area is characterized by its vertices, we only need to find which vertices the union will have.

Observation: the candidates to vertices of the union are:

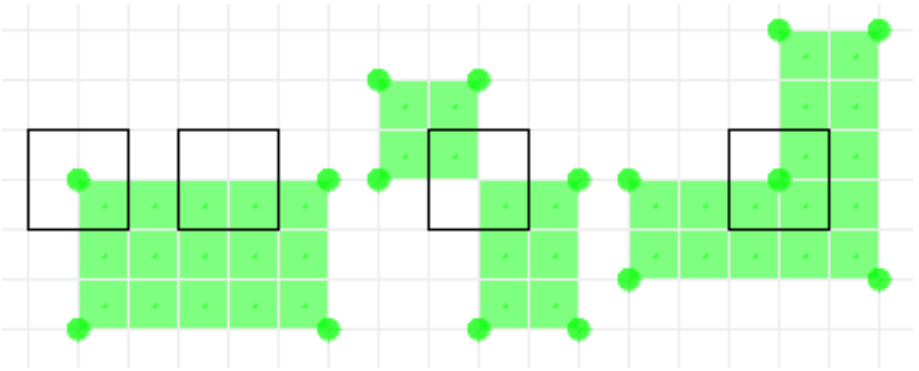
1. vertices of a_1
2. vertices of a_2
3. intersection points between edges of a_1 and a_2

How to know which ones?



Union

Remember: the vertices of an area are the points surrounded by exactly an odd number of cells contained in the area:



The valid candidates will be those with an odd number of adjacent cells contained in the union.

Algorithm:

For each candidate vertice:

1. Count the number of adjacent cells contained in a_1 or a_2 (or both) using membership checks.
2. If the number is 1 or 3, then the point is a vertice of the union.

Union

Optimizations:

It is expensive to do 8 (4 per area) membership checks for each candidate vertice.

Doing independent membership checks for adjacent cells is overkill, because they will be both contained or not contained unless there is an edge between them.

We can do only one membership check for one of the 4 adjacent cells, and figure out if the rest are contained using the previous observation. To quickly access the edges in a given line, use the hash tables mentioned earlier.

Moreover, in the case of edge intersections, we don't have to do any membership check, because they are always valid candidates (except double intersections, as seen below).

Union

Analysis:

1. vertices of a_1:

For each one, we have to do a constant number of membership checks in a_1 and a_2. Total: $O(m_1 \times \max(m_1, m_2))$

2. vertices of a_2:

Analogous to the previous case. Total: $O(m_2 \times \max(m_1, m_2))$

3. intersection points between edges of a_1 and a_2:

We have to figure out the edge intersections, but we don't need to do any membership checks. Total: $O(m_1 \times m_2)$

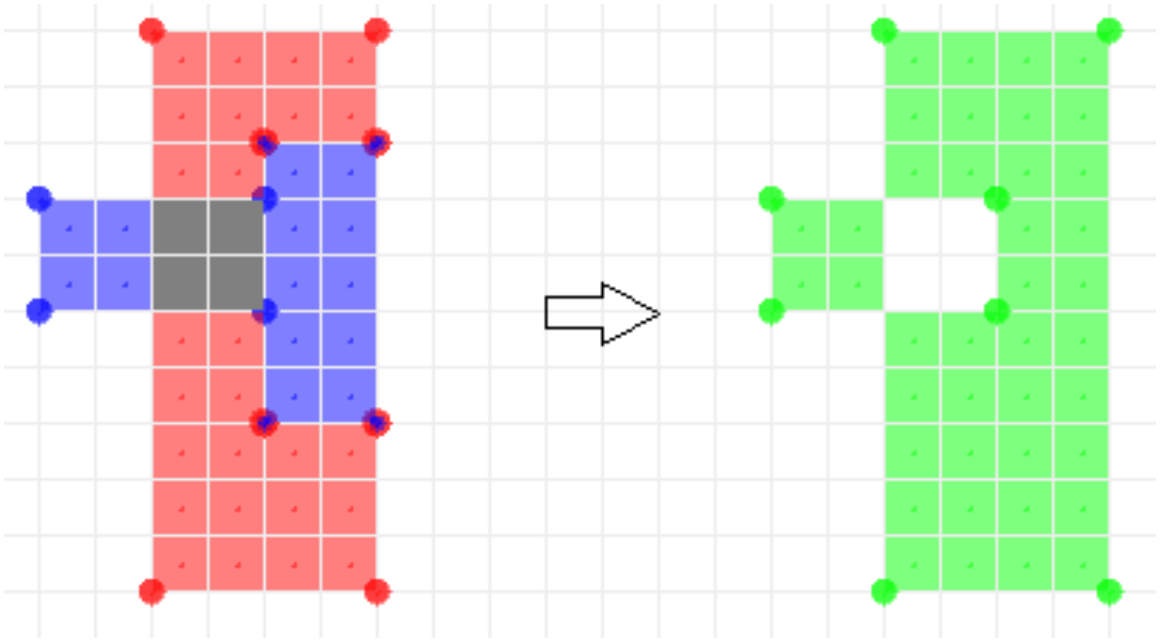
Total cost: $O(\max(m_1, m_2)^2)$

The fact that the cost is independent of n means that, in Geomatrix, it's equally expensive to add one element than to add one million, as long as they are 'shaped' in a rectangle.

Symmetric difference

Find the set of cells contained in either a_1 or a_2, but not both.

Observation: the set of vertices of the symmetric difference of a_1 and a_2 is the symmetric difference of the sets of vertices of a_1 and a_2.



The repeated vertices are not part of the symmetric difference, but every other vertex is.

Symmetric difference

Algorithm:

Initialize a hash table with the vertices of a_1.

For each vertice of a_2:

If it is in not in the hash table: add it.
Else: remove it.

At the end of the loop, the hash table contains the vertices of the symmetric difference.

Analysis:

We require constant time work for each vertice of a_1 and a_2:

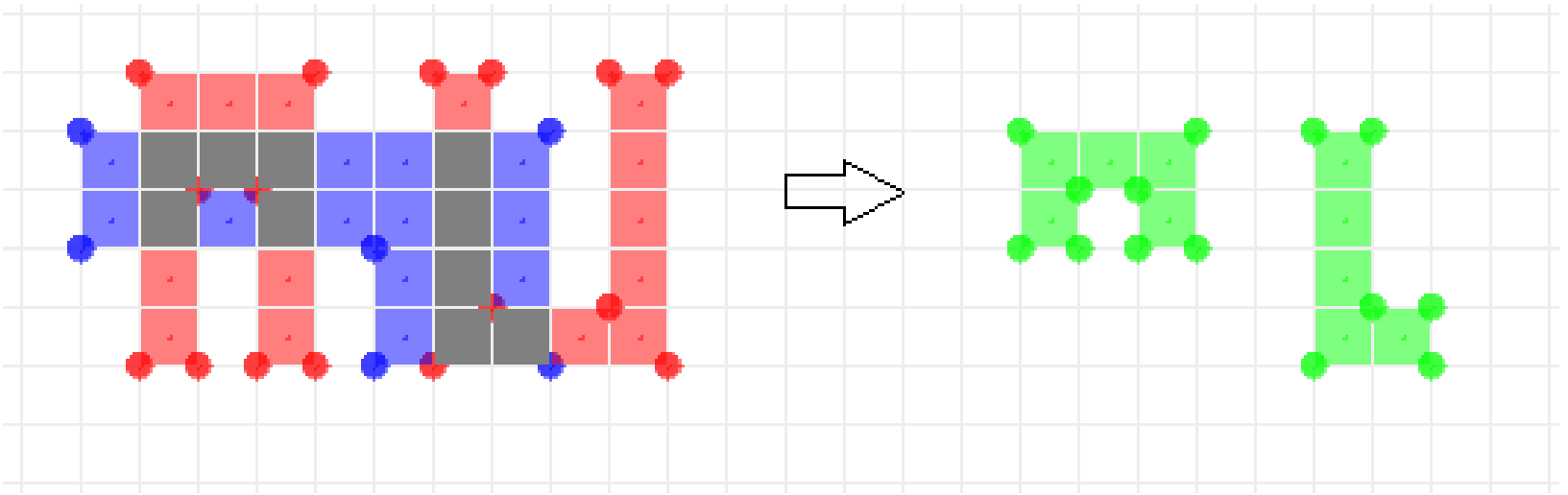
$$O(m_1 + m_2)$$

Intersection

Find the intersection of a_1 and a_2.

Just like with the union, the candidates to vertice of the intersection are the vertices of a_1 and a_2 and the edge intersections.

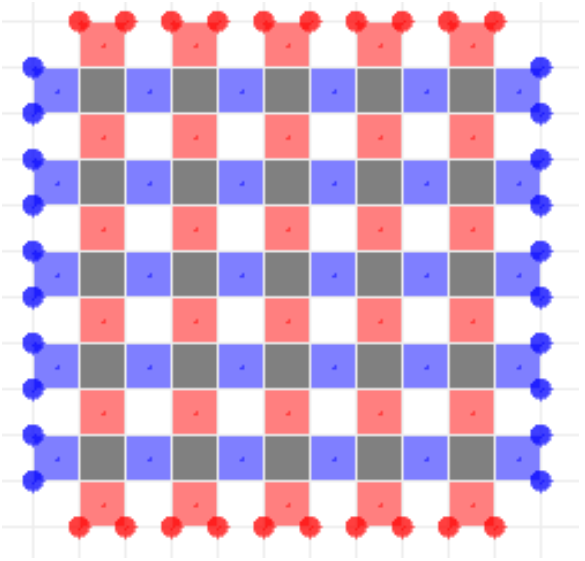
This time, the valid candidates will be those with an odd number of adjacent cells contained in the intersection (contained in both a_1 and a_2).



Intersection

Unlike the union, edge intersections might or might not be vertices of the resulting area.
Therefore, you have to do membership checks of the adjacent cells of each edge intersection.

In the worst case scenario, the number of edge intersections is $O(m_1 \times m_2)$.



A situation where the number of edge intersections is quadratic in the number of vertices. It is scalable to arbitrarily large areas.

Intersection

Performing $O(m_1 \times m_2)$ membership checks would rise the asymptotic cost to $O(m_1 \times m_2 \times \max(m_1, m_2))$.

However, with the following result of set theory, we can find a more efficient method:

$$A \cap B = (A \triangle B) \triangle (A \cup B)$$

(where $\triangle$ denotes the symmetric difference)

We can calculate the intersection performing 1 union and 2 symmetric difference operations (which, as we have seen, take linear time).

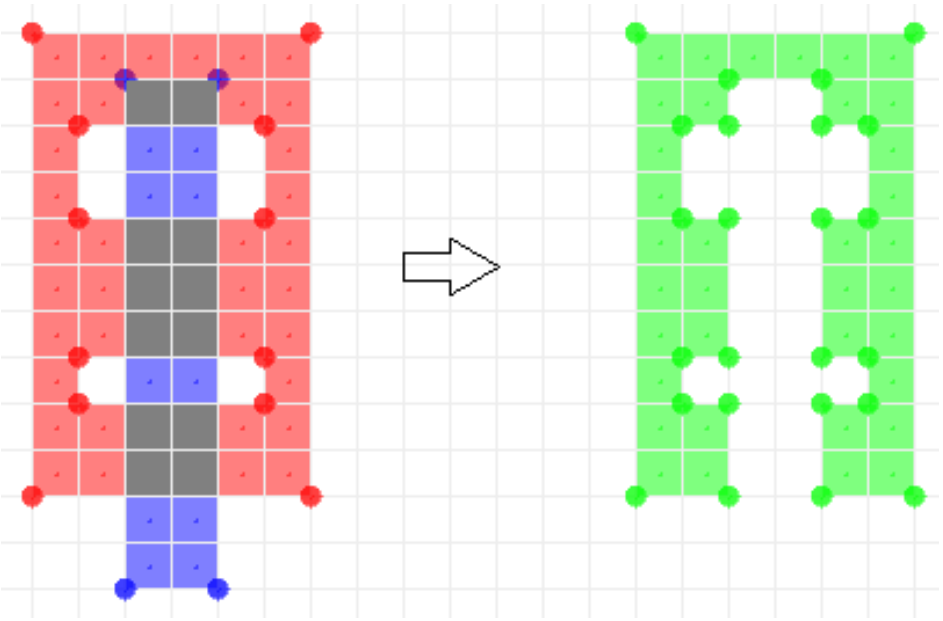
With this approach, we have the same asymptotic cost as the union.

Difference

Find the set of cells contained in a_1 but not in a_2

Again, we can use a result from set theory to reduce the difference operation to a combination of operations already defined:

$$A \setminus B = (A \cup B) \triangle B$$



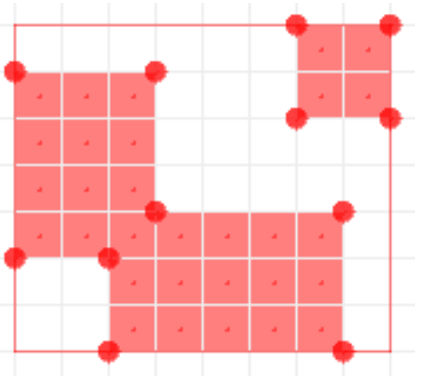
Since the cost of the union dominates the cost of the symmetric difference, this operation has the same cost as the union.

List of contained cells

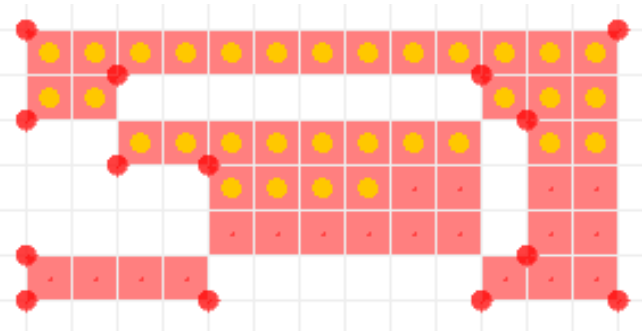
Returns a list with the set of contained cells

Naive algorithm:

Compute the bounding rectangle
For each cell within it:
    if it is contained, add it to the list



bounding rectangle



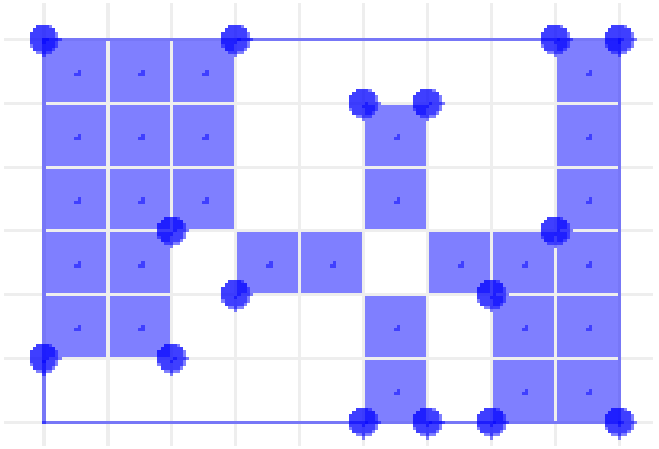
An area halfway through the process of iteration

Note: in the actual implementation, this operation returns an iterator through the contained cells instead of a list, which is basically equivalent.

List of contained cells

Again, we are doing overkill by repeatedly doing the membership check for adjacent cells.

We can do a single membership check for each row, and store the position of all the vertical edges that cross it. The contained cells will be those between the 1st and 2nd edges, between the 3rd and the 4th, and so on.



For any given row, the contained cells are those between the 1st and the 2nd vertical edge that cross it, between the 3rd and 4th edges, and so on.

List of contained cells

Analysis:

Clearly, we have to traverse all the contained cells, so in this case it will be impossible to design an algorithm not depending on n.

Moreover, for each row, we have to traverse the vertical edges to see which ones cross the row.
Assuming the area has a similar height and width, the number of rows is $O(\sqrt{(n)})$
Therefore, the total cost is:

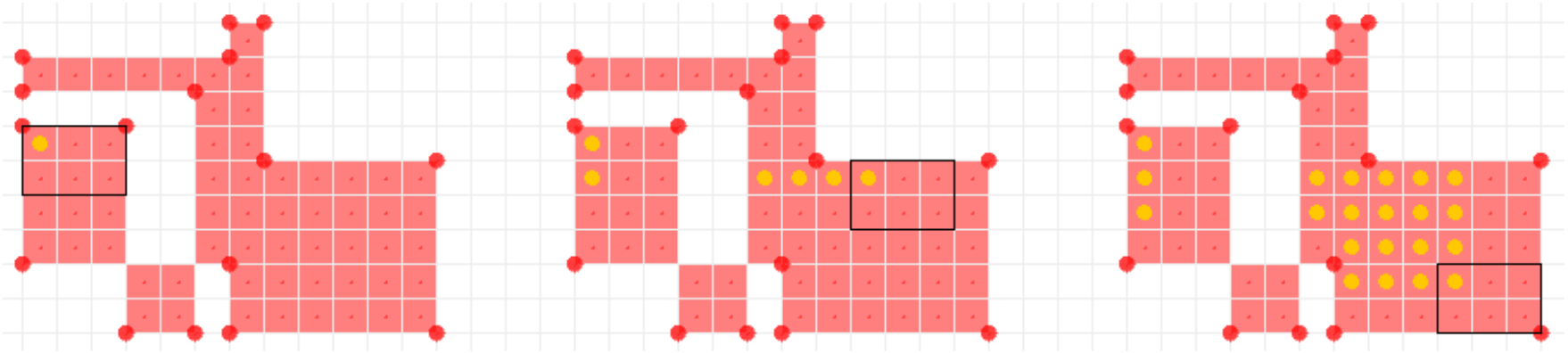
$$O(n + m\sqrt{n})$$

Optimization:

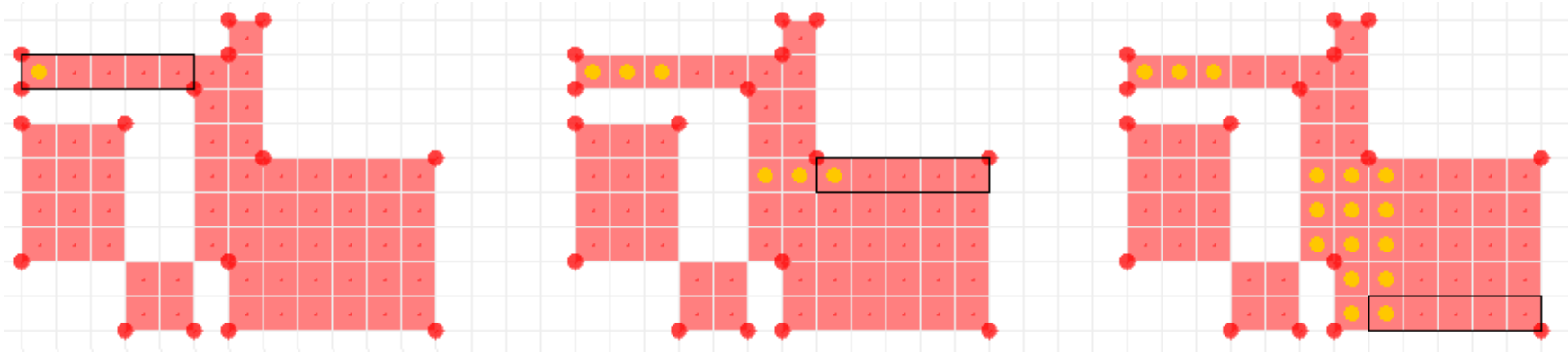
The first time we perform this operation, we can store in a hash table the list of vertical edges that cross each row. Therefore, for subsequent operations, we won´t need to go over the vertical edges at each row, and we will have an asymptotic cost of $O(n)$, the optimal lower bound.

List of contained rectangles

In this case, we want to find all the positions where we can place a rectangle of certain dimensions such that it is fully contained in the area.



The pictures show the algorithm iterating through the contained rectangles of dimensinon 3x2. The yellow dots denote the top left cell of the rectangles.



Different rectangle dimensions result in different sets of cells.

List of contained rectangles

Naive algorithm:

Generate the list of contained cells (with the algorithm we have seen).
For each contained cell:
    Check if the rectangle with the cell as top-left cell is contained in the area.

A rectangle is an area with 4 vertices. Therefore, the cost of the subset checks is $O(m \times 4) = O(m)$

If we have to do this for each cell, the total cost will be $O(n \times m)$

This cost is **very high** because we have to perform a linear amount of work for every single cell. We will try to improve it.

List of contained rectangles

Idea: we can try to trim the positions where the rectangle will obviously not fit.

Observation: Let w be the width of the rectangle.
Then, the left vertical edge of the rectangle will be at least w cells apart from the area's closest vertical edge directly to the right of it.
Similarly, let h be the height of the rectangle. The top horizontal edge of the rectangle will be at least h cells apart from the area's closest horizontal edge directly below it.

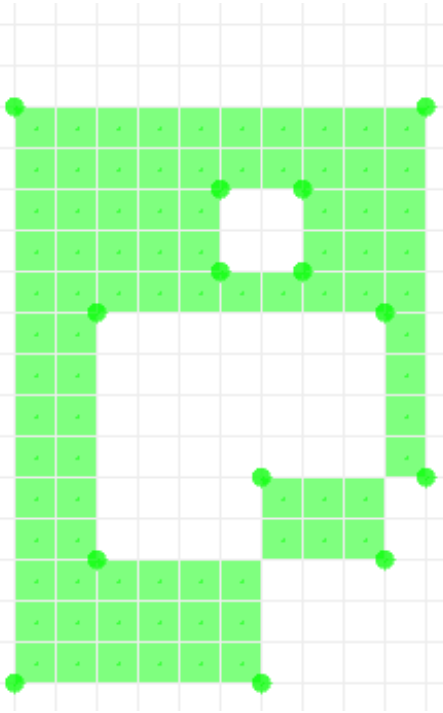
With this in mind, we can trim many positions. In the following algorithm, the intersection is the trim:

Algorithm:
1. Make two copies of a: a_2 and a_3
2. Shift a_2 $w-1$ units to the left (-1 to account for the width of the top-left cell)
3. Shift a_3 $h-1$ units upwards (-1 to account for the height of the top-left cell)
4. Compute the intersection of a, a_2 and a_3
5. Run the previous algorithm in the resulting area

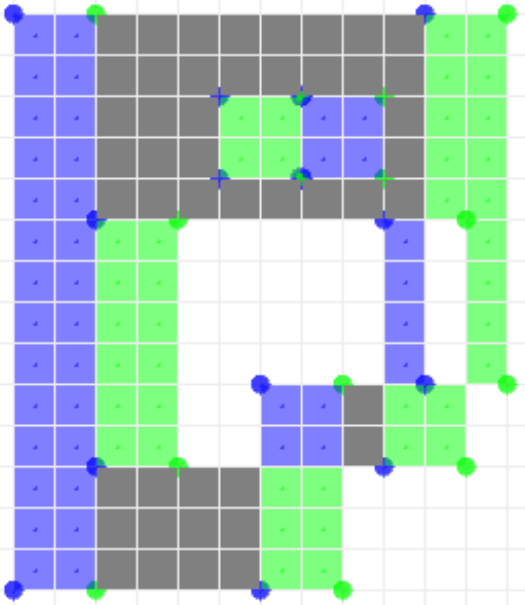
List of contained rectangles

Example: imagine that our rectangle has dimensions (3,3).

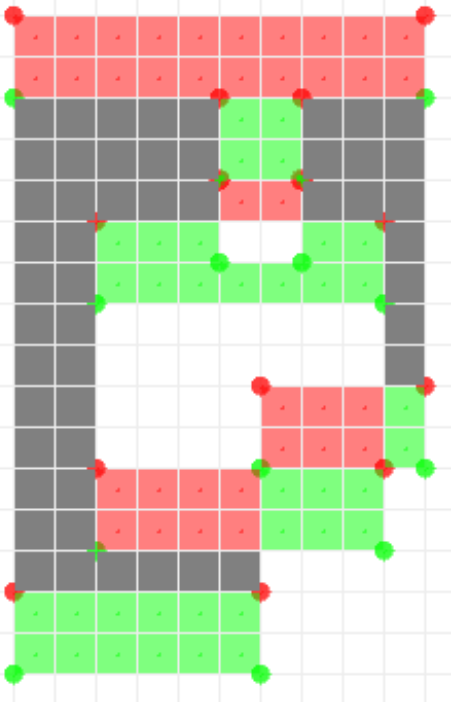
Original area:



Shifted 2 units to the left:

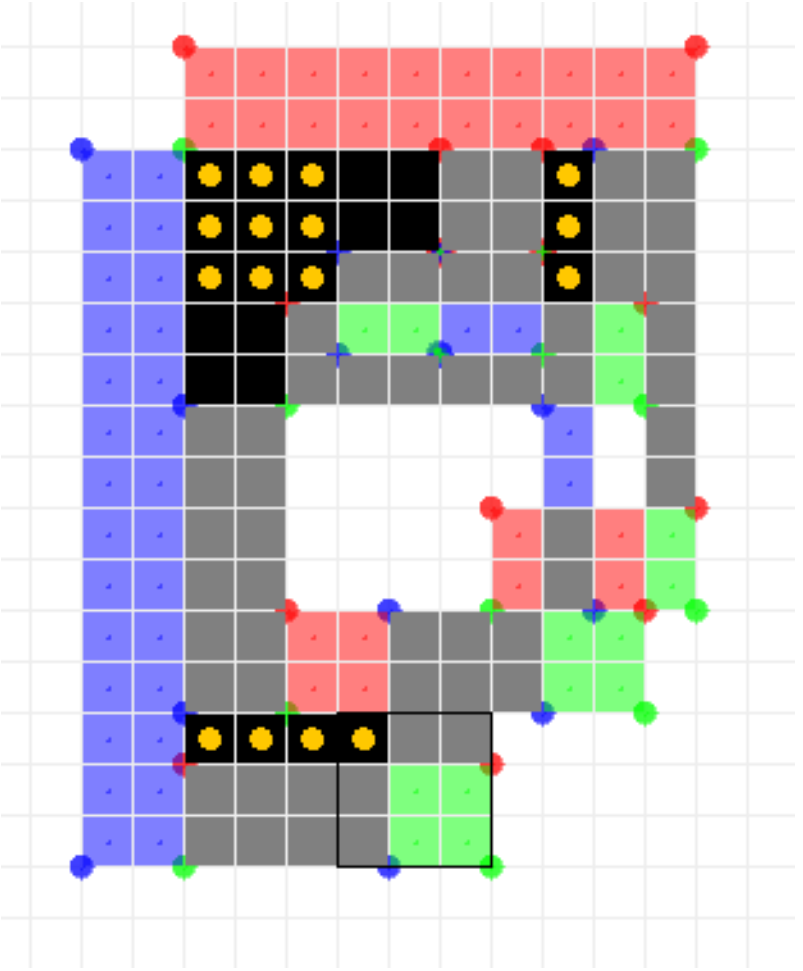


Shifted 2 units upwards:



List of contained rectangles

- The black cells are the intersection of the 3 areas.
- The yellow dots denote the solution (positions where a 3x3 rectangle can fit inside the original green area).
- The grey and green cells are the **trimmed cells**.



To make the trim, we had to perform 2 shifts (just moving the vertices a fixed amount) which have cost $O(m)$ each, and 2 intersection operations, with cost $O(m^2)$ each.
Nevertheless, the dominant part of the cost is still $O(n \times m)$.

List of contained rectangles

As we can see in the previous image, the trim took away many invalid positions, but not all of them. Therefore, we still had to check each of the remaining positions to see if the rectangle could fit.

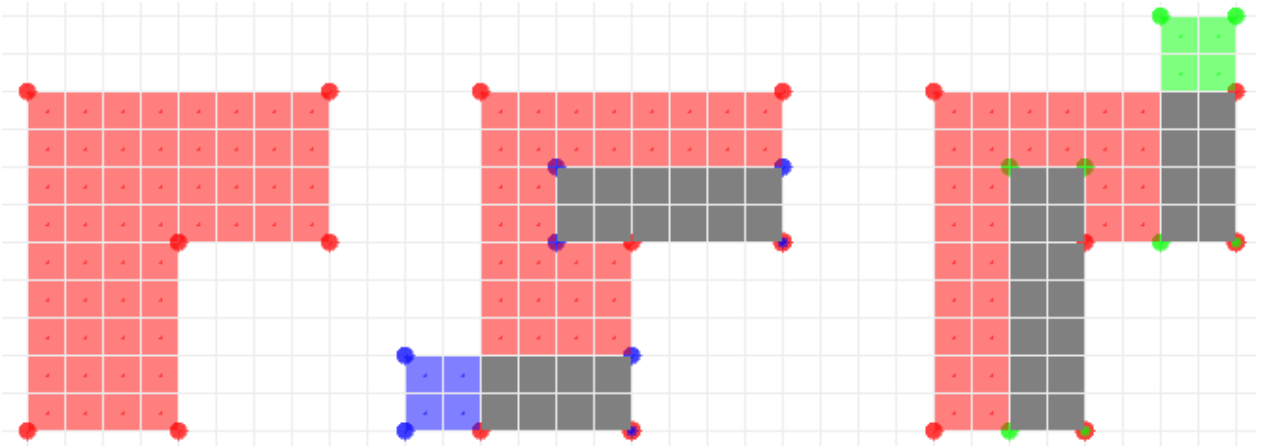
Goal: make an **exhaustive trim**

A rectangle whose top-left cell is within an area, will be fully contained in the area unless its edges intersect with the edges of the area.

Consequently, if we detect **for each edge** of the area the conflicting positions and trim them, the resulting area will contain the exact positions where the rectangles can be placed.

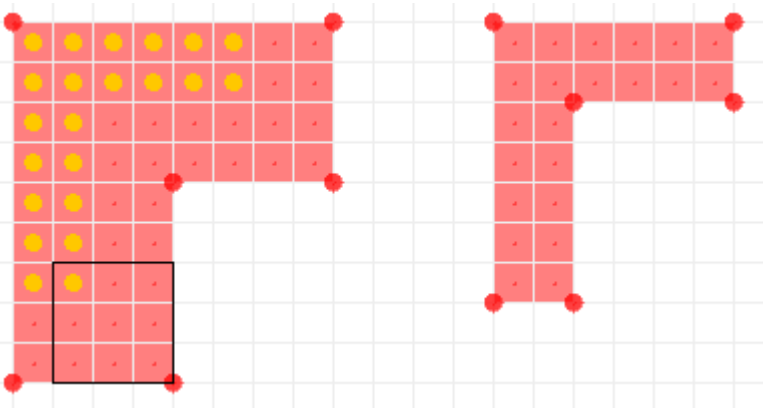
In particular, top and left edges won't have conflicting positions, because we consider as positions the top-left corner of the rectangles, so they don't expand in those directions.
On the other hand, bottom and right edges will have conflicting positions.

List of contained rectangles



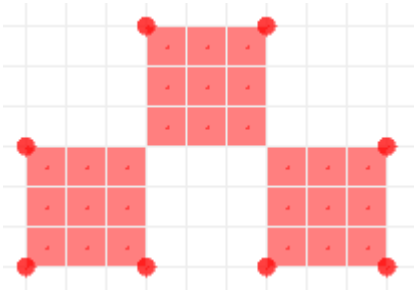
Example: consider the red area and a 3x3 rectangle. In blue, the positions that cause conflict with a bottom edge. In green, the positions that cause conflict with a right edge.

The yellow dots denote the solution.
The area to the right is the result after trimming the conflicting areas.
We can see that **the solution and this area match perfectly**.



List of contained rectangles

However, there is a flaw: the notion of “bottom” and “right” edges is not well defined, because some times edges can act as both top and bottom or left and right.



The largest horizontal edge is both top and bottom at different segments.

Solution: split edges in points where different edges intersect. We call the resulting set of segments (including the full edges that didn't intersect with other edges) *pseudoedges*.

A pseudoedge always has all the adjacent contained cells in the same side, so they can legitimately be called top/bottom or right/left pseudoedges.

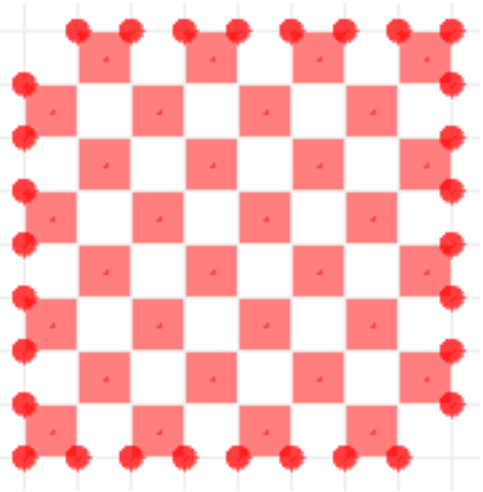
List of contained rectangles

Final algorithm:

1. Compute pseudoedges.
2. For each pseudoedge, compute the conflicting rectangle and substract it from the original area.
3. Return the list of cells contained in the area after all the substractions.

Analysis:

The method to compute pseudoedges is not detailed here, but its cost is smaller than the other steps of the algorithm.
Anyhow, in a worst case scenario, the number of pseudoedges can grow quadratically in the number of edges.



List of contained rectangles

For each of the $O(m^2)$ pseudoedges, we have to perform a substraction operation. Since the area we substract is always a rectangle, the cost of the substractions is $O(m)$ each.

Therefore, the second step has cost $O(m^3)$ in the worst case.

Finally, computing the list of cells of an area has cost $O(n + m\sqrt{n})$. The size of the area after all the substractions will depend on the size of the rectangle, but we know that at most it will be n, which corresponds to the case of a 1x1 rectangle.

Putting it all together, we get a total cost of:

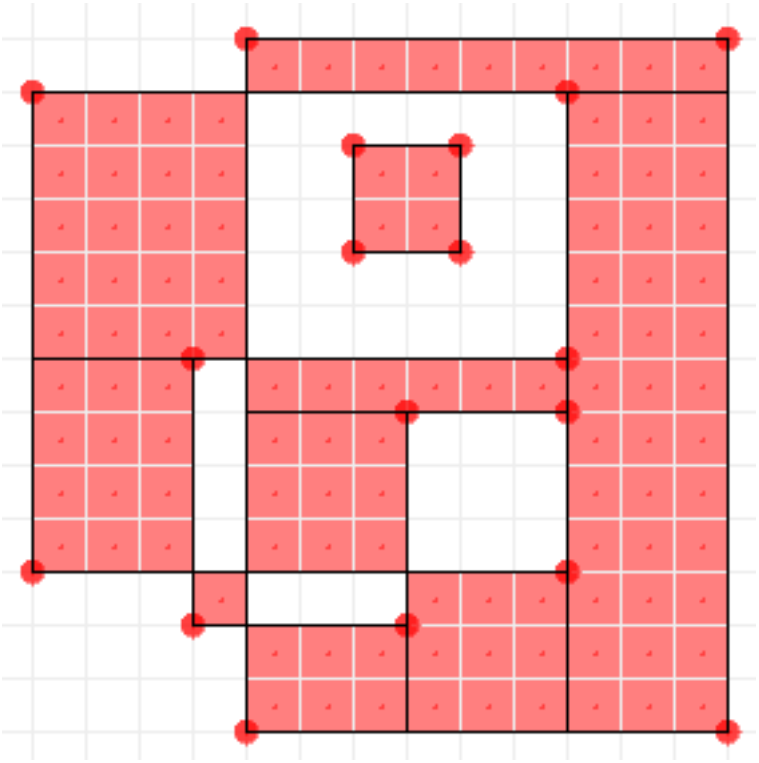
$$O(n + m\sqrt{n} + m^3)$$

Bottom line: we managed to get rid of the $n \times m$ factor by doing a more expensive preprocessing work.
Under the assumption that $n \gg m$, this change is very effective.

And more...

There are several more operations we can do with Geomatrix.

Some of them are not even resolved yet.
For example, finding an efficient algorithm for rectangulization (decomposing an area into a minimal number of rectangles).



Rectangulization: holes make naive greedy algorithms fail

Summary

Operation	Complexity
Membership check	$O(m)$
Subset check	$O(m_1 \times m_2)$
Union	$O(max(m_1, m_2)^2)$
Intersection	$O(max(m_1, m_2)^2)$
Difference	$O(max(m_1, m_2)^2)$
Symmetric Difference	$O(m_1 + m_2)$
Contained cells	$O(n + m\sqrt{n})$
Contained rectangles	$O(n + m\sqrt{n} + m^3)$

Under the assumption $n \gg m$, Geomatrix is an efficient alternative when you need this kind of operations