École polytechnique fédérale de Lausanne

Semester project

Bachelor in Mathematics

The stabbing problem

Author:
Florence Osmont

Supervisor:
Martina Gallato

Contents

Abstract

We present a PTAS for the general stabbing problem and an algorithm in $O(n^4)$ yielding an 8-approximation for the horizontal stabbing problem and its implementation.

Introduction of the problem

The general stabbing problem consists in finding a set of horizontal or vertical segments of minimal length that stabs a given set of n axis-aligned rectangles $\{R_i\}_{i=1}^n$ with $n \geq 1$. Concretely, each segment must starts and ends at the extremity of one or multiple rectangles. An example is given in the Figure 1. For any rectangle R_i let w_i be its width and h_i its height. Moreover, let $c(SOL)$ denotes the cost i.e. the length of the set of stabbing segments of the solution SOL. A restriction of this problem is the horizontal stabbing problem, where only horizontal segments are considered.

The problem was first enunciated by Chan *et al.* (2018), who proved its NP-hardness and the existence of a $O(1)$-approximation algorithm for both the horizontal and general problem. The constant-factor approximation relies on the decomposition of the instance into laminar ones. Then, Eisenbrand *et al.* (2021) presented a quasi-polynomial time approximation scheme (QPTAS) and an 8-approximation algorithm in $O(n^5)$ for the horizontal stabbing problem. Recently, Khan *et al.* (2021) found a $(2 + \epsilon)$-approximation algorithm in polynomial time for the general stabbing problem. Moreover, they also found a PTAS ($(1 + \epsilon)$-approximation algorithm in polynomial time) for some restrictions of this problem, such as the horizontal stabbing problem and the general one, with restrictions on the width and/or height of rectangles.

We will explain here the PTAS in $(n/\epsilon)^{O(1/\epsilon^3)}$ for the general stabbing problem with the restriction that for each rectangle its width must be at most its height. Then, we will present an 8-approximation algorithm in $O(n^4)$ derived from the one in Eisenbrand *et al.* (2021) and its implementation.

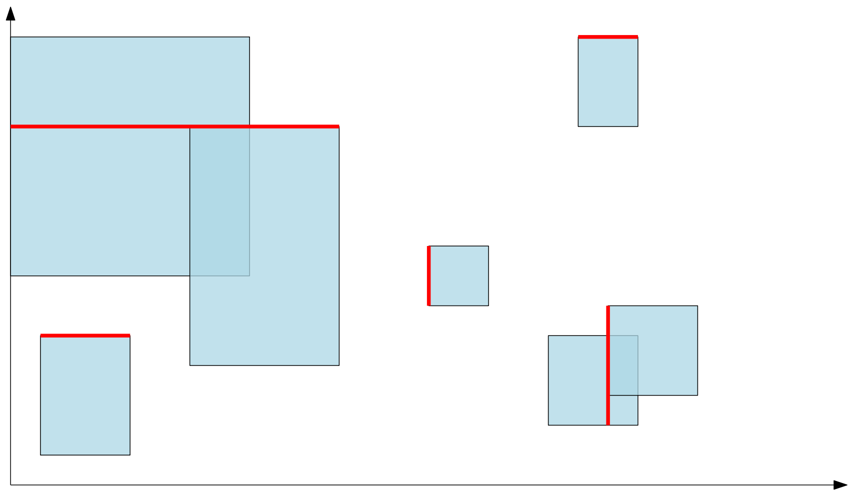


Figure 1: An example of the general stabbing problem

Part I

A PTAS for the general stabbing problem

In this part, we present a $(1 + O(\epsilon))$-algorithm in polynomial time for the general stabbing problem with the additional assumption that $w_i \leq h_i$ for any rectangle R_i. This result comes from Khan *et al.* (2021). The first section presents some preprocessing steps that are done to the instance. The second part presents the algorithm itself. The last part proves that this algorithm is in polynomial time and yields a $(1 + O(\epsilon))$ approximation.

1 Preprocessing

In this section, we present some preprocessing steps. The main goal of these steps is to reduce the number of possibilities the algorithm has to consider. Without loss of generality, we can assume that $1/\epsilon \in \mathbb{N}$.

Definition 1.1. *We say that some value $x \in \mathbb{R}$ is discretized if x is an integral multiple of ϵ/n.*

Theorem 1.1. *For any $0 < \epsilon < 1/3$, by losing a $(1 + O(\epsilon))$ factor in the approximation ratio, we can assume for each rectangle R_i that :*

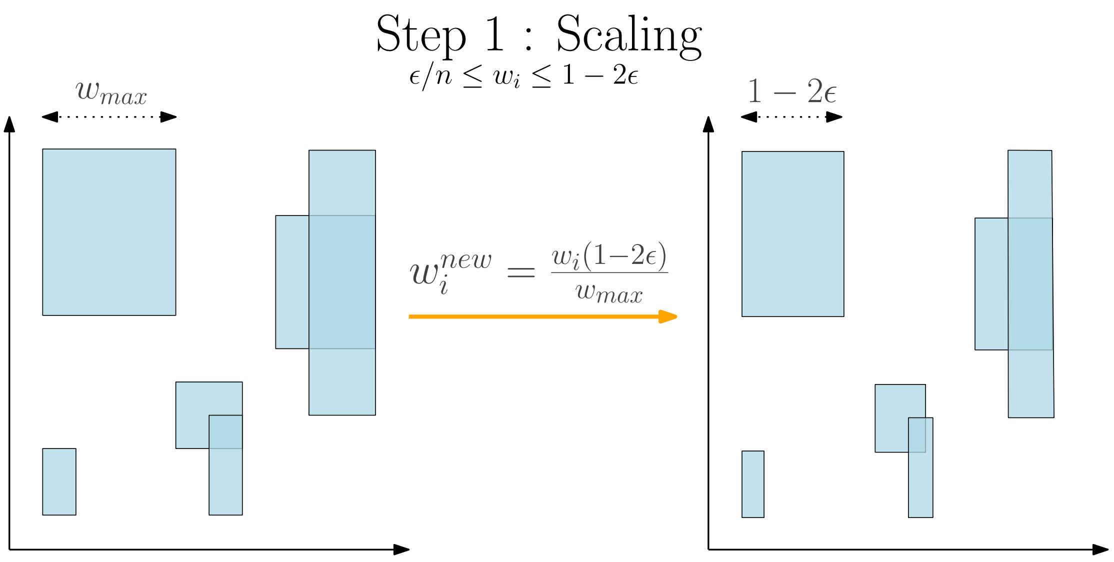
- $\epsilon/n \leq w_i \leq 1$,
- $x_1^{(i)}, x_2^{(i)} \in [0, n]$ *and are discretized,*
- $y_1^{(i)}, y_2^{(i)} \in [0, 4n^2]$ *and are discretized,*
- *each horizontal line segment in the optimal solution has width at most $1/\epsilon$.*

This is done in four steps :

Step 1 : Scaling Let R_{max} be the rectangle with the largest width w_{max}. For each rectangle R_i with width w_i, we change it to $w'_i = (1 - 2\epsilon/n)w_i/w_{max}$. Now R_{max} has width $1 - 2\epsilon/n$. Moreover, any rectangle with width less than ϵ/n can be stabbed greedily with segments of total length inferior to ϵ. Thus, for each rectangle $\epsilon/n \leq w'_i \leq 1 - 2\epsilon/n$. This step is shown in Figure 2.

Now we prove that this step only loss a factor of $1+O(\epsilon)$. We have that $\epsilon/n < 1/3n < 1/3$ as $n \geq 1$ and $c(OPT) \geq 1 - 2\epsilon/n$ as R_{max} must be stabbed. The optimal solution will also be scaled, thus the only addition to the cost comes from the greedy stabbing which gives a multiplicative loss of :

$$\frac{c(\text{Add at Step1})}{c(OPT)} \leq \frac{\epsilon}{1 - 2\epsilon/n} \leq \frac{\epsilon}{1 - 2(1/3)} = 3\epsilon = O(\epsilon)$$

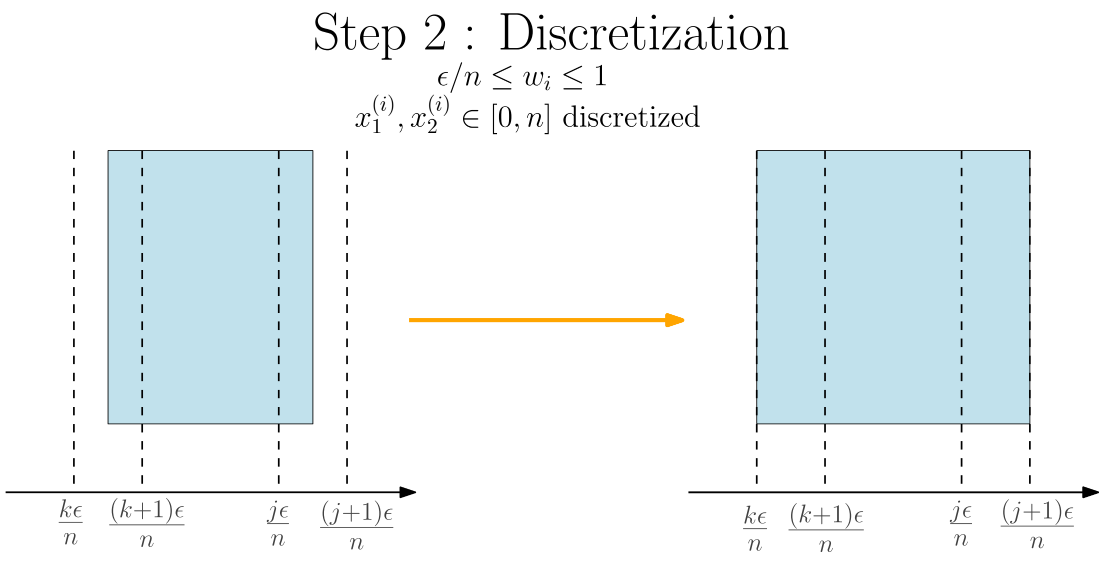


Step 1 : Scaling
$\epsilon/n \leq w_i \leq 1 - 2\epsilon$
w_{max}
$1 - 2\epsilon$
$w_i^{new} = \frac{w_i(1-2\epsilon)}{w_{max}}$

Figure 2: Scaling step

Step 2 : Discretization and bounding of the x-coordinates To discretize the x-coordinates of each rectangle, we extend them on both sides such that their new $x-$coordinates aligned with the nearest multiple of ϵ/n. This implies that for each rectangle $\epsilon/n \leq w_i' \leq 1$ as each rectangle's width was extended by at most ϵ/n. This step is shown in Figure 3. Moreover, we split the problem into subproblems with x-coordinates between 0 and n. This is possible because since the width of each rectangle is smaller than 1, the total width is smaller than n. Thus, if the smallest interval containing all the x-coordinates is bigger than [0,n] then some x^* coordinate is not covered by any rectangle and we split the instance into two smaller subproblems around it.

We added at most a factor of $2\epsilon/n \times n = 2\epsilon = O(\epsilon)$ to the cost of the solution.



Step 2 : Discretization
$\epsilon/n \leq w_i \leq 1$
$x_1^{(i)}, x_2^{(i)} \in [0, n]$ discretized
$\frac{k\epsilon}{n}$
$\frac{(k+1)\epsilon}{n}$
$\frac{j\epsilon}{n}$
$\frac{(j+1)\epsilon}{n}$
$\frac{k\epsilon}{n}$
$\frac{(k+1)\epsilon}{n}$
$\frac{j\epsilon}{n}$
$\frac{(j+1)\epsilon}{n}$

Figure 3: Discretization step

Step 3 : Discretization, stretching and bounding of the y-coordinates The heights of the rectangles in the input can be arbitrarily large as $w_i \leq h_i$. But, as we bounded the width by 1, the cost of the optimal solution $c(OPT)$ is less than n. Thus, there can not be a vertical segment of length bigger than n in OPT. We modify the y-coordinates as follows: if two consecutive coordinates are separated by a distance superior to $2n$, we reduce the distance between the two to the nearest multiple of ϵ/n that makes this distance less than $2n$ and, after, we do the same discretization step that was done for the x-coordinates. This step is shown in Figure 4

After this transformation, all the $2n$ y-coordinates are separated by at most $2n$ thus they are in the interval $[0, 4n^2]$. Moreover, the first part of this transformation didn't affect the cost of any optimal solution, as no segment could exist between two coordinates separated by a distance bigger than $2n$. Thus, the cost is affected in the same way as the transformation for the x-coordinate, and a factor of $2\epsilon = O(\epsilon)$ is added to the cost of the solution.

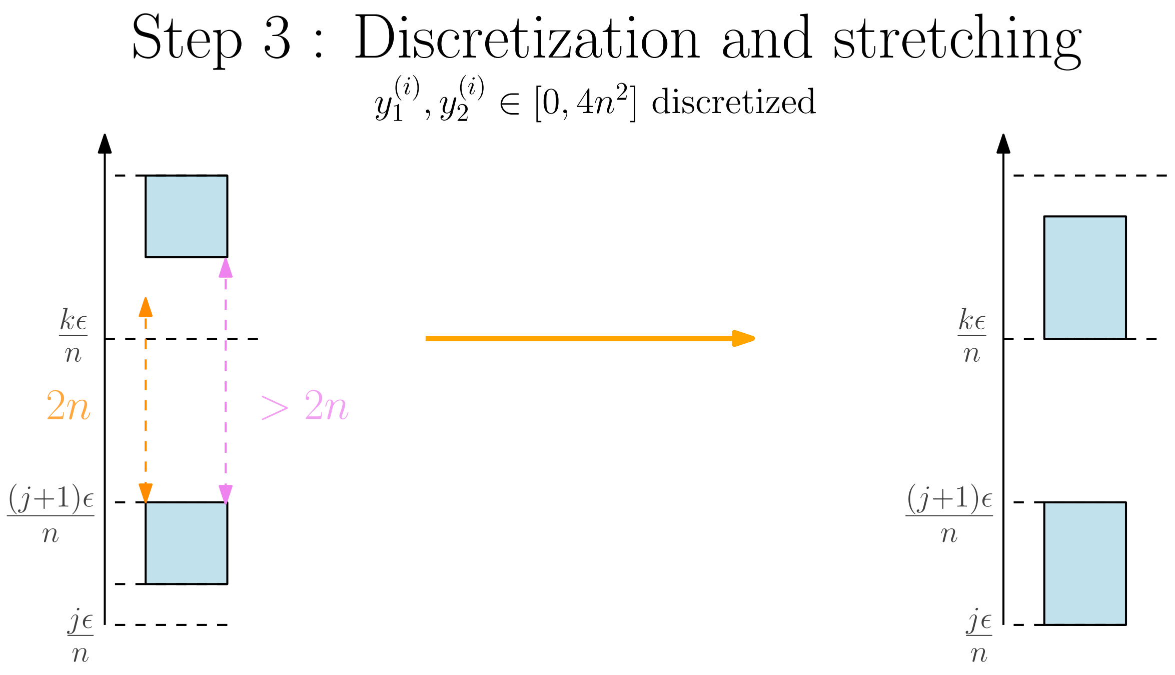

Step 3 : Discretization and stretching
$y_1^{(i)}, y_2^{(i)} \in [0, 4n^2]$ discretized
$\frac{k\epsilon}{n}$
$2n$
$> 2n$
$\frac{(j+1)\epsilon}{n}$
$\frac{j\epsilon}{n}$
$\frac{k\epsilon}{n}$
$\frac{(j+1)\epsilon}{n}$
$\frac{j\epsilon}{n}$

Figure 4: Discretization and stretching step

Step 4 : Division Now we consider our problem after the 3 first steps and OPT' is the optimal solution for this preprocessed instance with $c(OPT') \leq (1 + O(\epsilon))c(OPT)$. Now, we consider any horizontal segment in OPT' that is longer than $1/\epsilon$. From the left endpoint, we divide the segment into consecutive smaller segments of length $1/\epsilon - 2$ until the last one, that has a length equal to or smaller than $1/\epsilon - 2$. Then, we extend each of these smaller segments such that it completely stabs each rectangle that it intersects. This extension is at most 2 times the total maximum width of a rectangle, i.e. 2 by segment. This step is shown in Figure 5.

Eventually, the division only implies a fractional increase of $1 + O(\epsilon)$ for the approximation ratio. Indeed, if we consider all segment s_j in OPT' of length $k_j/\epsilon + \delta$ with $0 < \delta < 1/\epsilon$ and $k_j \geq 1$, the corresponding total increase is :

$$
\begin{aligned}
\frac{\sum_{j=1}^{r} k_j/\epsilon + \delta + 2(k_j+1)}{\sum_{j=1}^{r} k_j/\epsilon + \delta} &= 1 + \frac{2\sum_{j=1}^{r}(k_j+1)}{\sum_{j=1}^{r} k_j/\epsilon + \delta} \\
&\leq 1 + 2\epsilon \frac{\sum_{j=1}^{r}(k_j+1)}{\sum_{j=1}^{r} k_j} \\
&\leq 1 + 2\epsilon \frac{r + \sum_{j=1}^{r} k_j}{\sum_{j=1}^{r} k_j} \\
&\leq 1 + 2\epsilon \left(1 + \frac{r}{\sum_{j=1}^{r} k_j}\right) \\
&\leq 1 + 2\epsilon \left(1 + \frac{r}{\sum_{j=1}^{r} 1}\right) = 1 + 4\epsilon = 1 + O(\epsilon)
\end{aligned}
$$

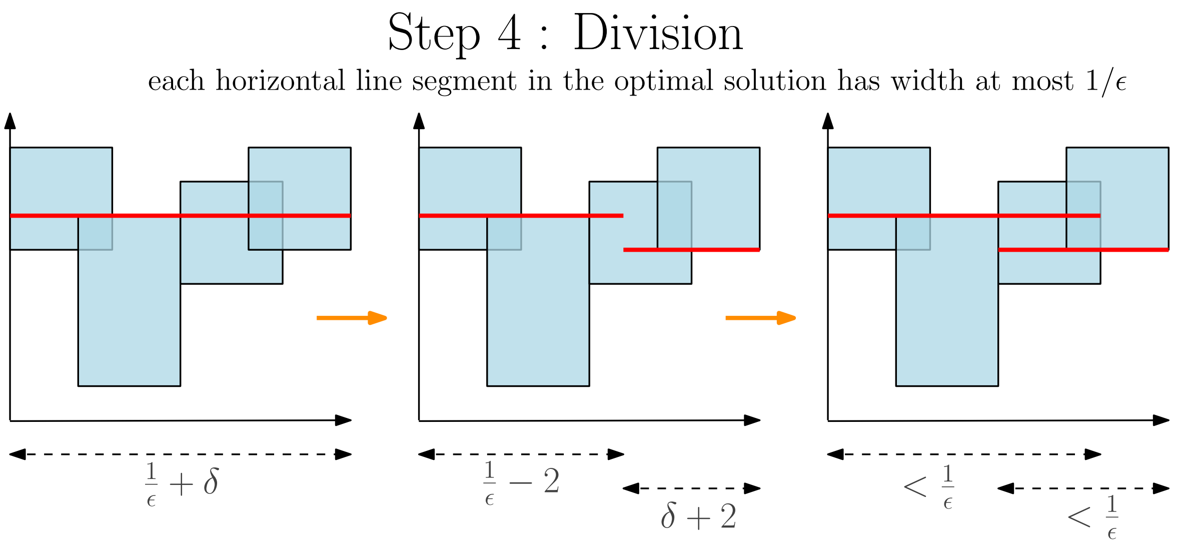

Step 4 : Division
each horizontal line segment in the optimal solution has width at most $1/\epsilon$
$\frac{1}{\epsilon} + \delta$
$\frac{1}{\epsilon} - 2$
$\delta + 2$
$< \frac{1}{\epsilon}$
$< \frac{1}{\epsilon}$

Figure 5: Division step

In the remaining description of the PTAS, we will assume to be working with pre-processed inputs, i.e. instances that satisfy the properties described in theorem 1.1.

2 Description of the dynamic program

In this section, we will present the algorithm itself. This algorithm is based on a dynamic program. Each division of the problem is defined by a cell $DP(E, \mathcal{L})$ where :

- E is a spatial division, i.e. E is a rectangle area in $[0, n] \times [0, 4n^2]$ with discretized coordinates which contains the rectangles to consider.
- $\mathcal{L}$ is a set of at most $3\epsilon^{-3}$ horizontal and vertical segments that are in E and have discretized coordinates.

An example of such a division is represented in Figure 6.

The DP-cell $DP(E, \mathcal{L})$ stores $SOL(E, \mathcal{L})$ such that the solution of the stabbing problem for the rectangles in E is $SOL(E, \mathcal{L}) \cup \mathcal{L}$. This means that the $DP(E, \mathcal{L})$ represents the subproblem of stabbing all the rectangles in E that are not already stabbed by segments in $\mathcal{L}$.

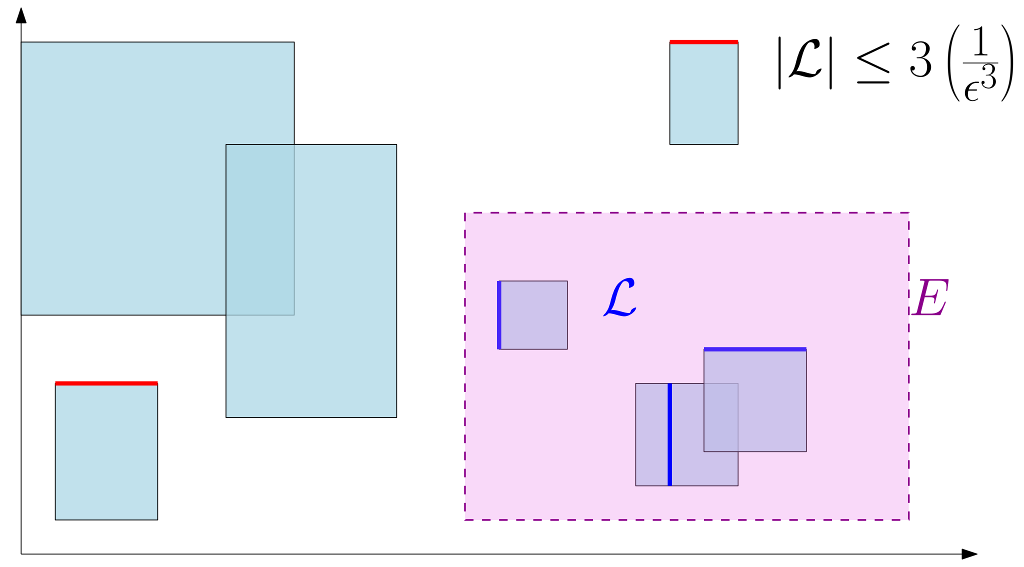


Figure 6: Example of a DP-cell

The starting problem is given by the original input and is represented by the DP-cell $DP([0,n] \times [0,4n^2], \emptyset)$. The basic case is when the segments in $\mathcal{L}$ stab each rectangle in E and thus the solution is $SOL(E,\mathcal{L}) = \emptyset$.

To define the other operations, we need the following definition :

Definition 2.1. *Let E be any spatial division and $\mathcal{L}$ any set of segments defined as before. The intersection between E and $\mathcal{L}$ is defined by :*

$$\mathcal{L} \cap E := \{\ell \cap E \mid \ell \in \mathcal{L} \text{ and } \ell \cap E \neq \emptyset\}.$$

We define an operation to divide the problem into two subproblems and call it the *trivial operation*. Let $\ell \in \mathcal{L}$ such that ℓ completely cut across E and let E_1 and E_2 be the division of the space on the left and right of ℓ that contains only rectangles not already stabbed by ℓ. Then we define the solution to be $SOL(E,\mathcal{L}) = SOL(E_1, \mathcal{L} \cap E_1) \cup SOL(E_2, \mathcal{L} \cap E_2) \cup \{\ell\}$. An example of this operation is shown in Figure 7. If there is more than one such ℓ then we pick according to an arbitrary global tie-breaking rule.

Now we define two operations that produce a set of candidate solutions. We will take $SOL(E,\mathcal{L})$ to be the solution of minimum cost among all the candidates.

- The *add operation* consists in adding a set of segments to the solution. Concretely, let $\mathcal{L}'$ be a set of horizontal or vertical segments in E with discretized coordinates such that $|\mathcal{L} \cup \mathcal{L}'| \leq 3\epsilon^{-3}$. We define the candidate solution of the DP-cell for $\mathcal{L}'$ to be $SOL(E,\mathcal{L}) = \mathcal{L}' \cup SOL(E, \mathcal{L} \cup \mathcal{L}')$.

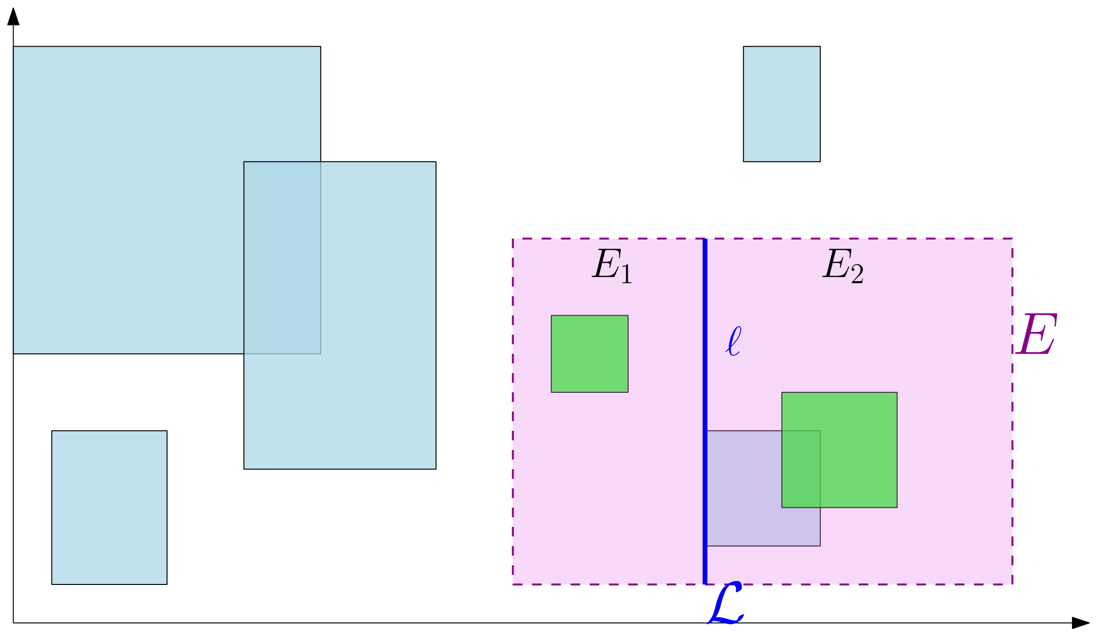


Figure 7: Example of a trivial operation

- The *line operation* is used to divide the problem into two subproblems. Let ℓ be a line with discretized coordinates such that ℓ completely cut across E and let E_1 and E_2 be the division of the space on the left and right of ℓ that contains only rectangles not already stabbed by ℓ and R_ℓ the set of these rectangles. Consider an $O(1)-$algorithm in polynomial time for the stabbing problem on R_ℓ (for example the algorithm in Chan *et al.* (2018)) and let $S(R_\ell)$ be the approximate solution output by it. We define the candidate solution of the DP-cell for ℓ to be

$$SOL(E,\mathcal{L}) = SOL(E_1,\mathcal{L}\cap E_1)\cup SOL(E_2,\mathcal{L}\cap E_2)\cup S(R_\ell).$$

  An example of this operation is shown in Figure 8

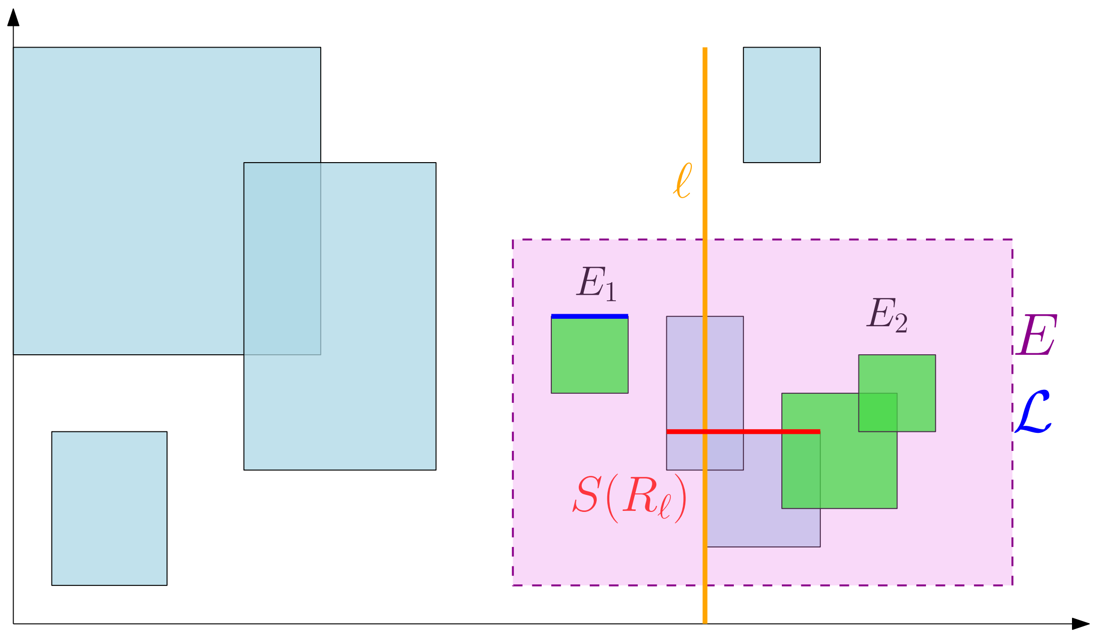


Figure 8: Example of a line operation

The entire algorithm is resumed below in Algorithm 1.

Algorithm 1 $DP(E, \mathcal{L})$

```
if $\mathcal{L}$ stabs all rectangles in $E$ then
    return $\emptyset$
else if $\exists \ell \in \mathcal{L}$ such that $\ell$ completely cut across $E$ then
    return $DP(E, \mathcal{L}) = DP(E_1, \mathcal{L} \cap E_1) \cup DP(E_2, \mathcal{L} \cap E_2) \cup \{\ell\}$
end if
$Add$=argmin$\{c(\mathcal{L}' \cup DP(E, \mathcal{L} \cup \mathcal{L}')) \mid \mathcal{L}' \subset E$ and $|\mathcal{L} \cup \mathcal{L}'| \leq 3\epsilon^{-3}\}$
$Line$=argmin$\{c(S(R_\ell) \cup DP(E_1, \mathcal{L} \cap E_1) \cup DP(E_2, \mathcal{L} \cap E_2)) \mid \ell$ has discretized coordinates and completely cut across $E\}$
return argmin$\{c(Add), c(Line)\}$
```

3 Proofs

In this section, we will present the proof that the algorithm presented in the previous section is a PTAS. First, we will show that this algorithm runs in polynomial time. After, we will prove that it gives a $(1+O(\epsilon))$-approximation. For this, we need to define what a DP decision tree is and construct one. Then we will prove that the cost of the solution given by the DP decision tree we constructed is inferior to $(1 + O(\epsilon))OPT$.

3.1 Proof of the polynomial time

Theorem 3.1. *The algorithm 1 has a running time of* $(n/\epsilon)^{O(1/\epsilon^3)}$.

Proof. As this algorithm is a dynamic program, the running time is the number of DP-cell times the time to run one DP-cell.

The number of DP-cells is given by the number of rectangles E and sets $\mathcal{L}$ that we consider. A rectangle E is defined by its two corners vertices which are themselves defined by a discretized x-coordinate and a discretized y-coordinate, thus :

$$\#E = \binom{\text{number of corner vertices}}{2} = \binom{\frac{n}{\epsilon/n} \times \frac{4n^2}{\epsilon/n}}{2} = O(n^{10}/\epsilon^4).$$

A set of segments $\mathcal{L}$ must have less than $3\epsilon^{-3}$ segments in it, thus:

$$\#\mathcal{L} = \binom{\#\text{Segments}}{3\epsilon^{-3}}.$$

A vertical/horizontal segment is defined by two vertical/horizontal discretized coordinates and one horizontal/vertical discretized coordinate, thus :

$$\#\text{Segments} = \#\text{vertical} + \#\text{horizontal} = \frac{n}{\epsilon/n}\binom{\frac{4n^2}{\epsilon/n}}{2} + \frac{4n^2}{\epsilon/n}\binom{\frac{n}{\epsilon/n}}{2} = O(n^8/\epsilon^3).$$

Combining all of these, we have that $\#\mathcal{L} = (n/\epsilon)^{O(1/\epsilon^3)}$ and

$$\#\text{DP-cell} = \#E \times \#\mathcal{L} = \left(\frac{n}{\epsilon}\right)^{O(1/\epsilon^3)}.$$

The time to run a DP-cell only depends on the time to consider all candidates for the line and add operations to take the minimum, i.e. :

$$\text{Time to run a DP-cell } = \#\text{Line operations } \times \text{time to find } S(R_\ell) + \#\text{Add operations}.$$

When defining the line operations, we chose a polynomial algorithm to find $S(R_\ell)$. The number of line operations is the number of lines considered. A line is defined by a horizontal and a vertical discretized coordinate, thus :

$$\#\text{Line operations} = \#\text{Lines} = \frac{n}{n/\epsilon} + \frac{4n^2}{n/\epsilon} = O(n^3/\epsilon).$$

The number of add operations is the number of possible sets of segments that can be added and thus the number of set $\mathcal{L}$ that we computed before. Finally :

$$\text{Time to run a DP-cell} = O(n^3/\epsilon) \times \textit{polynomial} + (n/\epsilon)^{O(1/\epsilon^3)} = (n/\epsilon)^{O(1/\epsilon^3)}.$$

Eventually, the running time of this algorithm is in :

$$(n/\epsilon)^{O(1/\epsilon^3)} \times (n/\epsilon)^{O(1/\epsilon^3)} = (n/\epsilon)^{O(1/\epsilon^3)}$$

□

3.2 Construction of a DP decision tree

Definition 3.1. *A DP decision tree is defined as follows:*

- *Each node corresponds to a cell $DP(S, \mathcal{L})$ and its corresponding solution $SOL(S, \mathcal{L})$.*
- *The root is $DP([0, n] \times [0, 4n^2], \emptyset)$.*
- *A leaf correspond to a cell such that $DP(S, \mathcal{L}) = \emptyset$.*
- *A node corresponding to $DP(E, \mathcal{L})$ has one child corresponding to $DP(E, \mathcal{L} \cup \mathcal{L}')$ if the edge between the two nodes corresponds to an add operation with respect to $\mathcal{L}'$.*
- *A node corresponding to $DP(E, \mathcal{L})$ has two children corresponding to $DP(E_1, E_1 \cap \mathcal{L})$ and $DP(E_2, E_2 \cap \mathcal{L})$ if the children correspond to two subproblems E_1, E_2 coming from a trivial operation or a line operation.*

The goal is to construct a DP-decision tree for which $c(SOL(S, \mathcal{L})) \leq (1 + \epsilon)OPT$.

To explain the construction, some intermediate definitions are needed.

Let $a \in \mathbb{N}_0$ random. We create a hierarchical grid of vertical lines by taking for each level $j \in \mathbb{N}_0$ the grid line $\{a + k\epsilon^{j-2}\} \times \mathbb{R}$ for all $k \in \mathbb{N}$. Note that each grid line of level j is included in all the sets of grid line of level $j + t$ for $t \geq 0$.

Definition 3.2. *In the context just described, we define :*

- *A segment $s \in OPT$ is of level j if its length is in $(\epsilon^j, \epsilon^{j-1}]$.*
- *A horizontal segment of some level j is well-aligned if both its left and right coordinates are of the form $a + k\epsilon^{j+1}$ for some $k \in \mathbb{N}$, i.e. they lie on a grid line of level $j + 3$ (or lower).*

- *A vertical segment of some level j is well-aligned if both its top and bottom coordinates are of the form $k\epsilon^{j+1}$ for some $k \in \mathbb{N}$, i.e. they lie on an imaginary horizontal grid line of level $j+3$ or lower.*

An example of a grid and some well-aligned segments of different levels is shown in Figure 9.

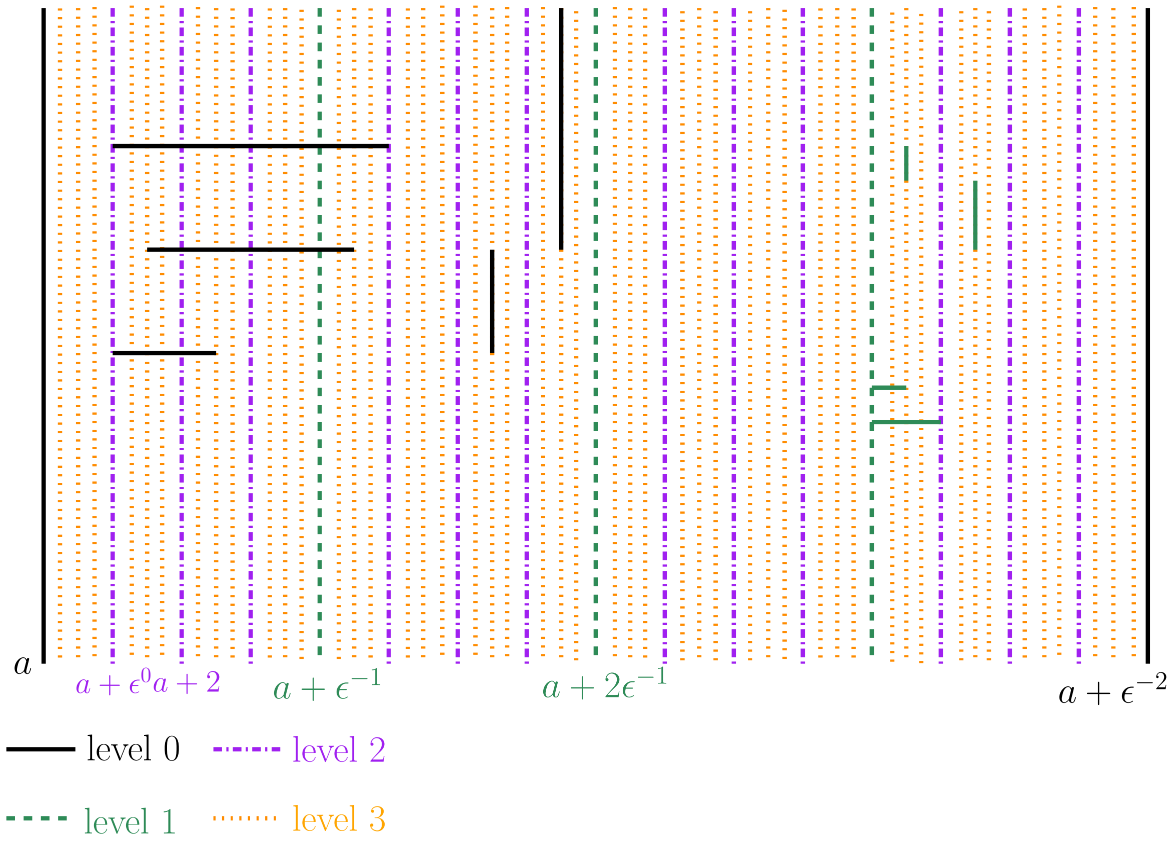


Figure 9: Example of a grid with well-aligned segments of different levels

Lemma 3.1. *(Lemma 3 in Khan* et al. *(2021)) By losing a factor* $1+O(\epsilon)$*, we can assume that each line segment* $s \in OPT$ *is well-aligned.*

Proof. Let s be an arbitrary segment in OPT of length $l \in (\epsilon^j, \epsilon^{j-1}]$. The relevant coordinates (top and bottom for a vertical line segment or right and left for a horizontal line segment) are each between some grid lines of level $j+3$. As the grid line of level $j+3$ are ϵ^{j+1} apart, to make s well aligned we would need to extend it by at most ϵ^{j+1} on each side. Thus, the new length l' of s is thus bounded as follows:

$$l' \leq l + 2\epsilon^{j+1} = l\left(1+\frac{2\epsilon^{j+1}}{l}\right) < l\left(1+\frac{2\epsilon^{j+1}}{\epsilon^j}\right) = (1+2\epsilon)l.$$

Therefore, the total length is bounded as such :

$$\sum_{s\in OPT} l' \leq \sum_{s\in OPT} (1+2\epsilon)l = (1+2\epsilon)OPT.$$

□

We can now construct the DP-decision tree T using the definitions given above. The construction is done recursively. At each step, we consider each subproblem given by the previous step. This is done as follows.

Level 0 : The root correspond to $DP([0,n] \times [0,4n^2], \emptyset)$.

1. Apply the line operation for each vertical grid line of level 0.
2. If there are more than ϵ^{-3} segments from OPT of level 0 in the subproblem, then order their endpoints non-decreasingly by their y-coordinates. Let $p_1, \dots, p_k$ be these endpoints ordered. For each $k' \leq k\epsilon^3$, apply the horizontal line operation on the line containing the endpoint p_{k'/ϵ^3}.
3. Apply the line operation on each line that contains a vertical segment from OPT that completely cuts across the subproblem space.
4. Let's call the resulting subproblem $DP(E', \emptyset)$. Apply the add operation to the segments from OPT of level 0 that intersect E' i.e. to the set
$$\mathcal{L}' := \{\ell \cap E' \mid \ell \in \text{OPT and } \ell \cap E' \neq \emptyset \text{ and } \ell \text{ is of level } 0\}.$$
5. Apply the trivial operation as long as it is possible.

Lemma 3.2. *Let $DP(E, \emptyset)$ be one of the subproblems after applying step 2. There are at most ϵ^{-3} segments from OPT of level 0 that have an endpoint inside E. Thus, at most ϵ^{-3} segments are added in step 4.*

Proof. In step 2, we apply the line operation to each endpoint that has an order that is a multiple of ϵ^{-3}. Thus, each resulting subproblem has at most ϵ^{-3} endpoints coming from segments from OPT of level 0. As a segment has two endpoints that may not be in the same subproblem, there are at most ϵ^{-3} segments from OPT of level 0 that have an endpoint inside E (if each segment had both endpoints in E then it would be at most $\epsilon^{-3}/2$).

As the following steps only do more divisions, the number of segments from OPT of level 0 in a subproblem can only become smaller than ϵ^{-3}. Thus, the set added by the add operation has less than ϵ^{-3} segments. □

For each $j \in \mathbb{N}^*$, assume by induction that each leaf of the current tree corresponds to a subproblem $DP(E, \mathcal{L})$ such that $\{\ell \cap E \mid \ell \in \text{OPT and } \ell \cap E \neq \emptyset \text{ and } \ell \text{ is of level } j' < j\} \subseteq \mathcal{L}$. We start considering one of these leaves and, as before, at each step we consider each subproblem given by the previous step.

Level j :

1. Apply the line operation for each vertical grid line of level j.
2. If there are more than ϵ^{-3} segments from OPT of level j in the subproblem, then order their endpoints non-decreasingly by their y-coordinates. Let $p_1, \dots, p_k$ be these endpoints ordered. For each $k' \leq k\epsilon^3$, apply the horizontal line operation on the line containing the endpoint p_{k'/ϵ^3}.
3. If there is a vertical segment in $\mathcal{L}$ of level $j' < j - 2$ with an endpoint p inside the subproblem space, then apply the line operation on the horizontal line containing p.
4. Apply the line operation on each line that contains a vertical segment from OPT that completely cuts across the subproblem space.
5. Apply the trivial operation as long as it is possible.

6. Let's call the resulting subproblem $DP(E", \mathcal{L})$. Apply the add operation to the segments from OPT of level j that intersects E" i.e. to the set

$$\mathcal{L}' := \{\ell \cap E" \mid \ell \in \text{OPT and } \ell \cap E" \neq \emptyset \text{ and } \ell \text{ is of level j}\}.$$

Lemma 3.3. *Let $DP(E', \mathcal{L})$ be one of the subproblems after applying step 2. There are at most ϵ^{-3} segments from OPT of level j that have an endpoint inside E. Thus, at most ϵ^{-3} segments are added in step 6.*

Proof. It's the same proof as Lemma 3.2 replacing 0 by any $j \in \mathbb{N}^*$. □

Theorem 3.2. *The constructed tree T is a DP-decision tree.*

Proof. Only a sketch of the proof will be given, for the formal proof see Lemma 6 in Khan *et al.* (2021).

The main point to prove is that the set of segments $\mathcal{L}$ for each subproblem contains at most $3\epsilon^{-3}$ segments. The proof is given by induction using Lemma 3.2 and 3.3. The key idea is that at the end of the steps of level j, the set $\mathcal{L}$ contains segments only from level $j, j-1$ and $j-2$. This is assured by the trivial operations from step 5 and the assumption that all segments in OPT are well-aligned. An insight on this fact can be seen for the horizontal segments in Figure 9 by looking at the segment of level 0 and the grid lines of level 3. □

3.3 Proof of the approximation

In this section, we will show that the cost corresponding to T is at most $(1 + O(\epsilon))OPT$.

Lemma 3.4. *(Lemma 7 in Khan* et al. *(2021)) There is a choice for the offset a such that the solution $SOL([0, n] \times [0, 4n^2], \emptyset)$ in T has a cost of at most* $(1 + O(\epsilon))OPT$.

Proof. The add operations are applied on segments from OPT and have a total cost of $c(OPT)$. The trivial operations are done on already added segments and thus have no additional cost. We will prove that for a random offset a, the approximation ratio for the line operations is $O(\epsilon \cdot OPT)$.

Cost of the horizontal line operations : Let OPT_{cell} be the set of segments from OPT with at least one endpoint in the *cell* and H the set of all the horizontal lines used in a line operation. Let consider a horizontal line operation of some level $j > 0$ along the line ℓ. This operation was done either in step 2 or in step 3.

- Step 2 : Two cells of width at most ϵ^{-2}, such that at least one of them contained at most ϵ^{-3} endpoints of segments from OPT of level j, are created. The minimal length of a segment of level j is ϵ^j. Thus, the cost of segments of level j for the cell is such that $c(OPT_{cell}) \geq \epsilon^{-3} \cdot \epsilon^j = \epsilon^{j-3}$ (the cost for the other cell is either counted for another line operation or has the same bound, as a consequence of Lemma 3.3).

- Step 3 : Two cells of width at most ϵ^{-2}, such that at least one of them contained one vertical segment from OPT of level $j' < j-2$, are created. Henceforth, the cost of the operation is the length of this segment and $c(OPT_{cell}) \geq \epsilon^{j-3}$.

In both cases, a segment of width ϵ^{j-2}(the width of the cell) is enough to stab all rectangles stabbed by ℓ. As we charge the cost of this segment to its two endpoints, we have:

$$c(H) \leq \sum_{cell} 2 \cdot \epsilon^{j-2} \quad \text{and} \quad c(OPT) = \sum_{cell} c(OPT_{cell}) \geq \sum_{cell} \epsilon^{j-3}.$$

Thus, the approximation ratio for the horizontal lines is :

$$\frac{c(H)}{c(OPT)} \leq \frac{\sum_{cell} 2 \cdot \epsilon^{j-2}}{\sum_{cell} \epsilon^{j-3}} \leq 2\epsilon.$$

Cost of the vertical line operations : The line operations, done in step 3 for level 0 and in step 4 for level j, do not add any cost as they are done on segments from OPT that completely cut across the relevant cell. Thus, the additional cost comes from the line operations done along the grid lines in step 1.

Let OPT_j be the cost of segments of level j in OPT. We first want to find the probability for a segment of level j to be intersected or *close* to a grid line (will be formally defined below). Consider a segment $\ell \in OPT_j$, either ℓ is horizontal or vertical :

- ℓ is horizontal : Let I_ℓ be the indicator variable representing the event that a grid line of level j or smaller intersects ℓ ($I_\ell = 0$ for vertical segments). By definition and preprocessing step 4, ℓ has a length $l \leq \epsilon^{j-1}$. If we take a random a, we have that the expectation of I_ℓ is the area where ℓ is cut by a grid line, if its right endpoint lies in it, over the entire possible area for its right endpoint to be. Thus :

$$\mathbb{E}[I_\ell] = \mathbb{P}[I_\ell = 1] = \frac{l \cdot \#\text{grid lines}}{\epsilon^{j-2} \cdot \#\text{grid lines}} \leq \frac{\epsilon^{j-1}}{\epsilon^{j-2}} = \epsilon.$$

- ℓ is vertical : Let J_ℓ be the indicator variable representing the event that a grid line of level j or smaller intersects the rectangle stabbed by ℓ ($J_\ell = 0$ for horizontal segments). By definition, the length of ℓ is such that $l \leq \epsilon^{j-1}$ for $j > 0$. For the rectangle stabbed by ℓ, the dimension must satisfy $w_i \leq h_i \leq l \leq \epsilon^{j-1}$. This means that to stab such a rectangle, ℓ has to lie *close to*, i.e. within $\pm\epsilon^{j-1}$ of the vertical grid line. Thus, the area where ℓ is close to a grid line has a size of $2 \cdot \epsilon^{j-1}$. As for the horizontal case, the expectation is :

$$\mathbb{E}[J_\ell] = \mathbb{P}[J_\ell = 1] \leq \frac{2\epsilon^{j-1}}{\epsilon^{j-2}} = 2\epsilon.$$

  For level 0, even though the vertical segment can be very long, the maximum width of a rectangle is at most 1. So ℓ has to lie within ± 1 of the grid line and the area where ℓ is close to a grid line of level 0 has a size of 2. Thus, as $\epsilon < 1/3$:

$$\mathbb{E}[J_\ell] \leq \frac{2}{\epsilon^{-2}} = 2\epsilon^{-2} \leq 2\epsilon.$$

We can now upper bound the total expected cost of segments in OPT (the cost for each segment ℓ is its length l_ℓ) intersected by vertical line operations as :

$$\begin{aligned}
OPT_{intersected} &= \mathbb{E}\left[\sum_j \sum_{\ell \in OPT_j} (I_\ell + J_\ell) \cdot l_\ell\right] \\
&= \sum_j \sum_{\ell \in OPT_j} \mathbb{E}\left[(I_\ell + J_\ell) \cdot l_\ell\right] \\
&= \sum_j \sum_{\ell \in OPT_j} l_\ell \cdot \mathbb{E}\left[(I_\ell + J_\ell)\right] \\
&\leq \sum_j \sum_{\ell \in OPT_j} l_\ell \cdot (\epsilon + 2\epsilon) \\
&= 3\epsilon \cdot \sum_j \sum_{\ell \in OPT_j} l_\ell \\
&= 3\epsilon \cdot OPT
\end{aligned}$$

Henceforth, there exists an offset a such that the cost of segments in OPT intersected by vertical line operations is at most $3\epsilon \cdot OPT$. Now, for the line operation, we used a constant-approximation algorithm to stab all the rectangles stabbed by ℓ. Let α be this constant. The total cost of the vertical line operations is thus $3\alpha \cdot \epsilon \cdot$ OPT.

Eventually, the total approximation ratio for the line operations is $2\epsilon + 3\alpha \cdot \epsilon = O(\epsilon)$

□

Now we prove the main theorem and its corollary for the horizontal stabbing problem.

Theorem 3.3. *(Theorem 8 in Khan* et al. *(2021)) There is a* $(1+\epsilon)-$*approximation algorithm for the general stabbing problem with a running time of* $(n/\epsilon)^{O(1/\epsilon^3)}$, *assuming that* $h_i \geq w_i$ *for each rectangle* R_i *in the input.*

Proof. We have shown in Theorem 3.1 that the algorithm described in section 4 is in polynomial time $(n/\epsilon)^{O(1/\epsilon^3)}$. Moreover, we proved in Theorem 1.1 that the preprocessing steps costs a factor $(1 + O(\epsilon))$. Eventually, we constructed a correct DP-decision tree in section 5.2 and shown in section 5.3 that its solution has a cost of $(1 + O(\epsilon))OPT$. As the algorithm in section 4 creates a decision tree of minimal cost, it will be inferior to $(1 + O(\epsilon))OPT$. □

Corollary 3.1. *(Corollary 11 in Khan* et al. *(2021)) There is a* $(1 + \epsilon)-$*approximation algorithm for the horizontal stabbing problem with a running time of* $(n/\epsilon)^{O(1/\epsilon^3)}$.

Proof. Use the same algorithm as in Theorem 3.3 after stretching vertically all the rectangles in the input such that it becomes very costly to stab any rectangle vertically. □

Part II
An 8-approximation algorithm for the horizontal stabbing problem

In this part, we consider the horizontal stabbing problem, i.e. the segments in the solution can only be horizontal. We will use interchangeably the name of a set and its cost (for example, using OPT instead of $c(OPT)$ for the cost). The goal of this part is to present an algorithm in $O(n^4)$ that gives an 8-approximation, i.e. $c(ALG(R)) \leq 8 \cdot c(OPT(R))$ for any instance R. This algorithm was first described in Eisenbrand *et al.* (2021).

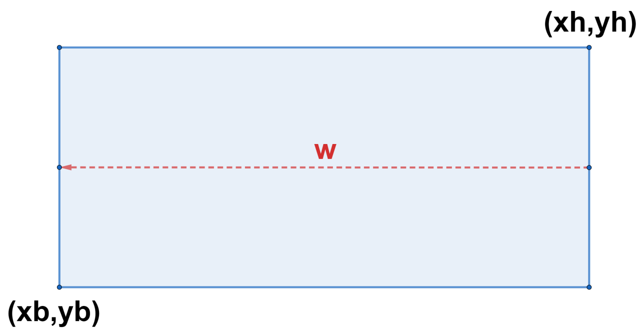
The first section will present the notations used in this part. The second one explains how, by losing a 4-approximation factor, we can transform the input into a laminar one. It also presents a method to get a feasible solution for the original input from the solution of the laminar input by losing another 2 factor. Then, in section 3, we present an algorithm that gives the optimal solution for a laminar input in $O(n^4)$. Putting these results together, we get the 8-approximation algorithm for the general instance. The fourth section is a proof that the approximation bound is tight and presents some improvement that can be done to the implementation. Eventually, the last section presents the code of the implementation of this algorithm.

4 Notations and representations

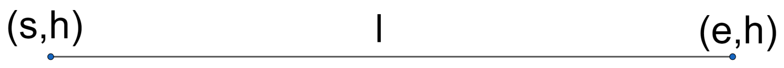
The input is a set of n axis-aligned rectangles $\{R_i\}_{i=1}^n$ with $n \geq 1$. Each rectangle is defined by its width w, its bottom-left corner (x_b, y_b) and its top-right corner (x_h, y_h).

A segment in a solution must be horizontal and aligned with the x-axis. It is defined by its y-coordinate h, two x-coordinates s and e with $s < e$ and its length $l = e - s$. Without loss of generality, we can suppose that a segment starts and ends at extremities of some rectangles and its y-coordinate corresponds to the top of some rectangle. This can be supposed by the nature of the stabbing problem, where we want a set of segments with minimum length stabbing all the rectangles. Indeed, it holds that for any solution, we can find an equivalent one with all y-coordinates aligned with the top of a rectangle.

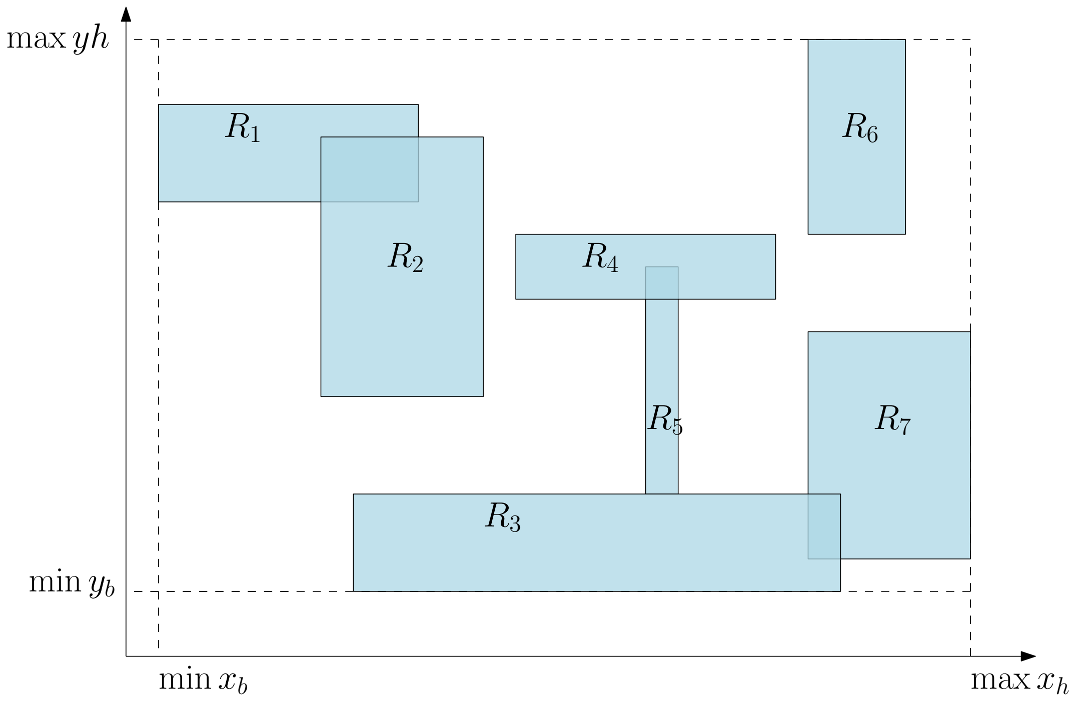
An input of the problem, called an *Ensemble*, is represented by a list of n rectangles $Rects$, with *maxRect* being the rectangle with the maximum width and two x-coordinates and two y-coordinates that are the minimum and maximum coordinates from the rectangles in $Rects$. These three definitions are pictured in Figure 10.



(a) Representation of a Rectangle



(b) Representation of a Segment



Rects=$[R_1, R_2, R_3, R_4, R_5, R_6, R_7]$
n=7
MaxRect=R_3=Rectangle with maximum width

(c) Representation of an *Ensemble*

Figure 10: Representations

5 Transformation to laminar

Definition 5.1. *A set of axis-aligned rectangles is laminar if, for any pair of rectangles in it, their projection on the x-axis are either disjoint or one is contained in the other.*

An example of a laminar set of rectangles is shown in Figure 11.

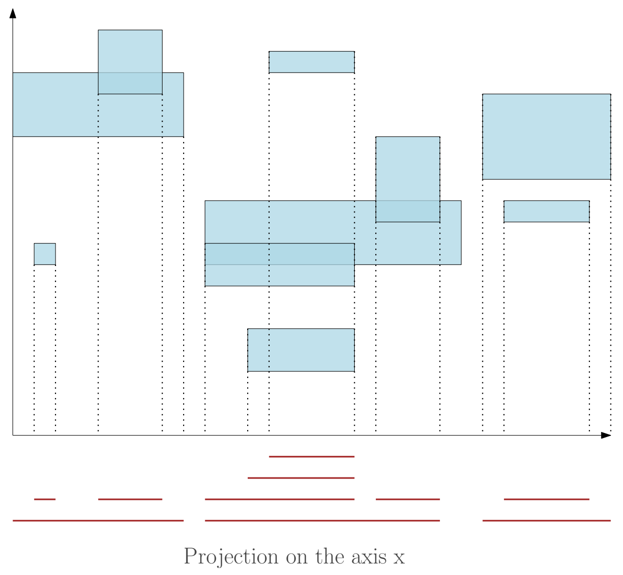


Figure 11: Example of a laminar instance

Now, we will explain a procedure to transform any general input to a laminar one.

Procedure: We transform each rectangle R_i in the original input with width w as follows :

1. Round its width to the closest upper power of 2 i.e. let its new width be $w' = 2^{\lceil \log_2(w) \rceil}$.
2. Translate its left x-coordinate to the closest multiple of its new width on the left, i.e. let its new x_b be $x'_b = (x_b//w')w'$.

This procedure is resumed in Figure 12 and shown on an example in Figure 13.

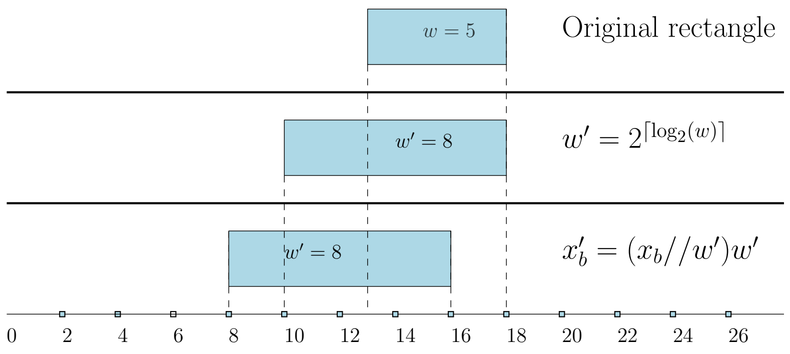


Figure 12: Procedure on one rectangle

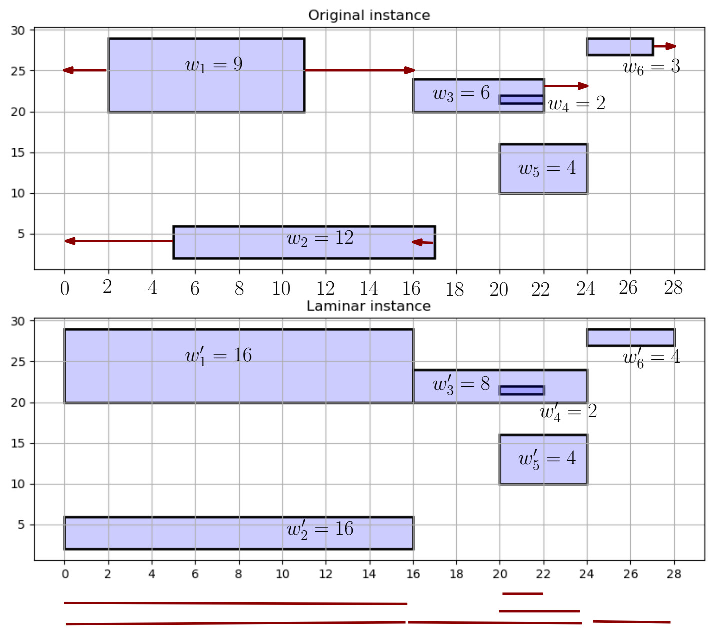


Figure 13: Example of the procedure on an input

The transformation presented introduced a factor 4 in the approximation ratio.

Lemma 5.1. *Let R be the original instance and L the laminar instance created from R by the procedure presented before. We have that $OPT(L) \leq 4OPT(R)$.*

Proof. For any feasible solution $FEAS(L)$, we have $FEAS(L) \geq OPT(L)$. Thus, we need to prove that there exists a feasible solution that satisfies $FEAS(L) \leq 4OPT(R)$.

Let us consider SOL, an optimal solution for R. For each segment $\ell = [s, e]$ of length $l(\ell) \in]2^{t-1}, 2^t]$ in SOL, we round s down and e up to the next multiple of 2^t to get a new segment ℓ'. Let us call SOL' this modified solution. We can prove that SOL' is a feasible solution for L with a total length inferior to $4 \cdot OPT(R)$.

First, we prove that SOL'$\leq 4 \cdot OPT(R)$. Let us consider $\ell' \in$ SOL' with corresponding segment $\ell \in$ SOL. There exist $t > 0$ such that $2^{t-1} < l(\ell) \leq 2^t$. Since we extended the extremities of ℓ to get ℓ' to the next multiple of 2^t, we added at most 2^t to the length (as $2^{t-1} < l(\ell) \leq 2^t$). Thus:

$$l(\ell') \leq 2^t + l(\ell) \leq 2 \times 2^t = 4 \times 2^{t-1} < 4 \times l(\ell),$$

and

$$\text{SOL'} = \sum_{\ell' \in \text{SOL'}} l(\ell') \leq \sum_{\ell \in \text{SOL}} 4 \cdot l(\ell) = 4 \cdot \text{SOL} = 4 \cdot OPT.$$

Now, we need to prove that SOL' is feasible for L. We will prove that any $\ell' \in$ SOL' covers all the rectangles in L corresponding to the rectangles in R stabbed by ℓ. This implies the wanted result.

Consider any $\ell' \in$ SOL' and an arbitrary rectangle r that was stabbed by the corresponding $\ell \in$ SOL with laminar counterpart r'. Let x_b^r (resp. $x_b^{r'}$) be the left x-coordinate of r (resp. r'), x_h^r (resp. $x_h^{r'}$) their right x-coordinate and s (resp. s') and e (resp. e') the left and right x-coordinate of ℓ (resp. ℓ'). As r was stabbed by ℓ, its width is $w \leq l(\ell) \leq 2^t$ and the extremities of ℓ are such that $s \leq x_b^r$ and $e \geq x_h^r$. The new segment ℓ' stabs r' if $s' \leq x_b^{r'}$ and $e' \geq x_h^{r'}$.

The first step of the laminar transformation implies that the width of r' is such that $w' \leq 2^t$ and the second ones implies that $x_b^{r'} \geq k \cdot 2^t$ with $k = \max\{z \in \mathbb{Z} \mid z \cdot 2^t \leq x_b^r\}$ i.e. the left x-coordinate of r' is bigger than the closest multiple of 2^t that is before r. This is due to the fact that the widths are rounded to powers of 2 and thus a bigger width will be divided by any smaller one. Moreover, by definition, s is down rounded to the closest multiple of 2^t i.e. $s' = k' \cdot 2^t$ with $k = \max\{z \in \mathbb{Z} \mid z \cdot 2^t \leq s\}$. As $s \leq x_b^r$, the factor $k' \leq k$ thus $s' \leq x_b^{r'}$.

The proof that $e' \geq x_h^{r'}$ is analogous to the proof for $s' \leq x_b^{r'}$. □

After transforming our input R to a laminar one L, we will get, with the algorithm presented in the next section, the optimal solution for the laminar instance $OPT(L)$. We need to transform it to make it feasible for the original input. For this, we double the length of each segment on the right. Let $ALG(R)$ be this final solution.

Lemma 5.2. *Using the notations described before, we have that $ALG(R)$ is feasible for R and $ALG(R) = 2 \cdot OPT(L)$.*

Proof. In step 2 of the procedure, we moved each rectangle in the original input by at most its new width w' on the left. Since a segment stabbing a rectangle has a length equal or bigger to its width, by multiplying it by two on the right, we are sure that it will cover the original rectangle. As we multiply each segment by two, we multiply the overall cost of the solution by two. □

Eventually, doing these two transformations introduced a factor 8 in the approximation ratio.

Theorem 5.1. *Supposing that we have an algorithm giving the optimal solution for the stabbing problem on a laminar instance, we have, using the previous notations, that $ALG(R) \leq 8 \cdot OPT(R)$.*

Proof. Lemma 5.2 states that $ALG(R) = 2 \cdot OPT(L)$ and Lemma 5.1 states $OPT(L) \leq 4 \cdot OPT(R)$ thus $ALG(R) \leq 8 \cdot OPT(R)$. □

Now, we need to find an algorithm that gives the optimal solution for a laminar input to have our 8-approximation algorithm.

6 Dynamic program on the laminar instance

In this section, we present an algorithm in $O(n^4)$ that gives the exact solution to the stabbing problem for any laminar instance. The key idea to define this algorithm is to use some properties that any laminar instance has.

First, this algorithm is a dynamic program. The initial problem can be separated into smaller subproblems as any subset of a laminar set is still laminar. Moreover, by definition 5.1, we have that the laminar instance is composed of ”independent” sets. Thus, we can divide along the vertical lines that separate two independent sets and treat each side as an independent subproblem. This can be seen on examples by looking at the projection on the x-axis in Figures 11 and 13.

Secondly, for any set of rectangles that share some segments when projected on the x-axis, there is always one bigger than the other in the sense that all the other rectangles in the set have a projection on the x-axis contained in his. In other words, it is the rectangle with the largest width in this set. This rectangle, that we called *maxRect* in Section 4, must be stabbed for any *Ensemble* that represents a division of the problem. Henceforth, in the final solution, there will be a line segment from one extremity to the other of these *maxRect*s. The y-coordinate of this segment will be the minimum over all the possibilities, considering that a segment is supposed to be on the top of a rectangle.

Description of the algorithm : The starting problem is given by the original input. The base cases are subproblems with only one rectangle in them or zero. The associated solution is a segment that stabs horizontally this rectangle (on its top).

The first divisions are done vertically using the independences. This is shown on an example in Figure 14. On each subproblem E (that we also called *Ensemble* in Section 4), we consider the rectangle with maximum width $w_{maxRect}$. We know that a segment must stab it and must be on the top of a rectangle, thus we take the segment corresponding to :

$$OPT(E) = w_{maxRect} + \min_{r \in inside(maxRect)} \left(OPT(E_r^{up}) + OPT(E_r^{down}) \right)$$

where E_r^{up} and E_r^{down} are the two subproblems created on the top and bottom of the segment, containing rectangles not already stabbed by the segment on top of r. The set $inside(maxRect)$ represents all the possible segments (by the rectangles that fix their y-coordinate). This is shown on an example in Figure 15.

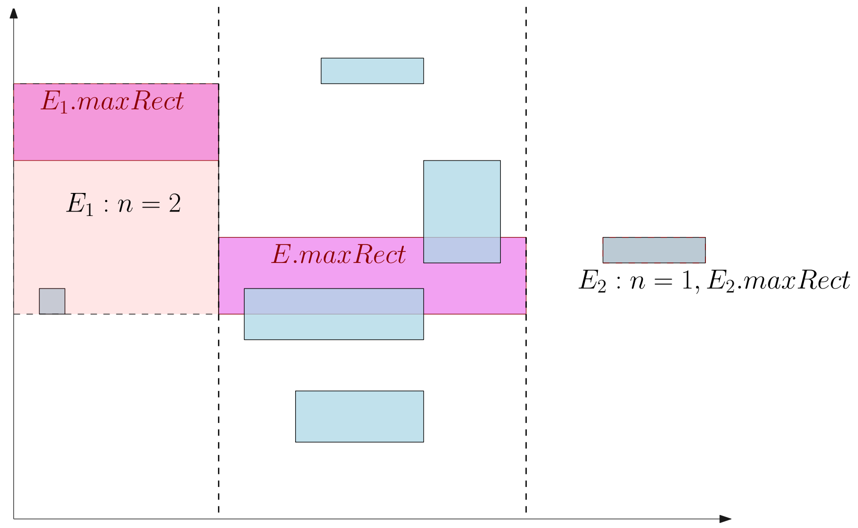
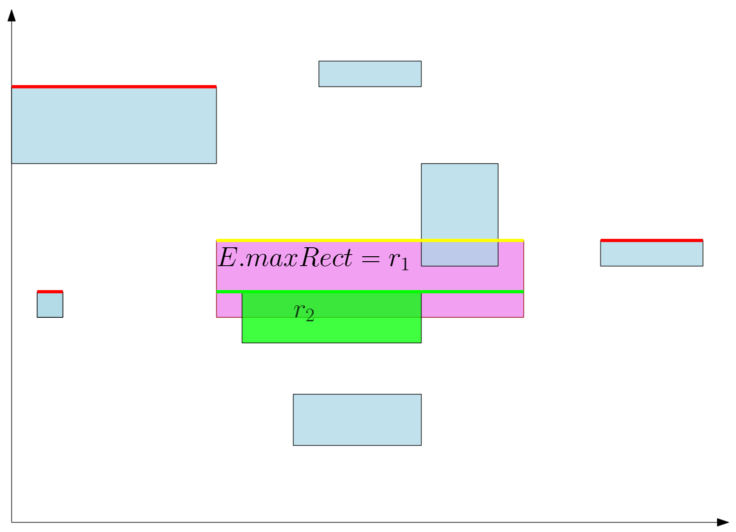
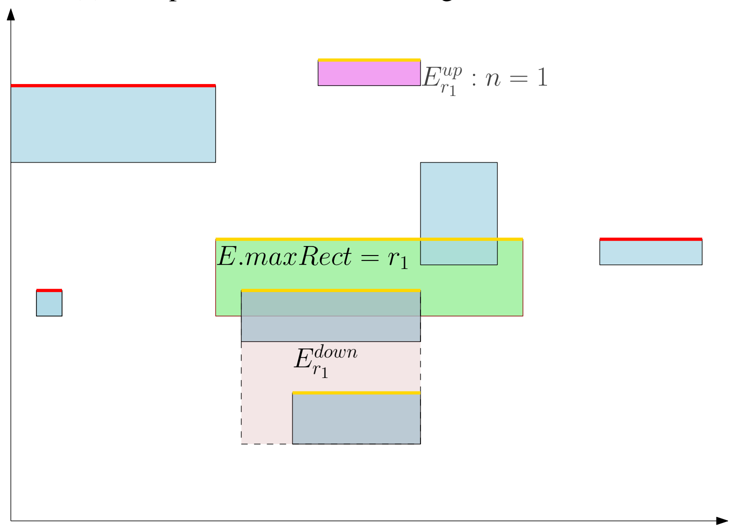


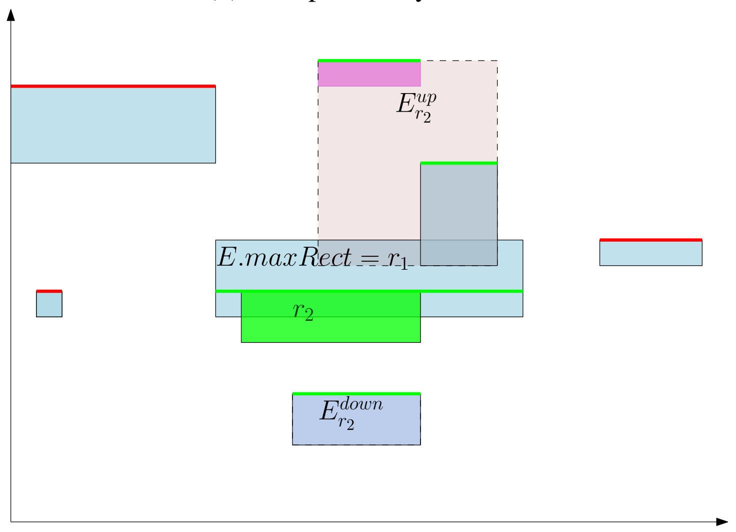
Figure 14: Example of vertical division giving three independent sets



(a) Two possibilities for stabbing *maxRect*



(b) First possibility



(c) Second possibility

Figure 15: Example of the choices for a horizontal division

The algorithm to compute $OPT(input)$ is resumed in Algorithm 2 (for the reasons of clarity the segments are not remembered in this algorithm, however, they will be in its true implementation in section 8). The sets E_{left} and E_{right} are the set on the left and right of $E.maxRect$ created by a vertical division.

Algorithm 2 OPT(E)

```
if $E.n = 0$ then
    Return 0
else if $E.n = 1$ then
    Return $w_{E.maxRect}$
else
    Return $w_{E.maxRect} + OPT(E_{left}) + OPT(E_{right}) + \min_{r \in inside(maxRect)} \left(OPT(E_r^{up}) + OPT(E_r^{down})\right)$
end if
```

Theorem 6.1. *The algorithm presented runs in* $O(n^4)$.

Proof. As the algorithm is recursive, its complexity is the number of divisions multiplied by the time taken to do one division.

Here we only do simple operations and a for loop. This for loop is in $O(n)$, as we do it on a subset of the set of rectangles.

The number of divisions is represented by the number of *Ensemble* we consider. For the y-axis, there are at most n choices for the bottom coordinate and n choices for the top coordinate, as the coordinates of an *Ensemble* correspond to the coordinates of a rectangle. Moreover, the laminar properties guarantee that no rectangle could have one x-coordinate inside the *Ensemble* and the other out, as they have to be either included in *maxRect* or completely dissociated. This implies that for the x-axis, we only have to choose the *maxRect* in the set, thus there are n choices.

Eventually, the total running time is $O(n \times n^2 \times n) = O(n^4)$.

□

7 Tight bound and improvements to the implementation

We showed in section 5 that the algorithm described by transforming the laminar instance, solving exactly the stabbing problem (with the algorithm from 6), and multiplying by 2 the solution gives an 8-approximation. In this section, we will show with a specific example that this bound is tight. Then, we will present some improvements to the implementation that were done to reduce the average ratio and avoid critical cases. For the moment, there is no formal proof that these improvements affect the theoretical bound.

7.1 Tight bound

We found a sequence of examples that have an approximation ratio (ALG/OPT) converging to 8 and for which it is necessary to expand the solution of the laminar instance.

Let's consider 3 rectangles for any $k > 0$:

- The first one with $x_b = 2^{k+1}$ and $w = 2^k + 1$,
- The second one with $x_b = 2^{k+1} - 1$ and $w = 2^k + 1$,

- The third one with $x_b = 2^{k+2} - 1$ and $w = 2$.

Since the transformation to laminar pushes the second rectangle to the left, while the first one stays at the same place, the DP-algorithm stabs the two rectangles independently. The third rectangle can not be entirely stabbed without augmenting the laminar solution. This is illustrated on an example in Figure 16. Thus, $ALG = (2^k + 1 + 2^k + 1) \times 2 = 2^{k+3}$ as the first two rectangles have a width of $2^k + 1$ and in the laminar instance, the third one is stabbed by the same segment as the first one. The last multiplication by two comes from the multiplication by 2 to go from the laminar solution to the final one.

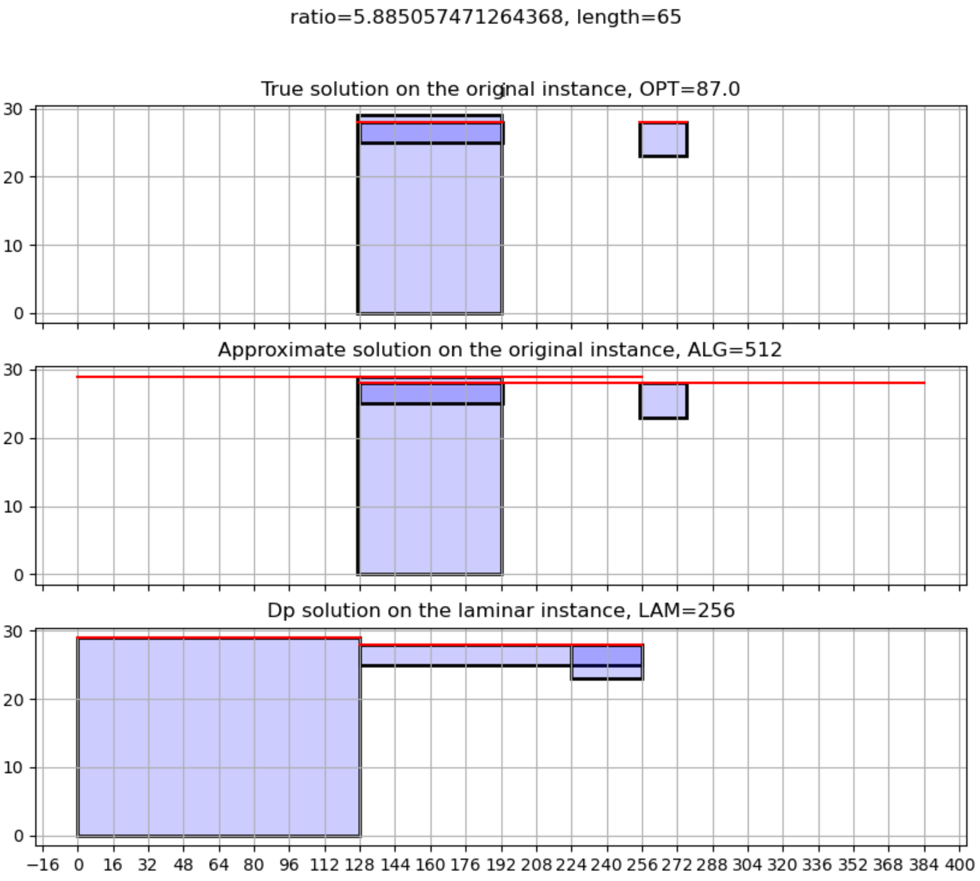


Figure 16: Solution given for one such example

The optimal solution stabs the first two rectangles with the same segment of length $2^k + 1 + 1 = 2^k + 2$ as their width is $2^k + 1$ and the second starts 1 unit before the first one. The optimal solution also stabs the third rectangle of width 2. Thus $OPT = 2^k + 2 + 2 = 2^k + 4$. This is pictured in Figure 17.

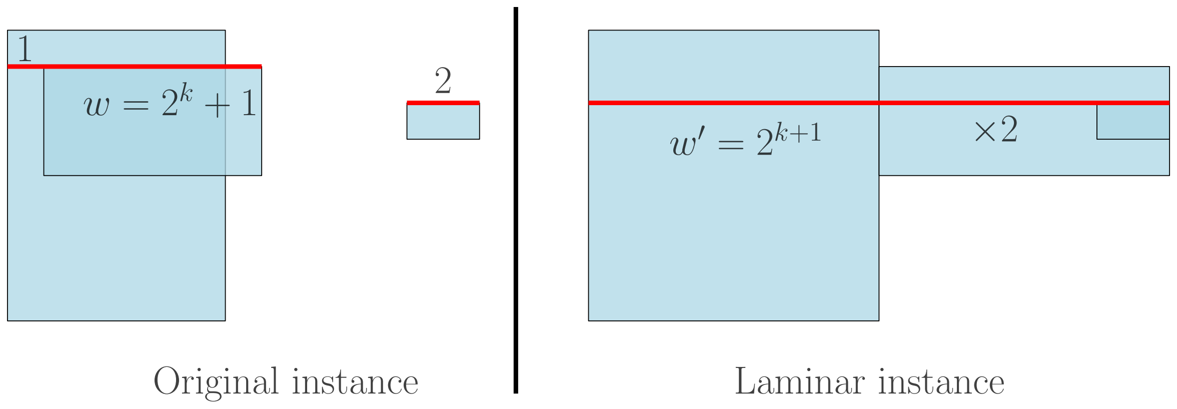


Figure 17: Inputs used to show the tight bound

Eventually, the approximation ratio for this input converges to 8 as k converges to infinity:

$$\frac{ALG}{OPT} = \frac{2^{k+3}}{2^k + 4} \to 2^3 = 8.$$

Thus, the 8 bound proven in theorem 5.1 is tight for this algorithm.

7.2 Improvement to the implementation

Considering the example presented before and some others obtained while running the algorithm, we observed that the ratio is particularly large when a large part of the segment does not cover any rectangle. To improve the covering, we first cut the segment such that it began on the left extremity of the first rectangle stabbed and finish on the right extremity of the last one. This allowed to solve some of the additional length coming from the shifting of rectangles and the doubling of segments.

However, due to the shifting during the transformation to laminar, some segments were stabbing two far away rectangles with this additional length between the two. This was partially solved by dividing the problem into *connected component* and solving the stabbing problem independently on each. Two rectangles are in the same connected component if there exists a path along the edges of some rectangles that connect their corners (without loss of generalities, we can only consider their bottom-left corner). In other words, if a segment stabs rectangles from different connected components, then a part of this segment does not cover any rectangle. Since this segment is not optimal and does not appear in a solution of minimal length, we can consider each connected component independently. An example of a division into connected components is shown in Figure 18.

After these two modifications, most of the additional length comes from rectangles being stabbed twice due to shifting. Unfortunately, as the code implemented does not keep track of which rectangles are stabbed by a segment, this should be done greedily. It was not implemented here.

Before these modifications, the average ratio was 2.61 (over 53434 random instances with $n < 20$). After, it was reduced to 1.09 (over 53651 random instances with $n < 20$). Moreover, even if the running time is still in $O(n^4)$, the algorithm is a bit quicker (as we run it on smaller instances that sum to n rectangles).

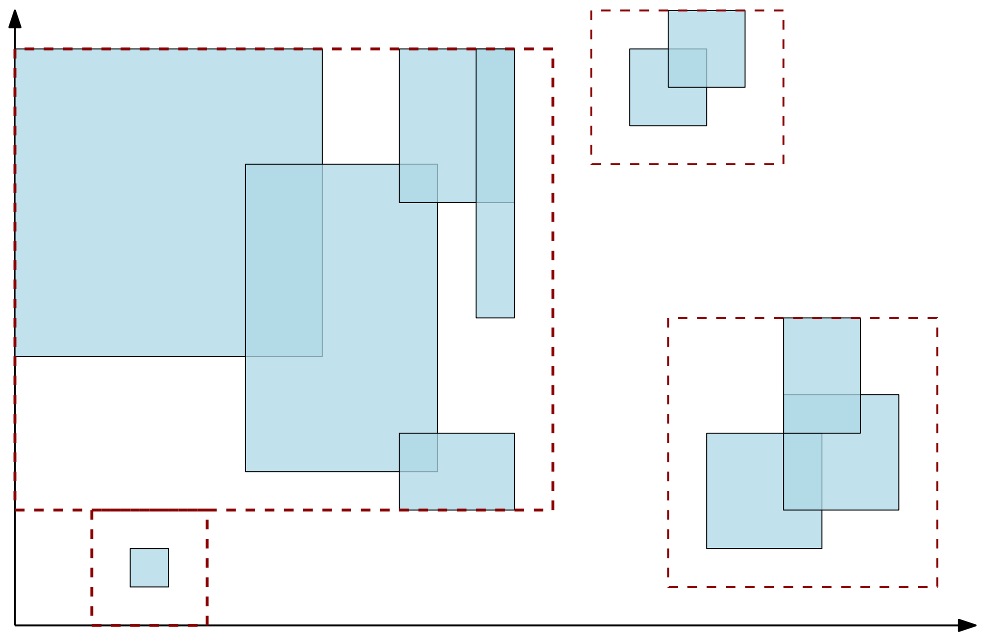


Figure 18: Example of a division into connected components

8 Presentation of the code

In this section, we will present the code for the implementation of this algorithm. To run all the functions, the libraries operator, math, copy, igraph, numpy, matplotlib, random, time and gurobipy need to be installed. The uses of the different files in the GitHub repository are also explained.

Class: Three classes are implemented in the file *ClassRectangle.py*:

- The *Rectangle* is defined as in Section 4.

```python
class Rectangle:
    def __init__(self,xb,yb,xh,yh):
        #(xb,yb) coordinate of the lower left corner, (xh,yh) coordinate of the upper right corner, w the lenght, must have xb<xh and yb<yh
        self.xb = xb
        self.yb=yb
        self.xh=xh
        self.yh=yh

    @property
    def w(self):
        return self.xh-self.xb

```

  They are initialized using the following constructor and their width is dynamically computed (change if the relevant coordinates of the rectangle change).

```python
    Rectangle(xb,yb,xh,yh)

```

- The *Segment* is also defined as in Section 4.

```python
class Segment:
    def __init__(self,start,end,h):
        #Construct the segment starting at (s,h) and finishing at (e,h) with lenght l=e-s, must have s<h
        self.s=start
        self.e=end
        self.h=h

    @property
    def l(self):
        return self.e-self.s
```

They are initialized using the following constructor and their length is dynamically computed.

```python
Segment(s,e,h)
```

- The *Ensemble* is also defined as in Section 4 but with additional attributes. They are a copy of the first list of rectangle given to the constructor *Origin_Rect*, a boolean *is_laminar* indicating if the original set of rectangle is laminar using the function *test_laminar()* and a dynamic *name* (string) of the form ”min x_b, min y_b, max x_h, max y_h” that will be used in dictionaries to refer to this instance. The attributes *n, minxb, minyb, maxxh, maxyh, MaxRect* and *name* are handled dynamically and will change if the set of rectangle in the *Ensemble* is changed (as during the transformation to laminar). Additionally, to the function *test_laminar*, the class has the *transform_to_laminar()* function to transform the set of rectangles using the procedure in Section 5 (the procedure will not be applied if the set is already laminar). The class has one function *coupure(x1,y1,x2,y2)* that gives a smaller *Ensemble*, included in it, defined by the rectangles with x-coordinates in $[x1, x2]$ and y-coordinates in $[y1, y2]$. The final function *inside()* returns the list of rectangles with their top inside *MaxRect* (the one considered for the min in Section 6). The corresponding code is :

```python
class Ensemble:
    def __init__(self,List_Rects):
        #Construct the ensemble of rectangles containing the n rectangles from Rects, minxb/minyb is the smallest coordinate for the left/bottom of a rectangle,
        # maxxh/maxyh  is the largest coordinate for the right/top of a rectangle, maxRect is the rectangle with the biggest lenght w
        self.Rects=List_Rects
        self.Origin_Rect=copy.deepcopy(List_Rects)
        self.is_laminar=self.test_laminar()

    @property
    def n(self):
        return len(self.Rects) #number of rectangles

    #coordinates of the frame of the ensemble
    @property
    def minxb(self):
        return min(self.Rects,key=attrgetter('xb')).xb

    @property
    def minyb(self):
        return min(self.Rects,key=attrgetter('yb')).yb

    @property
    def maxxh(self):
        return max(self.Rects,key=attrgetter('xh')).xh

    @property
    def maxyh(self):
        return max(self.Rects,key=attrgetter('yh')).yh

    #Rectangle with the maximum length in the ensemble
    @property
    def maxRect(self):
        return max(self.Rects,key=attrgetter('w'))

    #name determined by its frame, change if the frame is changed, if empty its name is '0'
    @property
    def name(self):
        if self.n==0:
            return '0'
        return str(self.minxb)+','+str(self.minyb)+','+str(self.maxxh)+','+str(self.maxyh)

    #test if laminar
    def test_laminar(self):
        lam=True
        for R1 in self.Origin_Rect: #seen as the "big" one
            for R2 in self.Origin_Rect: #seen as the one included inside
                if (R2.xb>R1.xb and R2.xb<R1.xh and R2.xh>R1.xh) or (R2.xb<R1.xb and R2.xh>R1.xb and R2.xh<R1.xh):
                    lam=False
                    return lam
        return lam

    #fonction to transform the general instance into a laminar instance (this change its name)
    def transform_to_laminar(self):
        if not self.is_laminar:
            for R in self.Rects:
                w=2**(ma.ceil(ma.log2(R.w)))
                R.xb=(R.xb // w)*w
                R.xh=R.xb+w

    #fonction to get a smaller sub-ensemble with lower corner (x1,y1) and uper corner (x2,y2)
    def coupure(self,x1,y1,x2,y2):
        return Ensemble([R for R in self.Rects if (R.xb>=x1 and R.yb>=y1 and R.xh<=x2 and R.yh<=y2 )])

    #fonction to get all the Rectangle inside of the maxRect and the maxRect itself
    def inside(self):
        ins=[]
        for R in self.Rects:
            if(R.xb>=self.maxRect.xb and R.xh<=self.maxRect.xh and R.yh>=self.maxRect.yb and R.yh<=self.maxRect.yh):
                ins.append(R)
        return ins
```

An *Ensemble* is constructed as :

```
Ensemble(List_Rectangles)
```

Functions: The file *Dpfonction.py* contains the useful functions that constitute the algorithm and the implementation improvements described before.

The recursive function *DPstabbing(E,opti,segm)* is the implementation of Algorithm 2 with *E* an *Ensemble*, *opti* a dictionary containing the optimal value for the *Ensemble* already considered and *segm* a list that keeps state of the segments associated with the solution.

```
#main part of the dynamic program, take an Ensemble E on which he
    operates,
# a dictionary opti to remember the already computed values and a list
    segm which contains the segments in the solution
def DPstabbing(E,opti,segm):
    #already computed
    if E.name in opti :
        segs=opti[E.name][1]
        if segs!=[]:
            segm.extend(copy.deepcopy(segs))
        return opti[E.name][0]

    #simple cases
    if E.n==0 :
        opti[E.name]=[0,[]]
        return 0
    if E.n==1 :
        seg=ClassRectangle.Segment(E.maxRect.xb,E.maxRect.xh,E.maxRect.yh
    )
        opti[E.name]=[E.maxRect.w,[seg]]
        segm.append(seg)
        return opti[E.name][0]

    #the lenght of the maxRect and the solution of the problem on its
    left and right
    opti[E.name]=[E.maxRect.w + DPstabbing(E.coupure(E.minxb,E.minyb,E.
    maxRect.xb,E.maxyh),opti,segm) + DPstabbing(E.coupure(E.maxRect.xh,E.
    minyb,E.maxxh,E.maxyh),opti,segm),[]]

    #the optimal choice to place the segment in MaxRect to cut between
    the top and bottom
    Rins=E.inside()
    optvert=0
    value=False #to record if optvert is a true value
    Ropti=E.maxRect
    segmtoadd=[]
    for R in Rins:
        segmtemp=[]
        tomin=DPstabbing(E.coupure(E.maxRect.xb,E.minyb,E.maxRect.xh,R.yh
    -1),opti,segmtemp)+DPstabbing(E.coupure(E.maxRect.xb,R.yh+1,E.maxRect
    .xh,E.maxyh),opti,segmtemp)
        if tomin<optvert or value==False :
            optvert=tomin
            Ropti=R
            value=True
            segmtoadd=copy.deepcopy(segmtemp)

    opti[E.name][0]+=optvert
    opti[E.name][1]=copy.deepcopy(segmtoadd)+[ClassRectangle.Segment(E.maxRect.xb,E.maxRect.xh,Ropti.yh)]
    segm.extend(copy.deepcopy(segmtoadd))
    segm.append(ClassRectangle.Segment(E.maxRect.xb,E.maxRect.xh,Ropti.yh))

    return opti[E.name][0]
```

The function *transform_to_feasible(E,segm)* is used to cut the useless parts of the segment in *segm* as described in Section 7.2 considering the rectangles in *E*.

```
#transform the set of segments to make a feasible solution for the original problem:
#check how much augmenting is necessary on each sides of a segment
def transform_to_feasible(E,segm):
    if not(E.is_laminar): #if E is laminar no changes are needed
        segm_feasible=[]
        for s in segm:
            doubled_end=s.e+s.l
            start=doubled_end
            end=s.s
            for R in E.Origin_Rect:
                stabbed=R.xb>=s.s and R.xh<=doubled_end and R.yb<=s.h and R.yh>=s.h #if R is stabbed by s
                if ( stabbed and start>=R.xb): #check if the new segment need to be extended to stab R
                    start=R.xb
                if (stabbed and end<=R.xh):
                    end=R.xh
            segm_feasible.append(ClassRectangle.Segment(start,end,s.h))
        return segm_feasible
    else:
        return segm
```

The function *cut_connected_component(E)* returns a list of *Ensembles* with rectangles from different connected components, as described in Section 7.2. The principle of the function is to associate each rectangle to a vertex in a graph and put an edge between two vertices if the two corresponding rectangles intersect. The connected components of the *Ensemble* correspond to the clusters in the graph.

```
#To cut the problem into a list of connected component to run seperately the use of graph is necessary
def cut_connected_component(E):
    #E is an ensemble and the fonction return a list of ensemble that can be treated independentaly
    g=igraph.Graph()
    g.add_vertices(E.n) #vertex i will correspond to Rects[i]

    #create an edge if two rectangles are touching each other
    for i in range(0,E.n):
        R1=E.Rects[i]
        for j in range(i+1,E.n):
            R2=E.Rects[j]
            #test if intersect : 1st line test x, 2nd line test y
            if ((R1.xb<=R2.xb and R2.xb<=R1.xh) or (R1.xb<=R2.xh and R2.xh<=R1.xh)or(R2.xb<=R1.xb and R1.xb<=R2.xh) or (R2.xb<=R1.xh and R1.
    xh<=R2.xh)) \
            and ((R1.yb<=R2.yb and R2.yb<=R1.yh) or (R1.yb<=R2.yh and R2.
    yh<=R1.yh)or(R2.yb<=R1.yb and R1.yb<=R2.yh) or (R2.yb<=R1.yh and R1.
    yh<=R2.yh)) :
                g.add_edges([(i,j)])

    #find the connected components in the graph
    Conn_comp=g.clusters()

    #associate each connected component in the graph to the corresponding
     ensemble
    List_E=[]
    for cluster in Conn_comp:
        List_Rect=[]
        for v in cluster:
            List_Rect.append(E.Rects[v])
        List_E.append(ClassRectangle.Ensemble(List_Rect))

    return List_E
```

Applications: The algorithm is used in three files *main.py, approxi_particular exemple.py* and *test approximation*. The first one only gives the solution returned by the algorithm, while the two others compare the optimal solution, the algorithm solution and the solution on the laminar instance. The second one is used to test on one instance, while the third one tests on multiple random instances.

The *Ensemble* is usually defined as in the following examples:

```
R1=ClassRectangle.Rectangle(0,60,64,81)
R2=ClassRectangle.Rectangle(32,75,64,100)
R3=ClassRectangle.Rectangle(64,9,83,20)
R4=ClassRectangle.Rectangle(60,15,84,22)
E=ClassRectangle.Ensemble([R1,R2,R3,R4])
```

or

```
List_Rect=[ClassRectangle.Rectangle(15,0,34,29),ClassRectangle.Rectangle
    (16,25,35,28)]
E=ClassRectangle.Ensemble(List_Rect)
```

or for a random case

```
n=rd.randrange(5,9) #max number of Rectangle in E -1
maxx=60 #maximum value for x-1 and y-1
maxy=60
List_Rect=[]
for i in range(0,n-1):
    xb=rd.randrange(0,maxx-1)
    yb=rd.randrange(0,maxy-1)
    List_Rect.append(ClassRectangle.Rectangle(xb,yb, rd.randrange(xb+1,
   maxx), rd.randrange(yb+1,maxy)))
E=ClassRectangle.Ensemble(List_Rect)
```

Then the different functions that constitute the algorithm are called as such (some additional lists are made if the laminar solution is also plotted) :

```
opti={}
segm_feasible=[]
list_E=Dpfonction.cut_connected_component(E)
for e in list_E:
    segms=[]
    e.transform_to_laminar()
    Dpfonction.DPstabbing(e,opti,segms)
    segm_laminar.extend(segms)
    local_feasible=Dpfonction.transform_to_feasible(e,segms)
    segm_feasible.extend(local_feasible)
sol_approx=sum(s.l for s in segm_feasible)
```

To plot the solution given by the algorithm, we use :

```
fig, ax=plt.subplots()
for R in E.Origin_Rect:
    ax.add_patch(Rectangle((R.xb,R.yb),R.w,(R.yh-R.yb),ec="black",fc
   =(0,0,1,0.2),lw=2))

for se in segm_feasible:
    ax.plot([se.s,se.e],[se.h,se.h],color='r')

ax.plot()
ax.set_title('Final solution on the original rectangles')

#tick postion
ax.xaxis.set_major_locator(plt.MultipleLocator(2))

ax.grid()
plt.show()
```

To find the optimal solution, we use the library gurobipy to solve the integer program constituted by all the possible segments. The code is also put below as a reference.

```
#create all feasible segments
    all_segm=[]
    for Rs in E.Rects: #give start
        for Re in E.Rects: #give end
            for Rh in E.Rects: #give hight
                if Rs.xb<Re.xh:
                    all_segm.append(ClassRectangle.Segment(Rs.xb,Re.xh,Rh
   .yh))

    #create list of weight/lenght
    w=[s.l for s in all_segm]

    #create M
    M = np.zeros((E.n, len(all_segm)))  # build matrix of set cover
   constraints
    i=0
    j=0
    for i in range(E.n):
        R=E.Rects[i]
        for j in range(len(all_segm)):
            s=all_segm[j]
            if s.h>=R.yb and s.h<=R.yh and s.s<=R.xb and s.e>=R.xh:  #
   conditions for a segment s to cover rectangle a
                M[i][j] = 1
            else:
                M[i][j] = 0

    #solve IP
    m = gp.Model("stab1")  # model IP
    x = m.addMVar(shape=len(all_segm), vtype=GRB.BINARY, name="x")  #
   variables
    obj=np.array(w) #objective function coefficients (weights)
    rhs = np.ones(E.n)
    m.setObjective(obj @ x, GRB.MINIMIZE)  # define objective function
    m.addConstr(M @ x >= rhs, name='c')
    m.optimize()
    exact_sol=m.objVal
    exact_segm=[all_segm[i] for i in range(len(all_segm)) if m.x[i]==1]
```

The file *test_complexity.py* returns a graph showing the time to run the algorithm for different n which allows to verify that the implementation is indeed in $O(n^4)$. The file *laminar_graph.py* only transforms the input set to a laminar one, without solving the stabbing problem.

Eventually, two folders contain less relevant files. The folder *figures* contain plots and text files containing ratios done at different moments of the implementation. The ones concerning the final program are in *figure and ratio after final changes*. The folder *test debug and presentation program* contains files to read the text files with the ratios and give their average (*average before.py* and *final average.py*), to test libraries(*test.py*) and also files that were used to debug (*test_time.py*) and one that became irrelevant with the implementation improvements (*limit_ratio.py*).

References

Chan, T. M., van Dijk, T. C., Fleszar, K., Spoerhase, J. and Wolff, A. (2018) Stabbing rectangles by line segments - how decomposition reduces the shallow-cell complexity. *CoRR* **abs/1806.02851**.

Eisenbrand, Gallato, Svensson and Venzin (2021) A qptas for stabbing rectangles .

Khan, A., Subramanian, A. and Wiese, A. (2021) A ptas for the horizontal rectangle stabbing problem.