rectangle-packer documentation

Release 2.0.1

Daniel Andersson

Jul 13, 2022

Contents

CHAPTER 1

Welcome to rectangle-packer

Primary use: Given a set of rectangles with fixed orientations, find a bounding box of minimum area that contains them all with no overlap.

This project is inspired by Matt Perdeck's blog post Fast Optimizing Rectangle Packing Algorithm for Building CSS Sprites[1].

- The latest documentation is available on Read the Docs[2].
- The source code is available on GitHub[3].

1.1 Installation

Install latest version from PyPI[4]:

```
$ python3 -m pip install rectangle-packer
```

Or clone the repository[5] and install with:

```
$ python3 setup.py install
```

1.2 Basic usage

```
# Import the module
>>> import rpack

# Create a bunch of rectangles (width, height)
>>> sizes = [(58, 206), (231, 176), (35, 113), (46, 109)]

# Pack
```

(continues on next page)

[1] http://www.codeproject.com/Articles/210979/Fast-optimizing-rectangle-packing-algorithm-for-bu
[2] https://rectangle-packer.readthedocs.io/en/latest/
[3] https://github.com/Penlect/rectangle-packer
[4] https://pypi.org/project/rectangle-packer/
[5] https://github.com/Penlect/rectangle-packer

(continued from previous page)

```
>>> positions = rpack.pack(sizes)

# The result will be a list of (x, y) positions:
>>> positions
[(0, 0), (58, 0), (289, 0), (289, 113)]
```

The output positions are the lower left corner coordinates of each rectangle in the input.

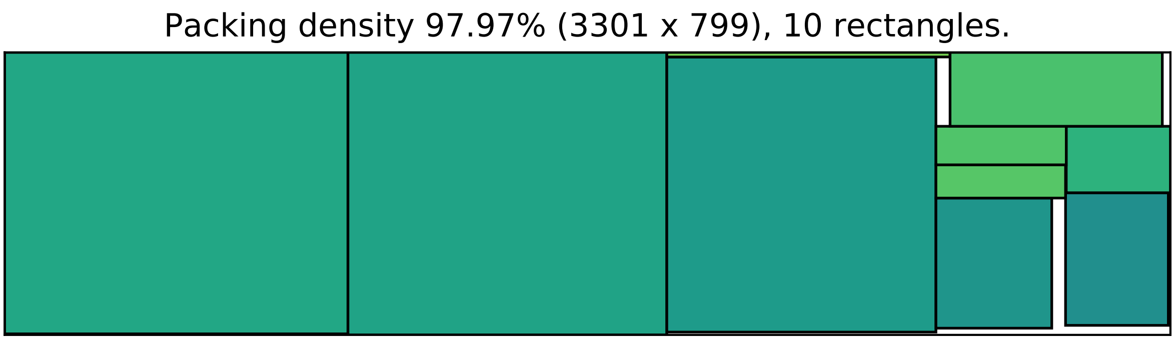
These positions will yield a packing with no overlaps and enclosing area as small as possible (best effort).

Note:

- You must use **positive integers** as rectangle width and height.
- The module name is **rpack** which is an abbreviation of the package name at PyPI (rectangle-packer).
- The computational time required by *rpack.pack()* increases by the number *and* size of input rectangles. If this becomes a problem, you might need to implement your own divide-and-conquer algorithm[6].

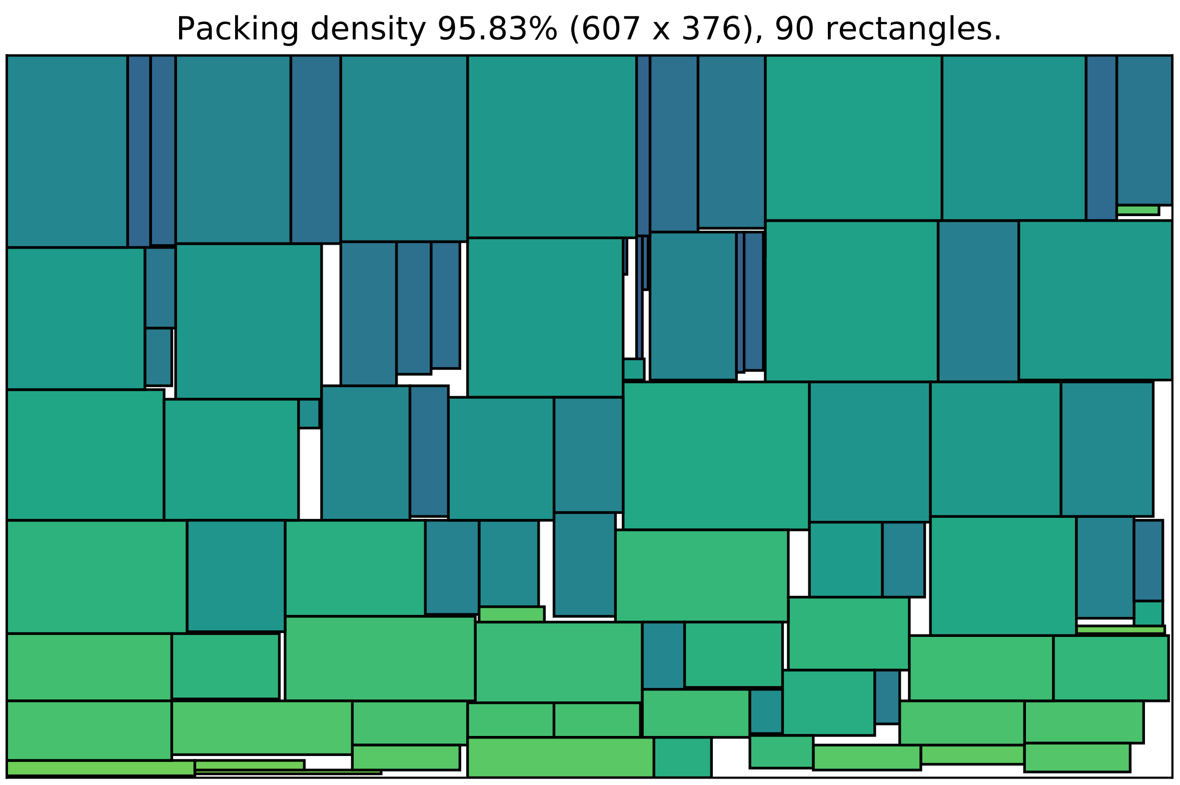
1.3 Examples

Example A:

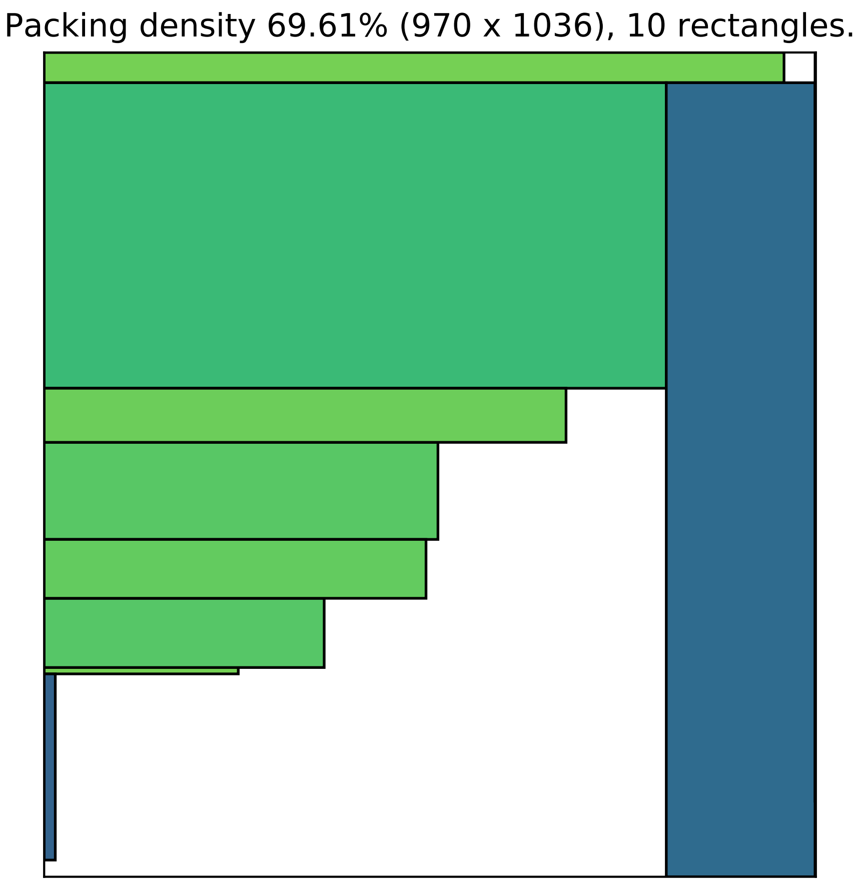


Example B:

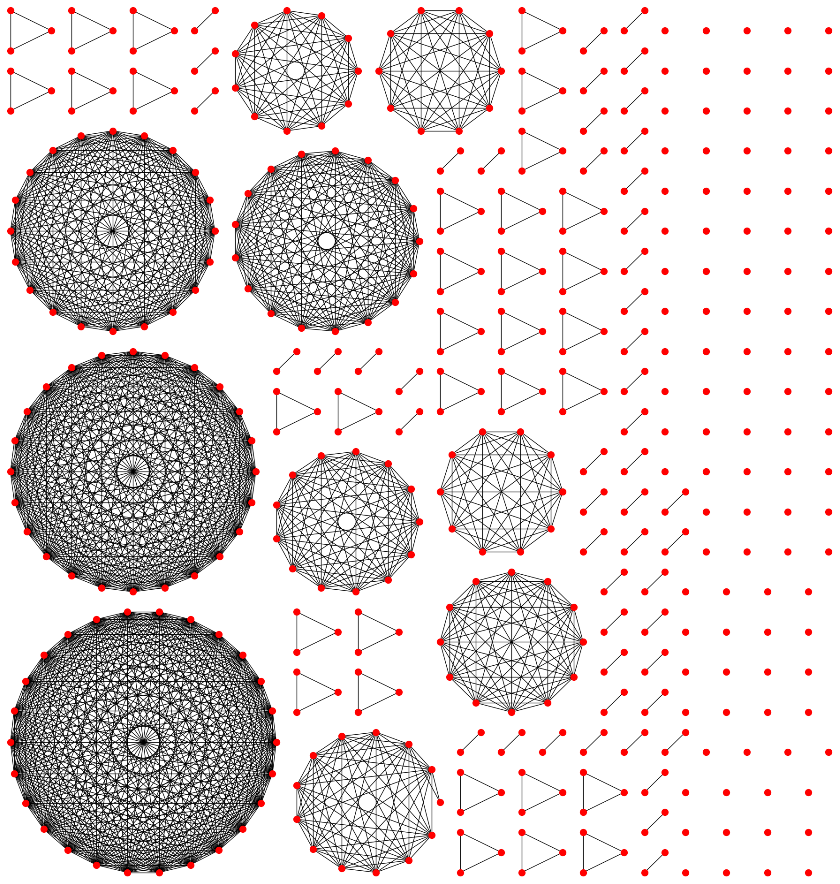
[6] https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm



Example C: Sometimes the input rectangles simply cannot be packed in a good way. Here is an example of low packing density:



Example D: The image below is contributed by Paul Brodersen, and illustrates a solution to a problem discussed at stackoverflow[7].



[7] https://stackoverflow.com/a/53156709/2912349

CHAPTER 2

Module Reference

2.1 Functions

`rpack.pack`(*sizes: Iterable[Tuple[int, int]], max_width=None, max_height=None*) → List[Tuple[int, int]]

Pack rectangles into a bounding box with minimal area.

The result is returned as a list of coordinates "(x, y)", which specifices the location of each corresponding input rectangle's lower left corner.

The helper function *`bbox_size()`* can be used to compute the width and height of the resulting bounding box. And *`packing_density()`* can be used to evaluate the packing quality.

The algorithm will sort the input in different ways internally so there is no need to sort `sizes` in advance.

The GIL is released when C-intensive code is running. Execution time increases by the number *and* size of input rectangles. If this becomes a problem, you might need to implement your own divide-and-conquer algorithm[8].

Example:

```
# Import the module
>>> import rpack

# Create a bunch of rectangles (width, height)
>>> sizes = [(58, 206), (231, 176), (35, 113), (46, 109)]

# Pack
>>> positions = rpack.pack(sizes)

# The result will be a list of (x, y) positions:
>>> positions
[(0, 0), (58, 0), (289, 0), (289, 113)]
```

Parameters

- **sizes** (*Iterable[Tuple[int, int]]*) – "(width, height)" of the rectangles to pack. *Note: integer values only!*

[8] https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm

- **max_width** (*Union[None, int]*) – Force the enclosing rectangle to not exceed a maximum width. If not possible, *rpack.PackingImpossibleError* will be raised.
- **max_height** (*Union[None, int]*) – Force the enclosing rectangle to not exceed a maximum height. If not possible, *rpack.PackingImpossibleError* will be raised.

Returns List of positions (x, y) of the input rectangles.

Return type List[Tuple[int, int]]

2.2 Exceptions

class rpack.**PackingImpossibleError**

Packing rectangles is impossible with imposed restrictions.

This can happen, for example, if *max_width* is strictly less than the widest rectangle given to *rpack.pack()*.

If possible, a partial result will be given in the second argument, with the positions of packed rectangles up till the point of failure.

2.3 Helper functions

rpack.**bbox_size**(*sizes*, *positions*) → Tuple[int, int]

Return bounding box size (width, height) of packed rectangles.

Useful for evaluating the result of *rpack.pack()*.

Example:

```
>>> import rpack

>>> sizes = [(58, 206), (231, 176), (35, 113), (46, 109)]
>>> positions = rpack.pack(sizes)

>>> bbox_size(sizes, positions)
(335, 222)
```

Parameters

- **sizes** (*List[Tuple[int, int]]*) – List of rectangle sizes (width, height).
- **positions** (*List[Tuple[int, int]]*) – List of rectangle positions (x, y).

Returns Size (width, height) of bounding box covering rectangles having *sizes* and *positions*.

Return type Tuple[int, int]

rpack.**packing_density**(*sizes*, *positions*) → float

Return packing density of packed rectangles.

Useful for evaluating the result of *rpack.pack()*.

Example:

```
>>> import rpack

>>> sizes = [(58, 206), (231, 176), (35, 113), (46, 109)]
>>> positions = rpack.pack(sizes)
```

(continues on next page)

(continued from previous page)

```
>>> packing_density(sizes, positions)
0.8279279279279279
```

Parameters

- **sizes** (*List[Tuple[int, int]]*) – List of rectangle sizes (width, height).
- **positions** (*List[Tuple[int, int]]*) – List of rectangle positions (x, y).

Returns Packing density as a fraction in the interval [0, 1], where 1 means that the bounding box area equals the sum of the areas of the rectangles packed, i.e. perfect packing.

Return type float

rpack.**overlapping**(*sizes*, *positions*)

Return indices of overlapping rectangles, else `None`.

Mainly used for test cases.

Example:

```
>>> sizes = [(10, 10), (10, 10)]

>>> positions = [(0, 0), (10, 10)]
>>> overlapping(sizes, positions) is None
True

>>> positions = [(0, 0), (5, 5)]
>>> overlapping(sizes, positions)
(0, 1)
```

Parameters

- **sizes** (*List[Tuple[int, int]]*) – List of rectangle sizes (width, height).
- **positions** (*List[Tuple[int, int]]*) – List of rectangle positions (x, y).

Returns Return indices (i, j) if i-th and j-th rectangle overlap (first case found). Return `None` if no rectangles overlap.

Return type Union[None, Tuple[int, int]]

CHAPTER 3

Benchmarks

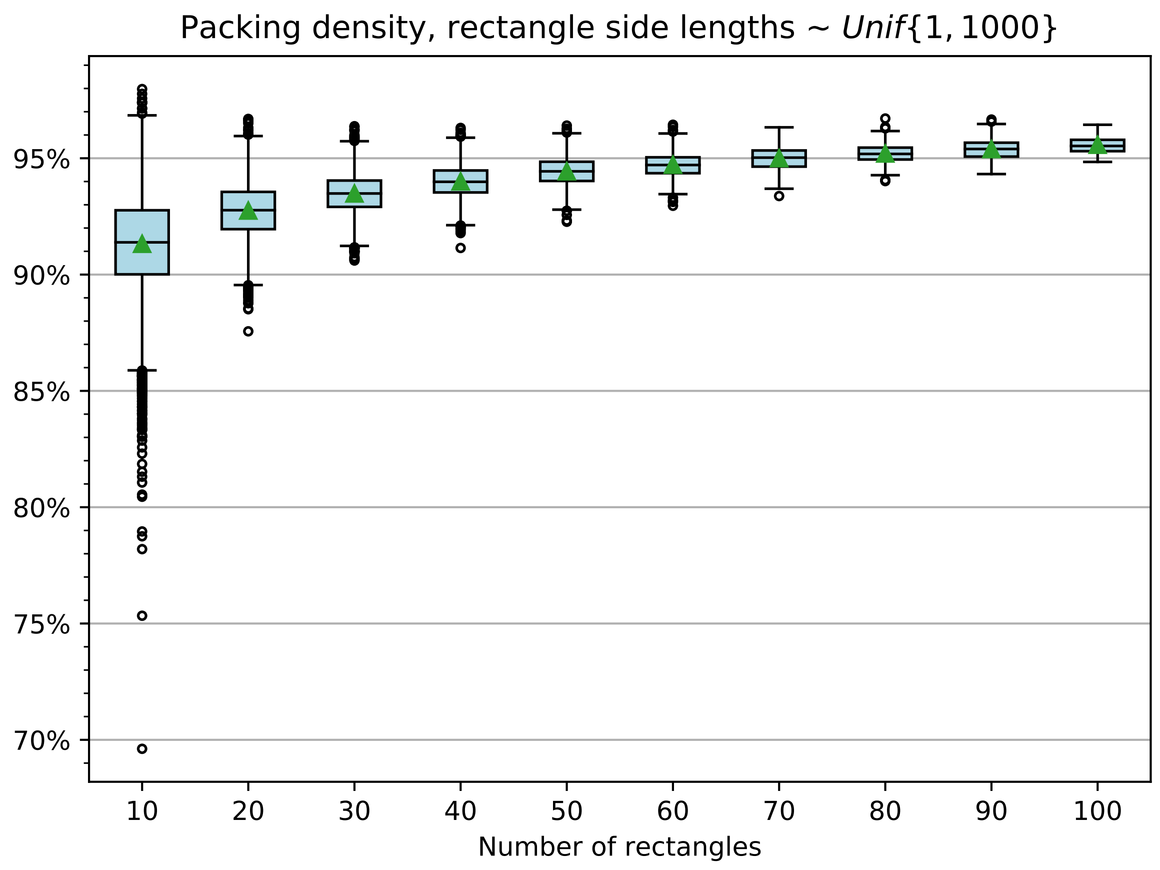
This section investigates the performance of `rpack.pack()`. The packing quality and time complexity are studied by packing many randomly generated rectangles.

3.1 Packing density

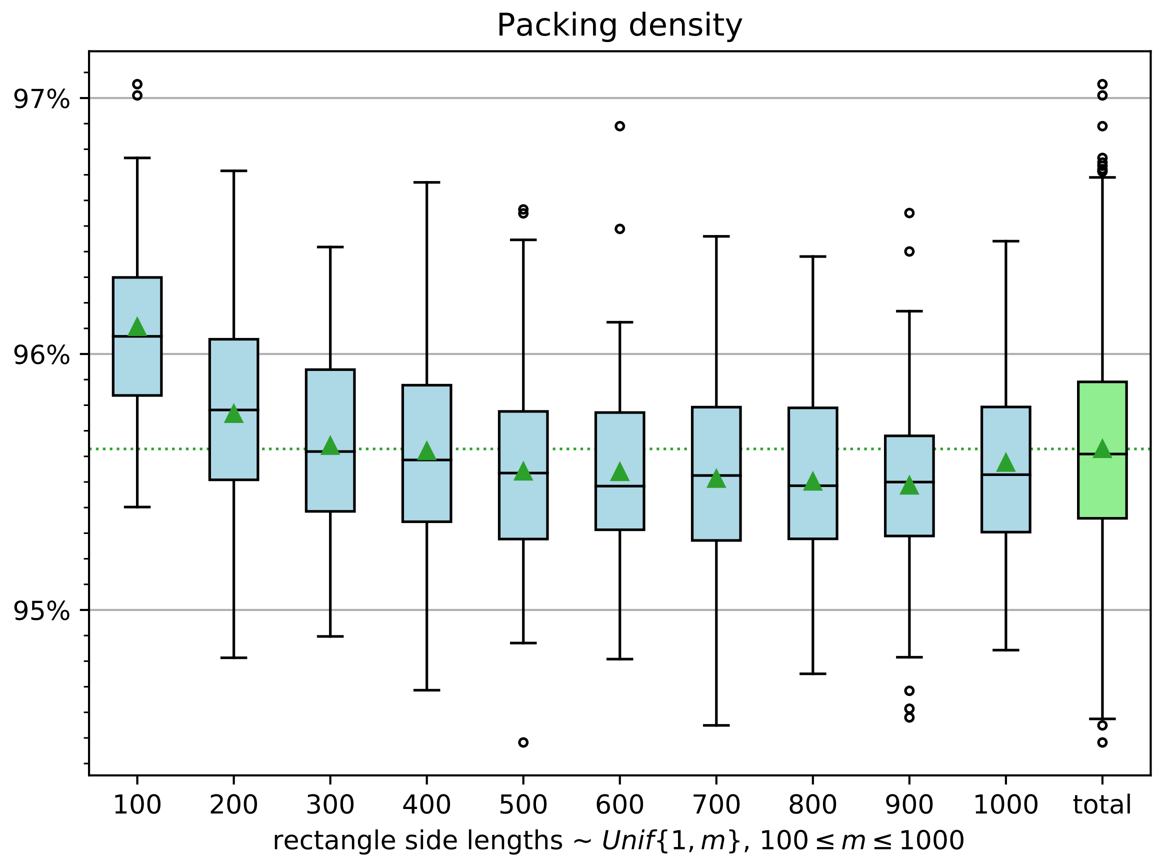
The *packing density* is defined as the fraction of the space filled by the rectangles in the bounding box. The objective of `rpack.pack()` is to obtain a packing of the greatest possible density.

The boxplot below shows how the packing density is correlated to the number of rectangles packed. Ten cases were generated, for 10, 20, . . . , 100 rectangles. For each case, rectangles were randomly generated by picking side lengths uniformly distributed in the interval [1, 1000]. If you are not familiar with boxplots, have a look at the boxplot[9] article at Wikipedia for an introduction.

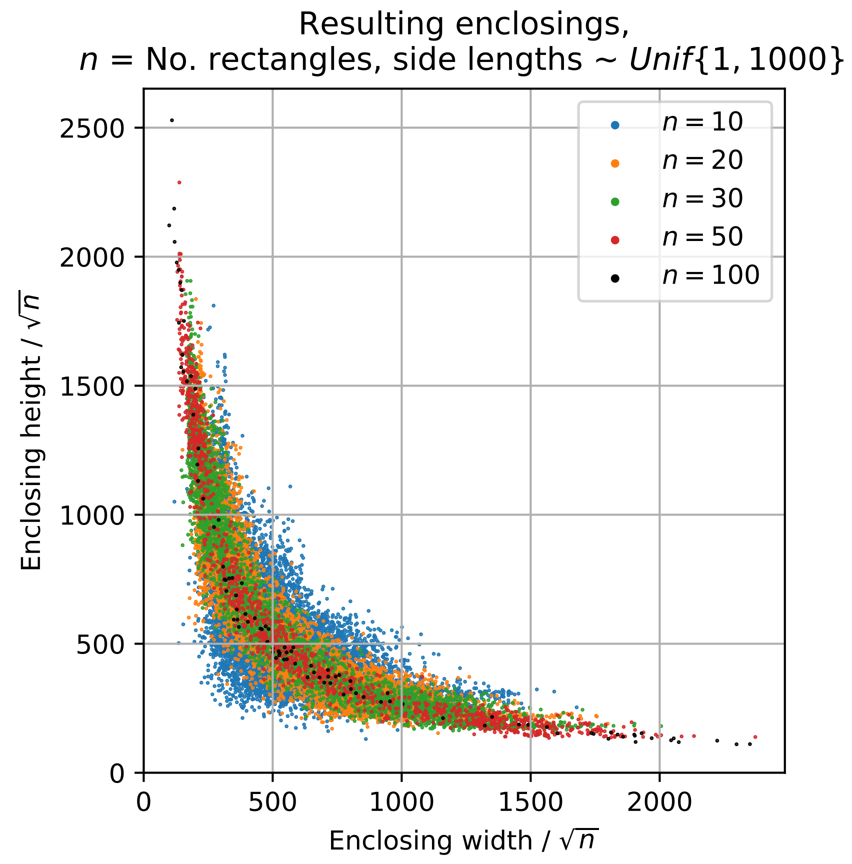
[9] https://en.wikipedia.org/wiki/Box_plot



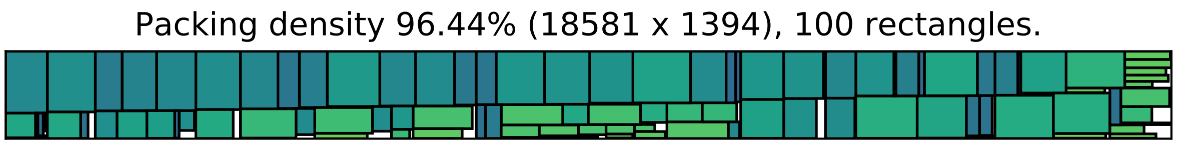
The boxplot below shows how the packing density varies by rectangle side lengths.



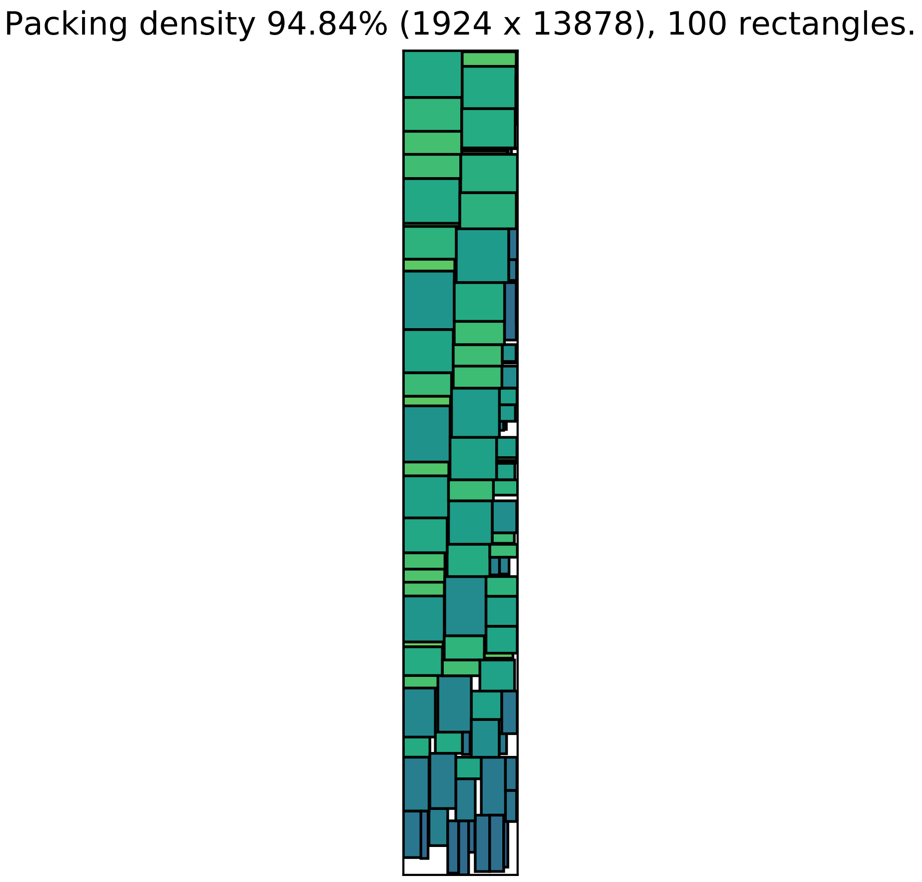
The scatter plot below shows how the minimal bounding box shape is distributed based on how many rectangles were packed. In all runs, random rectangles were generated by picking side lengths uniformly distributed in the interval [1, 1000]. It is interesting to note, that the minimal bounding box tends to be more extreme (very wide or very tall) when the number of rectangles increases.



Example of 100 rectangles, high packing density:



Example of 100 rectangles, lower packing density:



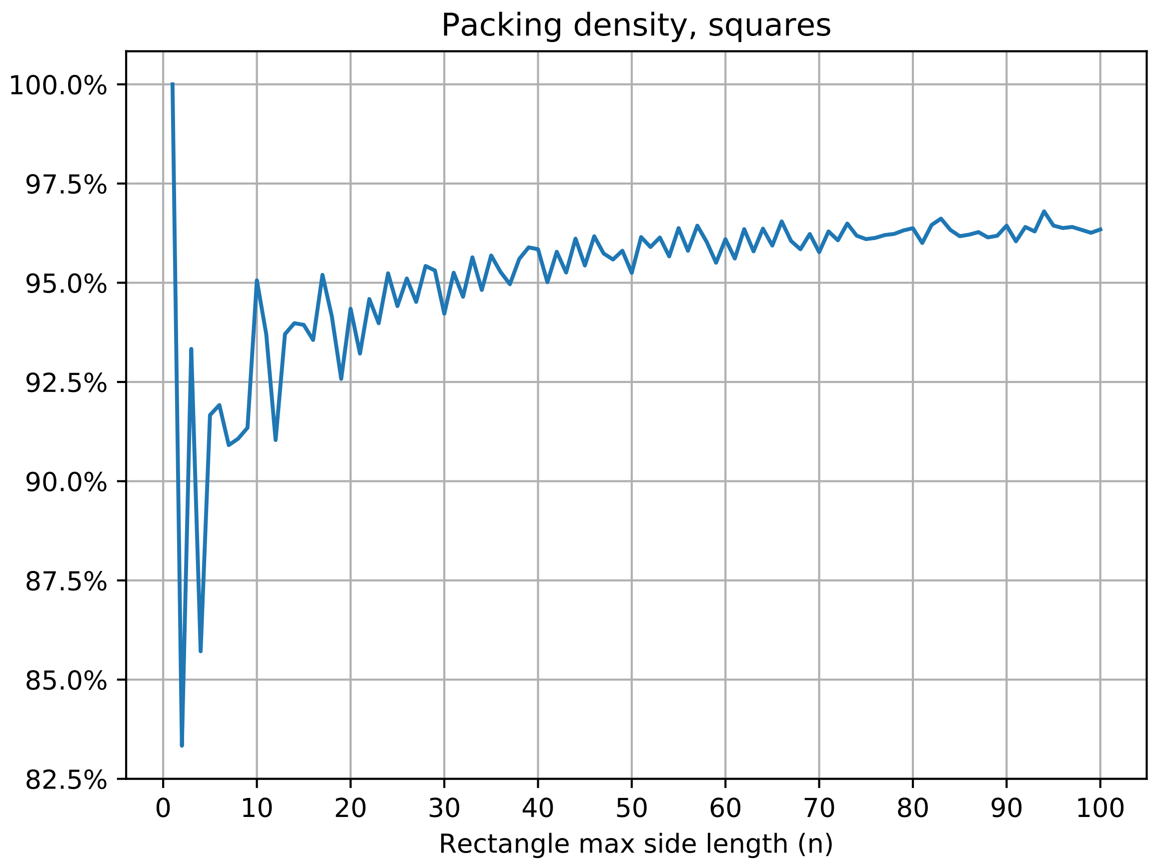
3.1.1 Squares test

The squares test packs an increasing number of square rectangles.

See this PDF, squares.pdf[10], for packed squares 1x1, . . . , NxN, for N = 1, . . . , 100. A summary is shown in the plot below.

See Richard E. Korf's paper Optimal Rectangle Packing: Initial Results[11] for the optimal solutions for all squares up to n=22.

[10] https://penlect.com/rpack/2.0.1/img/squares.pdf
[11] https://www.aaai.org/Papers/ICAPS/2003/ICAPS03-029.pdf

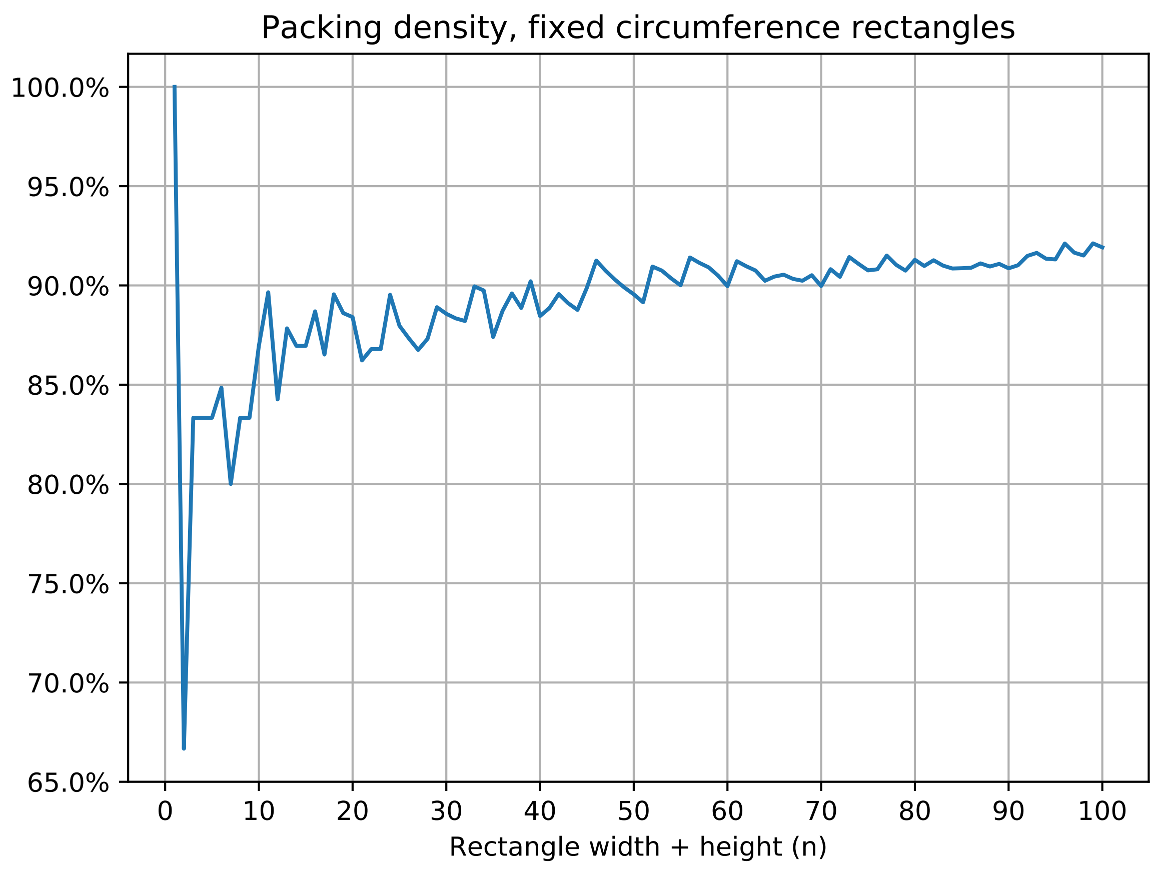


3.1.2 Circumference test

The circumference test packs an increasing number of rectagins having the same circumference. For exampe, N = 4, implies the rectangles 4x1, 3x2, 2x3 and 1x4.

See this PDF, circum.pdf[12], for packed rectangles Nx1, . . . , 1xN, for N = 1, . . . , 100. A summary is shown in the plot below.

[12] https://penlect.com/rpack/2.0.1/img/circum.pdf

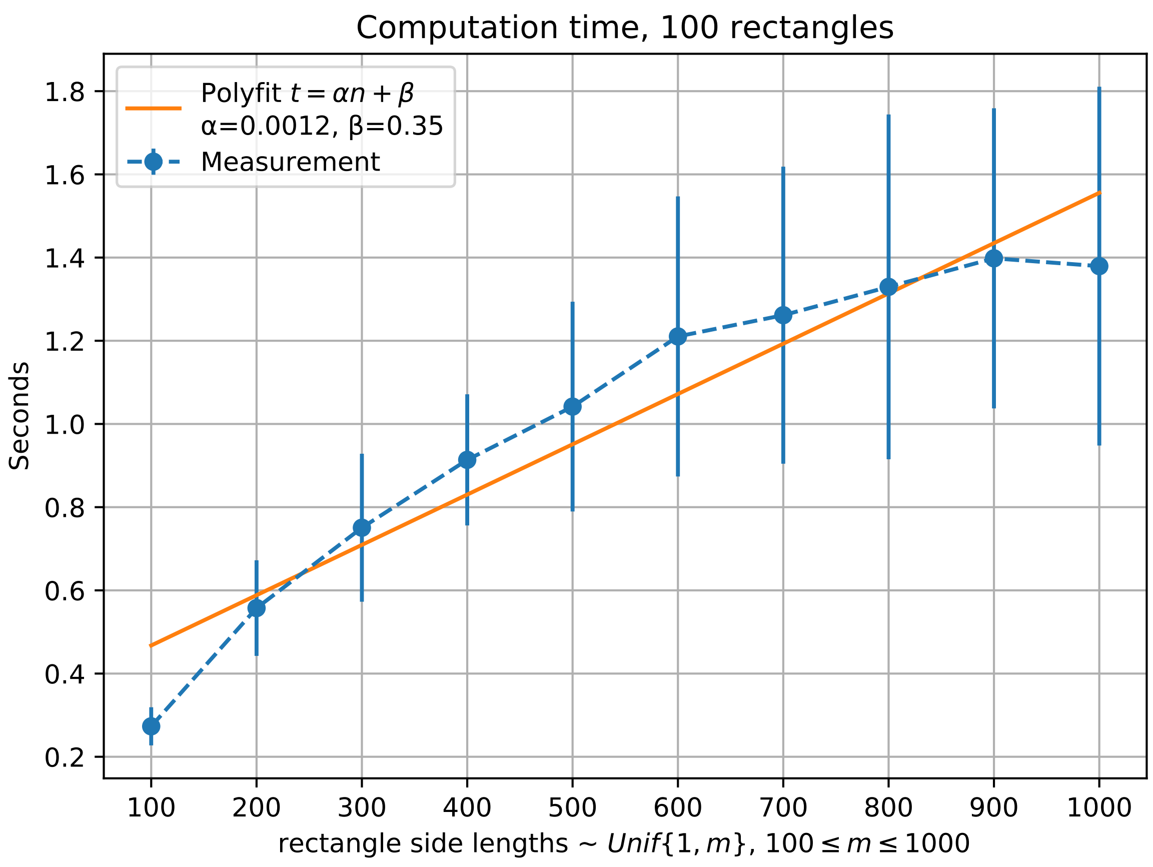


3.2 Time complexity

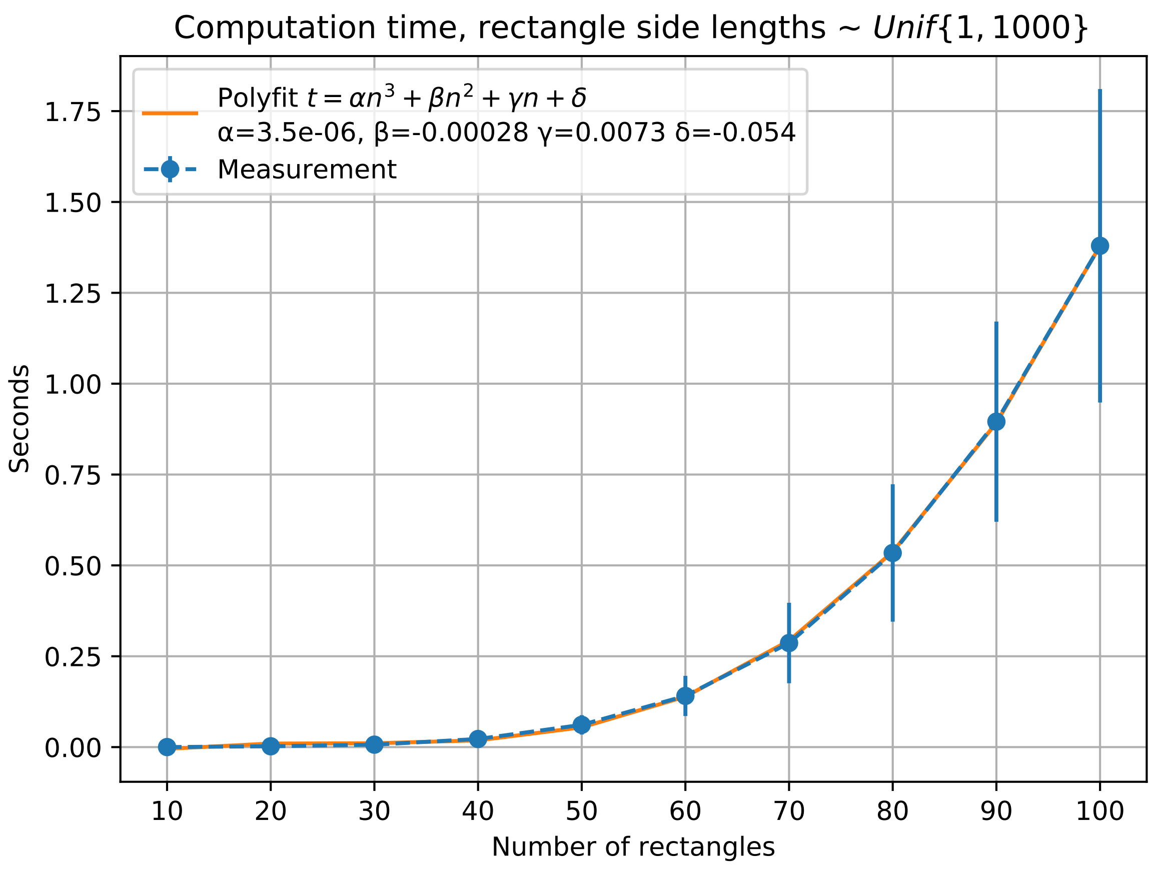
In computer science, the time complexity is the computational complexity that describes the amount of time it takes to run an algorithm.

This section presents how `rpack.pack()` performs with increasing rectangle count and size.

The figure below shows the increased computational cost when the average rectangle side length is increased. For each measurement point, 100 random rectangles were generated by picking independent uniformely distributed side lengths in the interval [1, 1000]. The procedure was repeated several times and the vertical lines indicates the standard deviation.



The figure below shows the increased computational cost when the number of rectangles are increased. For each measurement point, a set of random rectangles were generated by picking independent uniformely distributed side lengths in the interval [1, 1000]. For these 10 points, a qubic polynomial fits the data well (a quadratic does not). However, this is not a proof that the average complexety actually is O(n^3).


Computation time, rectangle side lengths ~ $Unif\{1, 1000\}$
Polyfit $t = \alpha n^3 + \beta n^2 + \gamma n + \delta$
α=3.5e-06, β=-0.00028 γ=0.0073 δ=-0.054
Measurement
Seconds
Number of rectangles
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
10
20
30
40
50
60
70
80
90
100

CHAPTER 4

The Algorithm

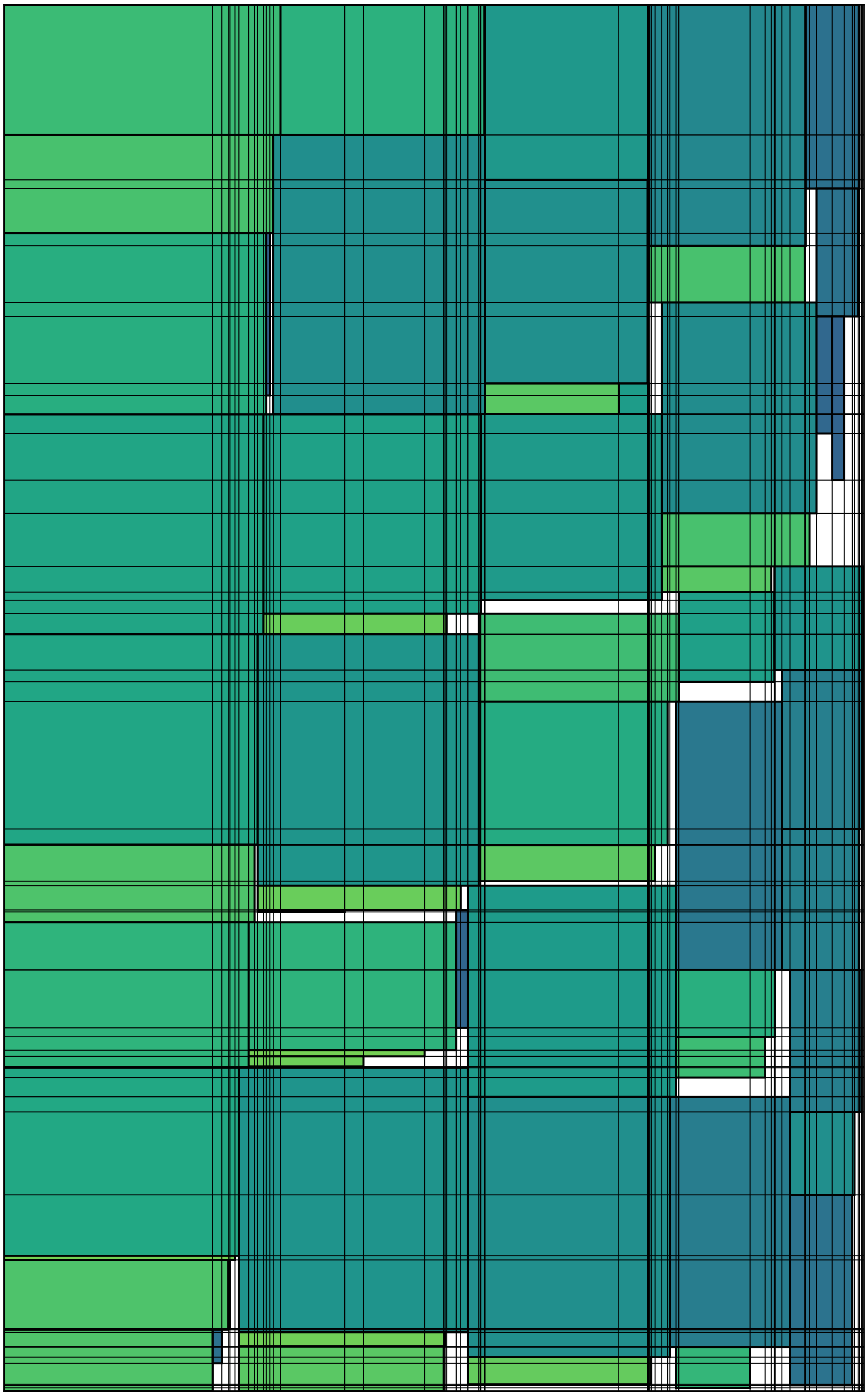
To find the minimum bounding box, a lot of candidate bounding boxes are tested.

The algorithm keeps track of a grid of cells internally. Each cell belongs to a column and a row. The index of the column and row are used together as a key in a lookup table (known as the "jump matrix") which contains and information needed to decide if the corresponding cell is free or occupied.

As more and more rectangles are added, the grid gets partitioned in smaller and smaller pieces, and the number of cells, columns and rows increases.

If all rectangles were successfully packed inside the bounding box, its area is recorded and another bounding box, with smaller area is selected - and the procedure continues on that candidate. When all feasible bounding boxes are tested, the best one is used for the final packing. The rectangles' positions from this final packing is returned by `rpack.pack()`.

Extra grid lines have been added to the image below to demonstrate how these cells are created.



The algorithm is not documented more than this yet. Until it is, you will have to study the files `src/rpackcore.c` and `rpack/_core.pyx` for more details.

CHAPTER 5

Changelog

5.1 Version 2.0.1 (2021-05-13)

Bugfixes:

- `rpack.pack()` behaved incorrectly when the arguments `max_width` and/or `max_height` were used. For instance, it could return incorrect rectangle positions which made the rectangles overlap.

5.2 Version 2.0.0 (2020-12-29)

Added:

- Two new keyword arguments to `rpack.pack()` named `max_width` and `max_height` used to optionally set restrictions on the resulting bounding box.
- Exception `PackingImpossibleError`.
- Function `rpack.bbox_size`.
- Function `rpack.packing_density`.
- Function `rpack.overlapping`.
- Build dependency to Cython.

Changed:

- Improved `rpack.pack()` algorithm to yield higher packing density.
- Improved `rpack.pack()` time complexity; ~x100 faster compared to version 1.1.0.
- The rectangles are sorted internally in `rpack.pack()` so the input ordering does no longer matter.
- Renamed `rpack.enclosing_size` to `rpack.bbox_size`.

Removed/deprecated:

- Function `rpack.group()`. It is still available at `rpack._rpack.group()` but will be removed completely in the future.
- Old implementation of `rpack.pack()`. It is still available at `rpack._rpack.pack()` but will be removed in the future.

Other changes:

- Updated Sphinx documentation.

5.3 Version 1.1.0 (2019-01-26)

Added:

- New function `rpack.group()`.
- Sphinx docs.

Changed:

- Improved test cases.
- Improved error handling.

5.4 Version 1.0.0 (2017-07-23)

Added:

- First implementation.

Python Module Index

Index