Interactive animations of exact methods for the two-dimensional orthogonal bin packing problem

Mike Ghesquiere

Department of Computer Science, University of Western Ontario

E-mail: mghesqui@uwo.ca

ABSTRACT

The two-dimensional orthogonal bin packing problem (2OPP) poses the question of whether a set of rectangles (items) can be packed into a single, larger rectangle of fixed size (bin). This is an NP-complete problem. Two exact algorithms: LMAO and TSBP were presented by Clatiaux et al [1]. *This paper presents these algorithms again alongside a Python animation to better visualize their execution for teaching and improving these algorithms.*

1 INTRODUCTION

The two-dimensional orthogonal packing problem (2OPP) poses the question of whether a set of rectangles (items) can be packed into a single, larger rectangle of fixed size (bin). This is a generalization of the one-dimensional bin-packing problem and as such is NP-complete. Practical applications include modelling the layout of a newspaper [1], cutting a set of rectangular pieces from a sheet of steel [2], or for generating CSS sprites [3].

1.1 FORMAL DEFINITION

Formally, an instance of 2OPP is a pair (A, B) where A is a set of items a_i to pack. B is a bin with width W and height H with $(W, H) \in \mathbb{N} \times \mathbb{N}$. Each item a_i has height h_i and width w_i with $(h_i, w_i) \in \mathbb{N} \times \mathbb{N}$.

A solution for (A, B) consists of a function $\pi: A \to \mathbb{N} \times \mathbb{N}$ such that π maps each item a_i to the coordinates (x_i, y_i) of its bottom-left corner. π must additionally satisfy:

1. $x_i + w_i \le W$ and $y_i + h_i \le H$ for all a_i in A
2. $[x_i, x_i + w_i) \times [y_i, y_i + h_i) \cap [x_i, x_i + w_i) \times [y_i, y_i + h_i) = \emptyset$ for all a_i, a_j unique in A

That is, each item must be within the bounds of B and no two items intersect one another. Note that rectangles are defined to be left-closed and right-open along each axis as two as two adjacent rectangles should not be considered overlapping.

1.2 OUTLINE

In section 2, the classical left-most downward approach covered by Hadjiconstantinou [2], Martello and Vigo [4] will be discussed. Clatiaux's *left-most active only* (LMAO) [1] algorithm, a refinement of this strategy will then be introduced in section 3 along with proof of correctness and uniqueness of LMAO ordering. The two-step branching procedure (TSBP) which improves LMAO further is then covered in section 4. Section 5 will discuss the comparative performance of all three approaches. Finally, implementation details and documentation for the animation produced is covered in sections 6 and 7 before concluding in Section 8.

2 CLASSICAL APPROACH

2.1 LEFT-MOST DOWNWARD RULE

A classical approach introduced by Hadjiconstantinou [2] which utilizes a *left-most downward* (LMD) strategy. That is, all items are placed with their left edge against the left edge of the bin or adjacent to the right edge of another item. Similarly, with their bottom edge against the bottom edge of the bin or adjacent to the top edge of another item. In Figure 1, this strategy is applied to the left, (valid!) packing of a_1, a_2, a_3, a_4 to achieve the packing on the right.

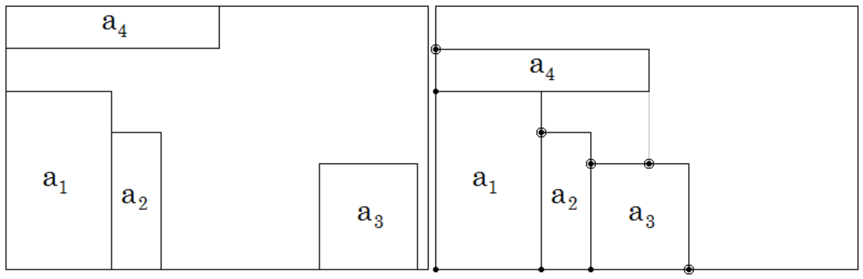


Figure 1 The left-most downward rule.

It should be obvious that this reduction still correctly solves the 2OPP problem. Let π_0 be a solution which does not satisfy LMD. Then there exists a_i such that either it's left or bottom edge is not adjacent to the bin or another item. So long as this is the case, there exists $\delta > 0$ such that a_i when placed at least one of $(x_i - \delta, y_i)$ or $(x_i, y_i - \delta)$ another valid solution will result. Let π_1 be such that δ is maximal.

Note that this procedure may *increase* the number of items which do not satisfy this rule. Let $\phi(\pi)$ and $\theta(\pi)$ be the minimal x_i and y_i respectively of all a_i which do not satisfy LMD. Then $\phi(\pi_j) \geq \phi(\pi_{j+1})$ and $\theta(\pi_j) \geq \theta(\pi_{j+1})$. After sufficiently many shifts π_N will be found with both $\phi(\pi_N)$ and $\theta(\pi_N)$ undefined (no unsatisfied a_i left). Inductively, π_N satisfies 2OPP as inherited from π_0 but all items now satisfy LMD. By this way, if a solution to 2OPP exists then, a solution which follows the left-most downward rule also exists. Thus without loss of generality we can restrict ourselves to only considering LMD solutions to 2OPP.

2.2 NON-UNIQUENESS

The classical algorithm using the left-most downward rule, involves selecting an item from A and selecting a LMD position. Then a depth first search through all LMD configurations for which the item is placed at that location. Although it appears to be originally presented by Hadjiconstantinou [2], Clautiaux denotes it as MV [1], respecting Martello and Vigo's treatment of the algorithm [4].

A significant flaw with this approach is the number of redundant configurations attempted. Consider the right packing given in Figure 1. To reach this configuration, a_1 must be placed first however, the following are all possibly enumerated independently:

$$(a_1, a_2, a_3, a_4) \quad (a_1, a_2, a_4, a_3) \quad (a_1, a_4, a_2, a_3)$$

Despite all being identical configurations of rectangles, it will possibly attempted three times if it does not allow a successful packing. Hadjiconstantinou [2] and Martello [4] both give several heuristics to avoid egregious cases. A graph theoretical model is also proposed by Fekete and Schepers [5].

3 LEFT-MOST ACTIVE ONLY (LMAO)

Clautiaux, Carlier and Moukrim [1] introduce their LMAO algorithm (Left-most active only) as an improvement to the MV strategy. Instead of considering all items and all active positions, *only* the left-most active position c, is considered. All items, (a_i, c) are considered as well as the possibility that the position is unused, $\bar{c}$.

This reduces the number of branches needed from $n \times l$ to $n + 1$ where n is the number of items and l the number of active points. Since each item will consume an active point and spawn at most 2 new ones (net $+1$ active points), the number of unused positions is bound by $n + 1$. Thus the depth of a LMAO tree is at most approximately twice the height of an MV tree which is more than made up for in the reduction to the recursion tree's width.

3.1 CORRECTNESS

The proof of correctness presented by Clatiaux [1], bootstraps heavily off of the correctness of MV. Any MV ordering can be re-ordered so that items are sorted by their left-most downward positioning, so wlog, assume $\pi_{MV} = (a_{i_1}, c_1), (a_{i_2}, c_1), \dots, (a_{i_n}, c_1)$ is such an MV ordering. We then construct π_{LMAO}, an equivalent LMAO ordering from π_{MV} iteratively. For each $\left(a_{i_j}, c_j\right)$, two cases can occur:

1. c_j is the left-most downward position. π_{LMAO} will also place a_{i_j} next in the sequence
2. c_j is not the current LMD position, c_k. Since π_{MV} is sorted, c_k is not used by any item and π_{LMAO} will place $\overline{c_k}$ next in the sequence and repeat this process until $c_k = c_j$.

Thus, any feasible configuration for which there is an MV ordering, there is also an LMAO ordering and LMAO solves 2OPP.

Algorithm: ***LMAO*** (B, unpacked, *packed*, *active*, *A*): (mandatory, *optional*)

Input: B is a bin consisting of ordered pair W and H. unpacked is a multi-set of points, (i.e. $\{(w_i, h_i)^j | 0 < j, 0 \le i \le n\}$) which have not yet been placed in B while packed is a set of rectangles (i.e. $\{(x_i, y_i, w_i, h_i)|0 \le i \le m\}$) already placed in B. active is the set of currently feasible positions and an associated axis (i.e. $\{(x_i, y_i)\text{: axis}|axis \in \{Null, Vert, Hor\}, 0 \le i < a\}$. Lastly, A is the total area of packed, unpacked and dummy rectangles.

Output: LMAO ordering of rectangles to pack the bin or [ ] if no such solution is possible.

```
1.   if ( packed = Null ):
2.        packed ← [ ]
3.   if ( active = Null ):
4.        active ← {(0,0): Null}
5.   if ( A = Null ):
6.        A ← total area of packed + total area of unpacked
7.   if ( unpacked = ∅ ):
8.        return packed
9.   if ( A > W∗H or active.is_empty( ) ):
10.       return [ ]
11.  lmao ← min( active ) projected onto packed
12.  if ( lmao.x = W or lmao.y = H or any rectangle in packed covers lmao ):
13.       return LMAO( B, unpacked, packed, active, A )
14.  for rect in unpacked:
15.       if ( rect fits in B and does not intersect any rectangles in packed ):
16.            newactive ← active + ( rect.left, rect.top ):Hor
                                   + ( rect.right, rect.bot ):Vert
17.            branch ← LMAO( B, unpacked−rect, packed+rect, newactive, A )
18.            if branch ≠ [ ]
19.                 return branch
20.  return LMAO( B, unpacked, packed+dummy, active, A+area of dummy )
```

*(dummy is an with size (**max**(1, **min**(active).x – lmao.x), 1) at location lmao)*

Method: ***Project***(point $= (x, y)$: axis, $R = \{(x_i, y_i, w_i, h_i)|0 \le i \le m\}$):

Input: Active point (with associated axis) and a set of packed rectangles R.

Output: Projection of point along axis onto next smallest edge of a rectangle.

```
1.   if axis = Null:
2.        return point
3.   elif axis = Vert:
4.        $x' \leftarrow \max_{r \in R} r.top$ if the line $x =$ point.x passes through $r$
5.        return ($\min(x, x'), y$): axis
6.   elif axis = Hor:
7.        $y' \leftarrow \max_{r \in R} r.right$ if the line $y = point.y$ passes through $r$
8.        return ($x, \min(y, y')$): axis
```

3.2 UNIQUENESS

As noted in section 2.2, an MV ordering is not unique. This can cause identical LMD configurations to be attempted by multiple branches. This cannot occur with LMAO orderings. For example, the configuration seen in Figure 1 can only be achieved through the sequence $(a_1, (0,0)), (a_4(0,5)), \overline{(0,6)}, (a_2, (3,0)), (a_3, (4,0))$.

We repeat the proof given by Clatiaux [1]. Let π_1 and π_2 be two distinct LMAO orderings and j be the first index such that $\pi_1(j) \neq \pi_2(j)$. One of two cases are possible:

1. $\pi_1(j) = (a_i, c)$ and $\pi_2(j) = (a_k, c)$ with $a_k \neq a_i$. a_i and a_k are both packed in the same coordinate so configurations produced by π_1 and π_2 are distinct.
2. $\pi_1(j) = \bar{c}$ and $\pi_2(j) = (a_k, c)$ or vice versa. c will not be used by any other item in π_1 and so a_k cannot be packed at c in the configuration produced by π_1.

Therefore π_1 and π_2 do not lead to equivalent configurations and so for any LMD configuration there is a unique LMAO ordering.

4 TWO-STEP BRANCHING PROCEDURE (TSBP)

The two-step branching procedure divides the placement of items into two methods with the goal of dismissing a large number of impossible configurations in the first. In the outer method, ***TSBP1*** the problem is relaxed to only determining feasible x-coordinates for all items. Once all items are placed at x-coordinates such that at no point the total height of items exceeds H, then an inner method ***TSBP2*** is launched to find y-coordinates for the x-coordinates given by the outer method. Since LMAO correctly solves 2OPP and any packing must follow the total height constraint, TSBP iterates over all feasible LMD solutions. Thus TSBP also solves 2OPP.

```
Algorithm: TSBP($B = (W, H)$, unpacked):
    Input: A bin $B$ with width $W$ and height $H$ as well as a set of unpacked items.
    Output: A packing of all items in unpacked into the bin.

1.  partialpack, state ← TSBP1(($W, H$), unpacked)
2.  while partialpack ≠ ∅:
3.      branch ← TSBP2(($W, H$), partialpack)
4.      if branch:
5.          return branch
6.      partialpack, state ← TSBP2( *state, skip ← True)
7.  return False
```

Method: ***TSBP1***($\underline{B = (W, H)}$, unpacked, *packed*, *hist*, *skip*):

Input: A bin B, a set of unpacked items, a set of packed items and positions, a "histogram" hist of total heights over the range $[0, W]$ and a boolean flag skip.

Output: A partial packing of B which assigns x-coordinates to all items in unpacked.

```
1.   if hist = Null:
2.       hist ← Histogram(W,H)
3.   if packed = Null:
4.       packed ← [ ]
5.   if unpacked = ∅ and not skip:
6.       return packed, state ← (B, unpacked, packed, hist)
7.   if hist = ∅:
8.       return False
9.   for rect in unpacked ∩ dummy:
10.      if rect = dummy:
11.          branch←TSBP1(B, unpacked, packed+(dummy, hist.$x_1$), hist+dummy)
12.      elif hist.$x_1$ + rect.x > hist.W:
13.          return False
14.      elif hist.fits(rect):
15.          branch ← TSBP1(B, unpacked−rect, packed+(rect, hist.$x_1$), hist+rect)
16.      else:
17.          branch ← False
18.      if branch:
19.          return branch
20.  return False
```

Method: ***TSBP2***($B = (W, H)$, partialpack, packed, active, A)

Input: A bin B, a set of partially packed items from ***TSBP1***, a set of packed items and positions, a list of currently active positions and total area A.

Output: A packing of items into B or [ ] if no such packing exists.

*(Identical to **LMAO**, with the following line changes)*

```
6.       A ← total area of packed + total area of ~~unpacked~~ partialpack
7.   if ( ~~unpacked~~ partialpack = ∅ ):
 ⋮
13.      return ~~LMAO~~ TSBP2( B, ~~unpacked~~ partialpack, packed, active, A )
13b. unpacked ← {$a_i | a_i$ ∈ partialpack, $x_i = lmao.x$}
 ⋮
17.          branch ← TSBP2(B, partialpack−rect, packed+rect, newactive, A)
 ⋮
20.  return TSBP2( B, partialpack, packed+dummy, active, A+area of dummy )
```

5 PERFORMANCE

5.1 LMAO

Clatiaux [1] found a huge improvement using LMAO over MV. According to their results LMAO solved many cases for which MV could not find a solution in feasible time. Furthermore, in cases where both algorithms found a solution, in all but one instance, LMAO had at least a speed-up of at least 20. There was one case where LMAO was outperformed by MV ($\epsilon = 20, n = 15, \text{feas.} = F$) but this took less 1 and 3 seconds to execute for MV and LMAO respectively. Since this is an easy case, it is somewhat understandable that a more sophisticated method (Clatiaux et. al also included many additional lower bounds in their implementation of LMAO), was outperformed by the naïve approach. These comparisons were not repeated.

5.2 TSBP

TSBP was also reported to give a huge increase in performance. It solved all instances which were solvable by MV or LMAO in near 0 CPU time as well as arriving at a solution for many instances unfeasible for the other two algorithms. It seemed to give comparable but complementary performance to Fekete and Scheper's [5] graph theoretic approach. Ignoring the three benchmarks where *Graph* could not find a solution, a solution was always found by at least one algorithm in less than 15 seconds. Clatiaux [1] concludes that "(the TSBP) method is complementary with the method of Fekete and Schepers, as it returns better results for several instances. It seems that the graph theoretical model is not the only one to lead to interesting results and other ways can still be explored to improve results for 2OPP".

6 IMPLEMENTATION

The LMAO and TSBP algorithms were implemented in Python 2.7 and using the following packages: **`random`** (for generating random benchmarks), **`heapq`** (for the HeapSet) **`csv`** (for importing item sets from file) and **`pygame`** (for the animation). It is also recommended to use on a system with the Segoe UI typeface (included in Windows 7 onwards) for optimal experience.

6.1 DATA STRUCTURES

6.1.1 ActivePoint

Originally implemented as a simple ordered pair, it became apparent that having an attached *axis* helps immensely. This axis can either by *Vertical*, *Horizontal* or *None* and denotes the direction which the point is *projected* when determining the left-most active point.

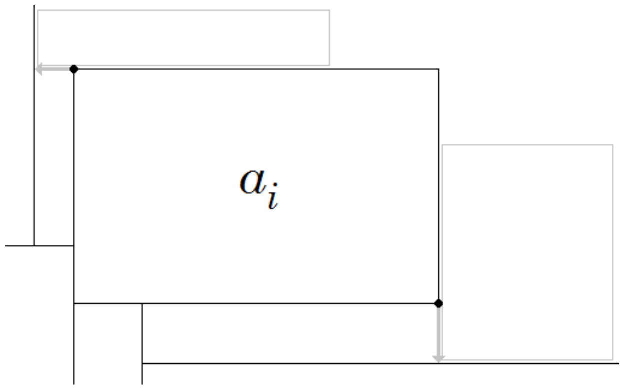


Figure 2 Each a_i will place two active points. One at its top-left corner with a horizontal axis and one in its bottom-right corner with a vertical axis. This is used to "project" the active point.

Whenever a non-dummy item a_i is added to the bin, two active points will be spawned: one in the top-left corner of the item with a horizontal axis and one in the bottom-right corner with a vertical axis. An item a_j which shares its bottom edge with the top edge of a_i will be placed with a position somewhere along the horizontal line projecting from this point.

6.1.2 HeapSet

A heapset is simply a heap which does not not permit duplicate items. This was used for the set of active points. Implementation of a heap from **`heapq`** was used as the base for this class. **`heapq`** uses a binary heap which is not ideal. Each iteration of LMAO uses 1 *delete-min,* at most 2 *insert*s per rectangle. As such with $O(n)$ *insert*s for every delete-min, it may be worthwhile to use a Fibonacci heap. This would decrease the time of heapset operations per LMAO iteration from $O(n \log l)$ to $O(n + \log l)$ where n is the number of unpacked rectangles and l the number of active points.

6.1.3 Counter

A *counter* is simply a dictionary with integer values. A key can be incremented whereby it will be added with value 1 if it does not already exist or the current value associated with the key will be incremented by one if it already exists in the dictionary. Decrementing a key will reduce the key at that value by one and then remove it if it's value is now 0. If the key being decremented is not in the dictionary, an exception will be raised.

The counter data structure is used by LMAO and the outer tree of TSBP to track unpacked rectangles. Only each unique rectangle will be attempted at a given location. Particularly in the case of *quilts* (see section 7.1.1), many duplicate items can occur.

6.1.4 Histogram

The outer tree of the TSBP method relies on "cutting each rectangle into strips, and placing these strips all at the same x-coordinate". At any given point, the height of these strips must not exceed the total height of the bin. The physical action of cutting rectangles is unnecessary.

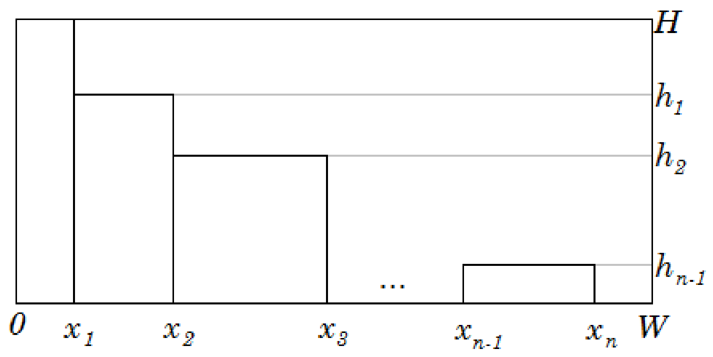


Figure 3 Visual depiction of the histogram $(W, H, b = [(x_1, h_1), (x_2, h_2), \dots (x_n, 0)])$

The *histogram*, data structure consists of an integer height and width as well as a list b of integer pairs. These integer pairs each represent an interval and are in the form (x_i, h_i) where x_i is the start of the interval and h_i is the interval's height. x_{i+1} will be the start of the next interval and also the end of x_i. At all points we maintain the inequalities:

$$0 \leq x_1 < x_2 < \cdots < x_n < W$$

$$H > h_1 > h_2 > \cdots > h_n \geq 0$$

Any interval with height H is omitted. Thus, should $x_1 \neq 0$ then this implies the height of the interval $[0, x_1) = H$. The last interval, x_n will track the height from the interval $[x_n, W)$.

6.1.4.1 Histogram.Fits(item)

In order to maintain the strict ordering of h_i, an item $a = (w, h)$ may only be added at x_1. Should $x_1 + w > W$ or $h > H$ then the histogram will never be able to accommodate a and this method returns *False*. This method will also return *False* if $h_1 + h > H$. However, in this case it may be possible to place a in the histogram either after a dummy is inserted or other items are added. If b is empty, then the bin histogram is completely full so this will also return *False*. Otherwise, the item a can fit into the histogram and this method returns *True*. This method has $O(1)$ time complexity.

6.1.4.2 Histogram.Add_Dummy()

Much in the same way as a dummy rectangle for the LMAO algorithm, a dummy item in a histogram is used to consume the active location and mark it as not used by any real item. This dummy item will effectively be an item with width $x_2 - x_1$ and height $H - h_1$ but practically, this is done by simply deleting the first element of b. This method has $O(1)$ time complexity.

6.1.4.3 Histogram.Add(item)

To add an item to the histogram, first ensure that the item fits using *Histogram.Fits(item)*. If the item does not fit, an exception will be raised. It will increment by h all intervals in the range $x_1 \dots x_1 + w$ and construct a point at $x_1 + w$ if it does not already exist.

7 DOCUMENTATION

7.1 PROBLEM SETS AND BENCHMARKS

7.1.1 Quilts

A quilt, is defined as an integer square made up of smaller integer squares. Of interest are *optimal quilts* with minimal *order* or number of smaller constituent squares. Solutions were found online [6] and present interesting examples for visualizing 2OPP. Originally posed as mathematical puzzles, these sets tend to form solutions which are not immediately visible to a user. All sets are tightly bound, positive instances of 2OPP. The smallest size squares were pruned from each solution so that each will use dummies. I believe this to be more illustrative of the algorithm as well being more difficult (branches with any dummy items cannot immediately be dismissed for exceeding total area). Quilts of size 14 to 36 are currently implemented and can be found in **`benchmarks`**. A quilt benchmark of size 18 can be constructed using:

```
>>> benchmarks.quilt(i=18)
```

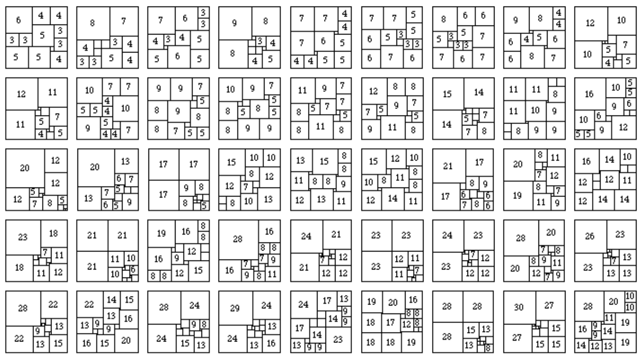


Figure 4 Optimal quilts of size 14 through 58.
Available online: http://www.mathpuzzle.com/MAA/06-Square%20Packing/mathgames_12_01_03.html

7.1.2 Randomized Items

Clatiaux [1] advocates using a randomized benchmark. This is constructed from parameters ε and n as follows. First a list of n random numbers such that their sum is $(1-\varepsilon)WH$ is generated. Then each element is factorized into a product of two numbers corresponding to w_i and h_i. This method is also available in **`benchmarks`** and can be called as follows:

```
>>> benchmarks.random(e=0.1, n=15, (20,20))
```

By default $(W,H) = (20,20)$ (the parameter is optional). I found this method to often create sums with prime terms. This leads to a disproportionate number of long strips. More often than not, these strips will exceed the dimensions of the bin. To eliminate these trivial cases, I recommend instead the use of:

```
>>> benchmarks.random2(e=0.1, n=15, (20,20))
```

…which will generate random items until all items are less than the dimensions of the bin.

7.1.3 User Generated Sets

Custom item sets can be created from a .csv file. The file should contain only the list of items with one item per row, with height and width separated by commas. A file in this form can be imported using:

```
>>> benchmarks.fromfile(filename="myitems.csv")
```

Note: bin size must be given separately for item sets from file.

7.2 PACKING

Methods to call ***LMAO*** and ***TSBP*** are available in `packing` and can be called as follows:

```
>>> packing.lmao((W,H), items, anim=True)

>>> packing.tsbp((W,H), items, anim1=True, anim2=False)
```

To use these methods with problem sets from 7.1 use `*` to unwrap the outputs, for example:

```
>>> packing.lmao(*benchmarks.quilts(18))

>>> packing.lmao(*benchmarks.random2(0.1, 15), anim=False)

>>> packing.tsbp(*benchmarks.fromfile("myitems.csv"), anim2=True)
```

7.3 DATA STRUCTURES

The data structures implemented were not intended to be used independently and as such are not fully featured. Nevertheless, the following are available in `util`:

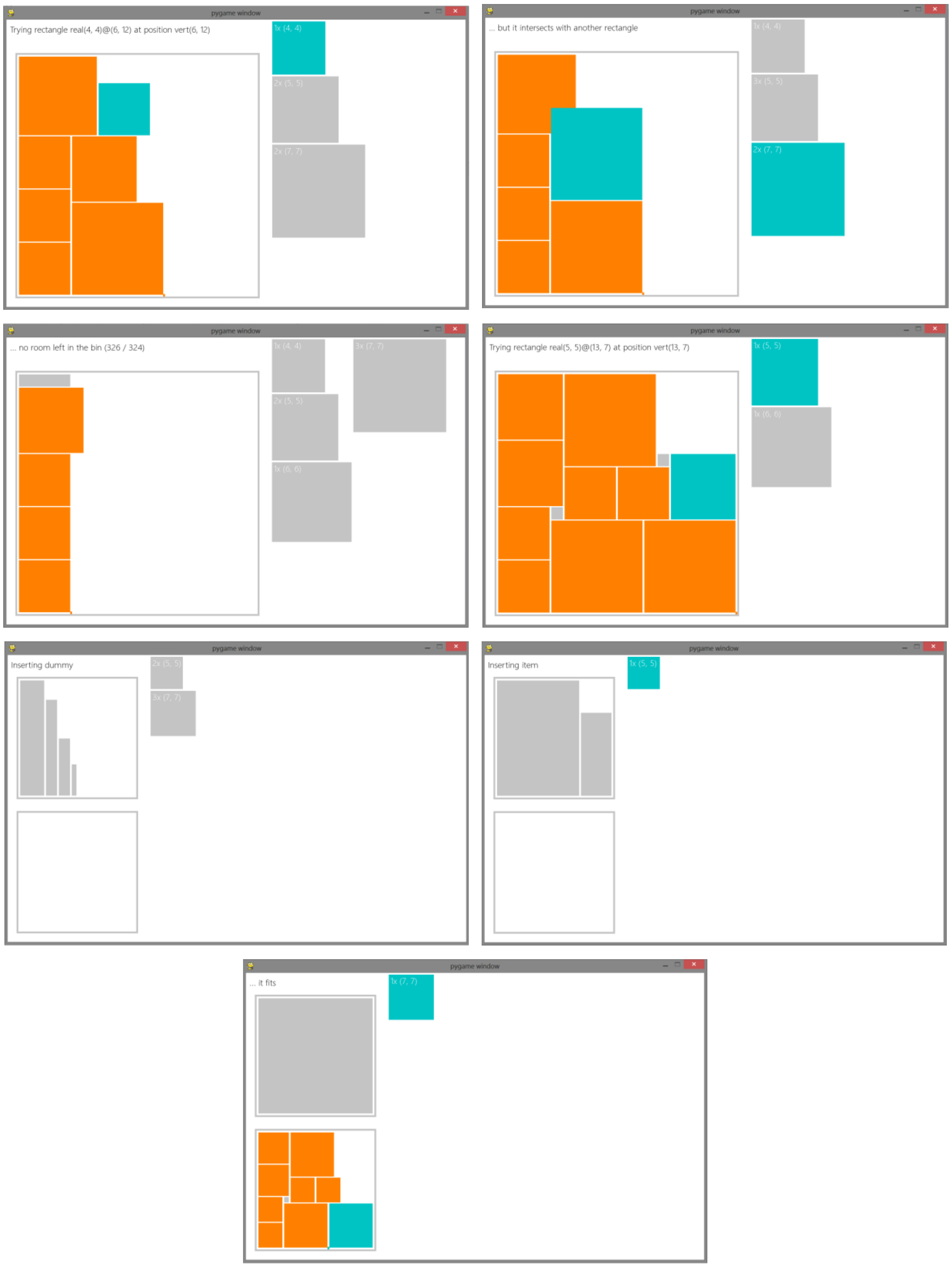
```
util.Rectangle
    __init__(self, (W,H), (X,Y), is_real=True)
    __str__(self)
    __repr__(self)
    __intersect__(self, other)
    top(self)
    bottom(self)
    right(self)
    left(self)
    covers(self, (x,y))
    size(self)
    pos(self)
    area(self)

util.ActivePoints
    __init__(self, x, y, axis)
    __repr__(self)
    __cmp__(self, other)
    pos(self)
    hits(self, rectangle)
    project(self, packed)
```

```
util.heapset
      __init__(self, q=[])
      __repr__(self)
      __nonzero__(self)
      __contains__(self, item)
      __iter__(self)
      copy(self)
      push(self, x)
      pop(self)
      peek(self)

util.counter(dict)
      incr(self, x)
      decr(self, x)

util.histogram
      __init__(self, (w,h), a=[(0,0)])
      __repr__(self)
      __nonzero__(self)
      fits(self, (w,h))
      add_dummy(self)
      add(self, (w,h))
```

8 CONCLUSION

An implementation of LMAO and TSBP were created in Python as well as an animation tool to visualize the packing approaches of these two algorithms. These may be useful for teaching as well as for analyzing cases in which they may be improved.

Screenshots



Trying rectangle real(4, 4)@(6, 12) at position vert(6, 12)

... but it intersects with another rectangle

... no room left in the bin (326 / 324)

Trying rectangle real(5, 5)@(13, 7) at position vert(13, 7)

Inserting dummy

Inserting item

... it fits

Figure 5 Sample of the animation running on quilt(18) for LMAO (top four screenshots) and TSBP (lower three).

REFERENCES

[1] F. Clatiaux, J. Carlier and A. Moukrim, "Exact method for the two-dimensional orthogonal packing problem," *European Journal of Operational Research,* vol. 183, no. 3, pp. 1196-1211, 2007.

[2] E. Hadjiconstantinou and N. Christofides, "An exact algorithm for general, orthogonal, two-dimensional knapsack problems," *European Journal of Operational Research,* vol. 83, no. 1, pp. 39-56, 1995.

[3] M. Perdeck, "Fast Optimizing Rectangle Packing Algorithm for Building CSS Sprites," Code Project, 14 June 2011. [Online]. Available: http://www.codeproject.com/Articles/210979/Fast-optimizing-rectangle-packing-algorithm-for-bu. [Accessed 4 April 2013].

[4] S. Martello and D. Vigo, "Exact Solution to the two-dimensional finite bin packing problem," *Management Sci,* vol. 44, pp. 388-399, 1998.

[5] S. Fekete, J. Schepers and J. Van der Veen, "An exact algorithm for higher-dimensional orthogonal packing," *Operations Research,* vol. 55, no. 3, pp. 569-587, 2007.

[6] E. Pegg, "Math Games: Square Packing," 1 December 2003. [Online]. Available: http://www.mathpuzzle.com/MAA/06-Square%20Packing/mathgames_12_01_03.html. [Accessed 4 April 2013].