WHEN I GROW UP, I WANT TO BE A 2D BIN PACKER

September 10, 2010 – 12:47 am

By *okay*

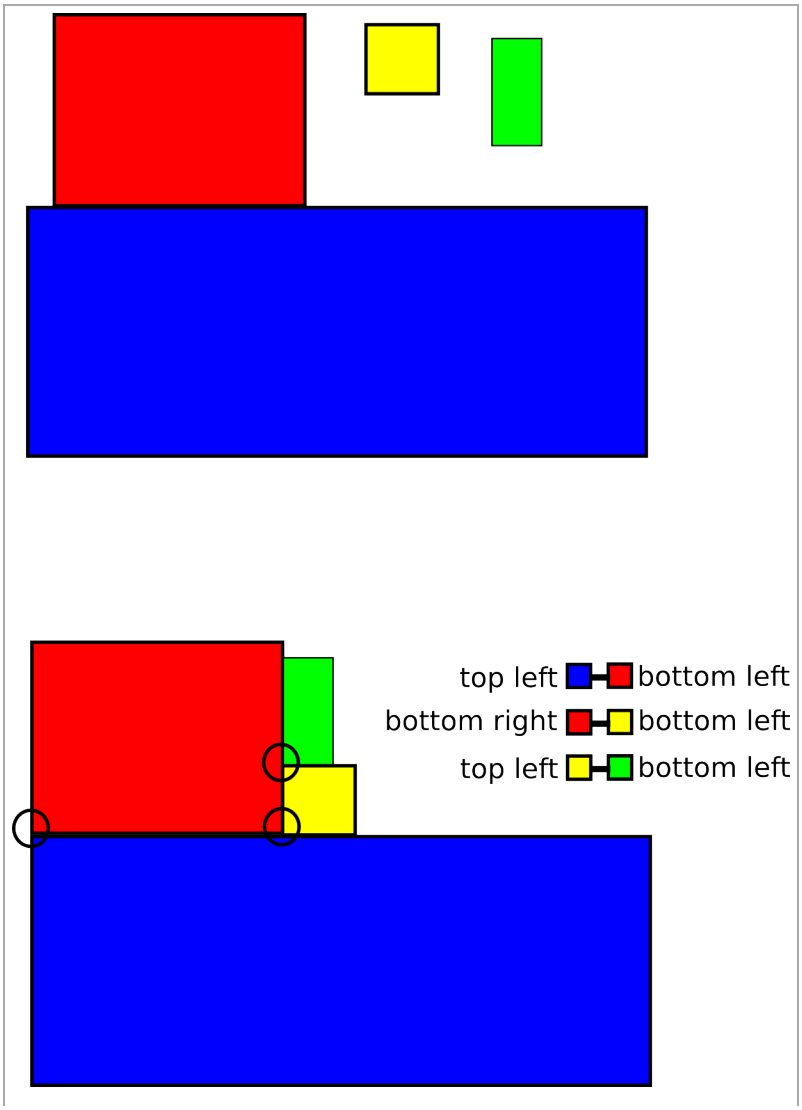
Posted in *Algorithms*

Tagged *rectangles*

Comments (3)

Earlier this month, I came across the Dropbox challenge: Packing your dropbox, a recruiting problem, based around packing rectangles in 2d as efficiently as possible. Simply stated, given up to 100 rectangles with varying heights and widths, what is the smallest rectangular area you can place them in?

Well, let's start off with the basics – Given that you have K rectangles, there are many, many, many ways of arranging them together on a 2d plane. How many? Well, to make things easy, let's assume that for any given arrangement of rectangles in a 2d space, you can always fill the same space (or less) by attaching all rectangles corner to corner with one shared edge. (Try it out – I believe it, but it's unproven).



Translating a free floating placement into corner attachments

The pseudo code for generating these placements of rectangles would be something like:

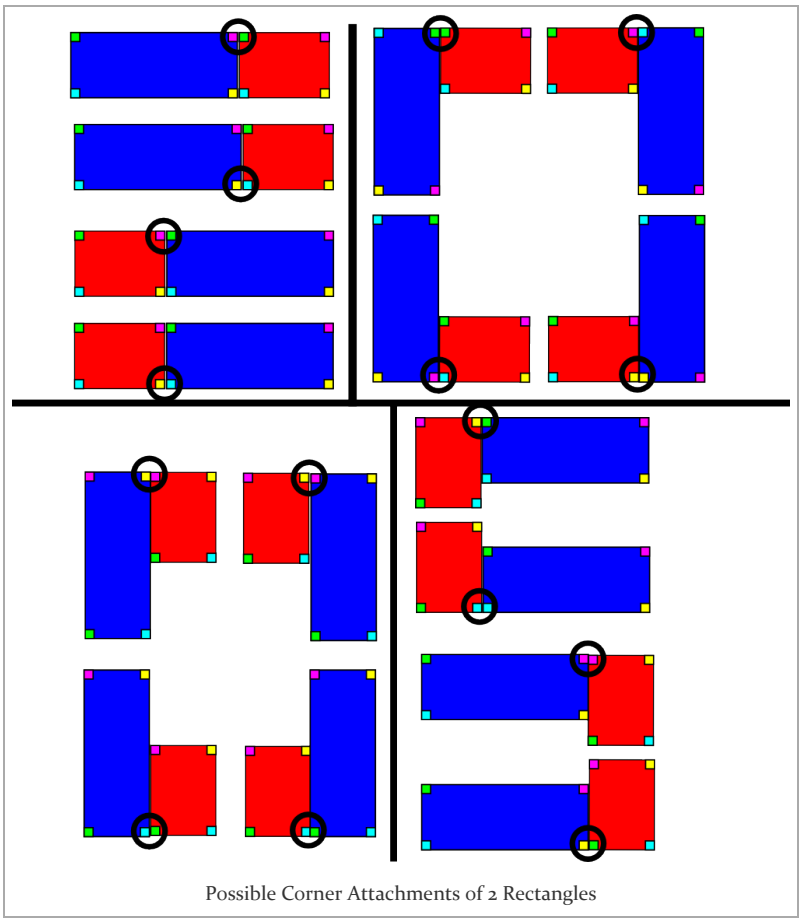
```
recursive_place:
  if all rectangles are placed:
    find area of bounding box of rectangles

  for each placed rectangle in rectangles:
    for each unplaced rectangle in rectangles:
      for corners A,B in corner combinations:
        continue if corner A is unavailable on unplaced rectangle
        continue if corner B is unavailable on placed rectangle

        place corner A of unplaced rectangle on corner B of placed_rect

        call recursive_place
        pick unplaced_rect up off the board
```

Notice that for any two rectangles, you can attach them together in one of 16 ways, as seen in the following illustration.



Possible Corner Attachments of 2 Rectangles

There are also K! (think permutations: nPn = n!) of ordering the rectangles, so given our assumption that two rectangles must be connected by a corner and share an edge, we can see that there are roughly K!*(16^K) ways of placing K rectangles on a board.

To put that in perspective, there are 3.9*10^18 ways of arranging 10 rectangles corner to corner.

We can cut this number down significantly if we make two observations when placing rectangles:

1. There is symmetry under reflection
2. There is symmetry under rotation

Notice how there are 4 groups of rectangles. Each group can be represented by one piece, if we account for the above symmetries and we can then simulate all placements just by using 4 possible corner attachments.

- Top Right – Top Left
- Bottom Right – Bottom Left
- Top Right – Bottom Right
- Top Left – Bottom Left

This brings the number of possible arrangement down to K!*4^K, a slightly more manageable number (only 3800 trillion for 10 rectangles), but still unpleasant to deal with. Trying having your computer count up to that number:

```
for x in xrange(3 800 000 000 000):
  continue
```

Not that fast, right? We can optimize our away past that, as well, if we make a few more observations:

1. Any solution with overlapping rectangles is unacceptable, so we can ignore any interim placements that have overlap
2. If the current area is greater than or equal to the best area we've seen for a full placement of rectangles, we know any further placements can only keep area equal or increase it so they are futile.
3. The order we connect rectangle corners doesn't really matter as long as all rectangles are placed.

For the first observation, it's relatively easy to notice that by not exploring any solution that has overlap, we ignore a lot of impossible possible placements.

For 2), this is trivially true and can cut down on examining duplicate solutions with the same area.

For 3), it's a little more difficult to see how we can take advantage of ordering. We were able to reduce the problem size by taking only a subset of the corners, earlier, but how can we use ordering to choose a subset of possible placements to examine?

If you know about memoization, you know that we can track any specific branch we've visited by creating some descriptive string that is then marked as visited when we are finished exploring it. We can then make sure we never visit a branch twice.

Well, if we decide to choose a unique representation of possible placed rectangles that accounts for ordering differences, we can also skip analagous branches to a branch we've already visited.

Many years of school have taught me all about sorting and this seems like the perfect opportunity: Sort the rectangles (by height, width, size, whatever) in order to remove duplicate orderings and create a representative string in order to keep track of branches visited.

Re-writing our code with these optimizations, we get:

```
recursive_place:
  if all rectangles are placed:
    return area of bounding box of rectangles

3)
  calculate representative string of rectangles
  if string is in already visited branches:
    return

  current min area = maximum int
  for each placed rectangle in rectangles:
    for each unplaced rectangle in rectangles:
      for corners A,B in corner combinations:
        continue if corner A is unavailable on unplaced rectangle
        continue if corner B is unavailable on placed rectangle
        place corner A of unplaced rectangle on corner B of placed_rect
2)
        if area of placed rectangles is greater than best area: goto cleanup
1)
        if there is any overlap caused by this placement: goto cleanup

        current min area = minimum of (call recursive_place) and (current min area)

        cleanup: pick unplaced rectangle up off the board

  add representation string to visited branches
```

That's not so bad... it's just a matter of implementing the routines needed to place rectangles, create a representation of a board and verify there is no overlap.

Here's a sample implementation in python

It accepts input in the format of the dropbox problem – i.e.

```
n
h1 w1
h2 w2
...
hn wn
```

Unfortunately, with all of these optimizations in place, it still takes my machine roughly 2 minutes to find the optimal placement for 10 rectangles using python. If you want to start finding solutions in 10+ space, it makes sense to start looking at approximation algorithms that do a good enough job, which will be covered in a future post.

If you can't wait for that, there's a lot of information on Wikipedia about the Bin Packing Problem.

3 COMMENTS

Getting Close Enough: [...] Picking up from the last post, we noticed at the end that once there are more than 10 rectangles, it isn't really feasible to find their optimal

Approximation Algorithms for bin packing | RecrecLabs Chatter wrote:
October 2, 2010 at 8:49 am

placement by examining all possibilities. In this post, I'll be going over one strategy for generating a good-enough solution. [...]

Anonymous wrote:
March 7, 2011 at 10:18 pm

One thing I would do as an easy and immediate optimization is to attach all rectangles that have a identical length height or width.

Kat wrote:
April 12, 2011 at 9:08 pm

@anonymous, how would that help? Putting boxes of equal lengths or widths together isn't always optimal.

LEAVE A REPLY

Your email is *never* shared.

Comment

Name

Email

Website

Post Comment

← Introducing Recrec Gen1

Finding variables declared outside __init__ in

PRODUCTS

Recrec: Gen1

PAGES

About
Contact
FAQ
Products
Status

ARCHIVES

April 2011
January 2011
October 2010
September 2010
August 2010

TAGS

algorithms cappuccino introspection python rectangles shedskin

CATEGORIES

Algorithms
Frameworks and Libraries
Uncategorized
Web Development

Search for:

Search

META

Log in
Entries RSS
Comments RSS
WordPress.org