Counting Tilings by Enlightened Brute Force

David desJardins

JIM Conference 2024

The problem

Given a region and a set of tiles, we want to answer questions like:

1. Is it possible to tile the region with this set of tiles?
2. How many ways are there to tile the region with this set of tiles?
3. In a random tiling with these tiles, what is the probability that tile X appears in position P?
4. In a random tiling with these tiles, how many of each tile are used, on average?

Jim wants to gather empirical data to formulate hypotheses for rigorous analysis.

In some cases, large computations are required to get enough useful data.

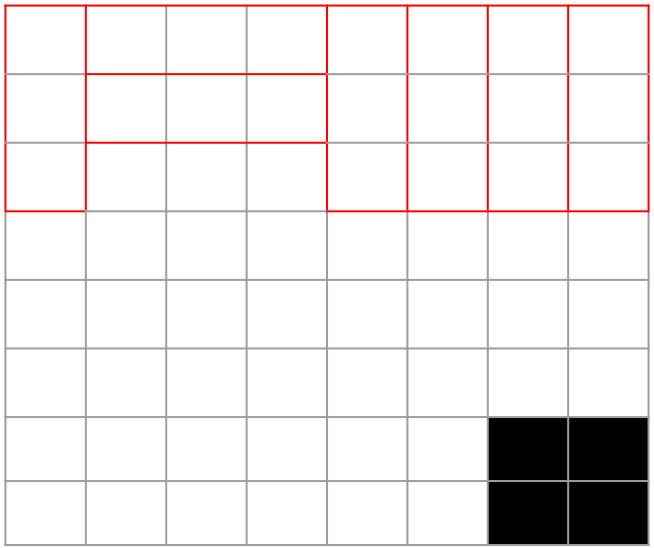
Wide variety of possible parameters

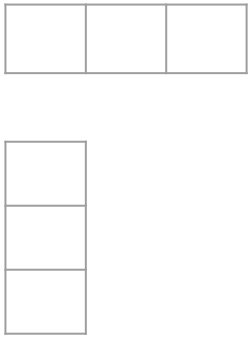
1. Few or many tiles
2. Small or large regions
3. Connected or disconnected tiles
4. Square grid or other grids
5. Highly constrained or loosely constrained

My goal was to produce a general and flexible program, rather than optimize for one particular set of parameters.

Mapping into a square grid suffices for many problems. For example, a hex grid is equivalent to a square grid with every other square removed. Triangular grids can be warped into a square grid.

Dynamic Programming (exp-time in cells, not in counts)



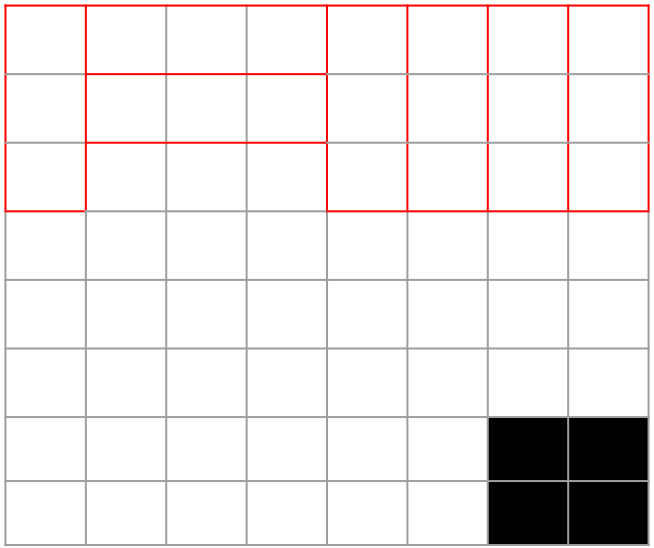


The most general algorithm for counting tilings (and associated computations) is dynamic programming.

Dynamic programming counts tilings of partial regions.

For example, how many ways are there to tile the indicated region of 60 cells, with these 2 tiles? (Answer: 850.)

Dynamic Programming (continued)

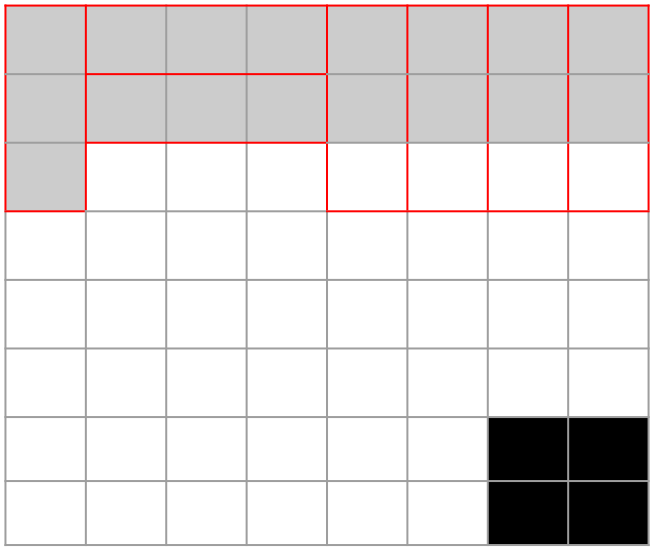


There are 3 tilings of the indicated subregion (2 others, plus this one).

Each of those can be completed to all 60 cells, in 36 different ways.

The value of dynamic programming is that it lets us count 3 groups of 36, rather than 108 separate tilings.

Dynamic Programming (continued)

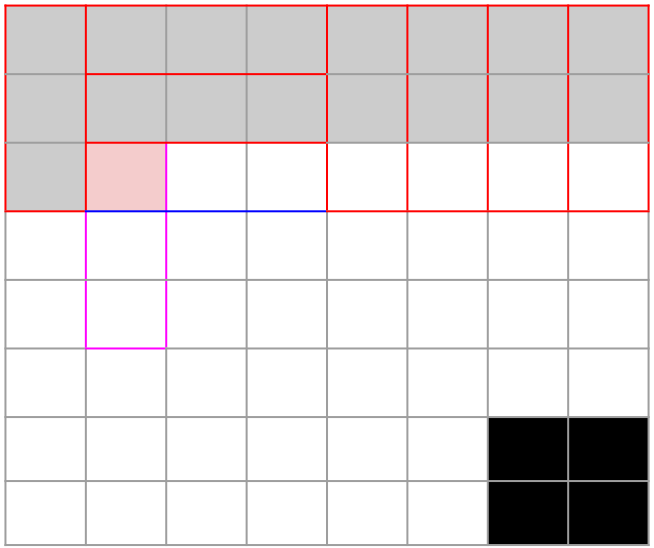


We need a set of regions such that every tiling will be counted exactly once.

One way to do this is to specify a set of cells, and include all tiles (and only those tiles) that overlap that region.

For example, if we cover the 17 grey cells, the illustration is one way to do so.

Dynamic Programming (continued)

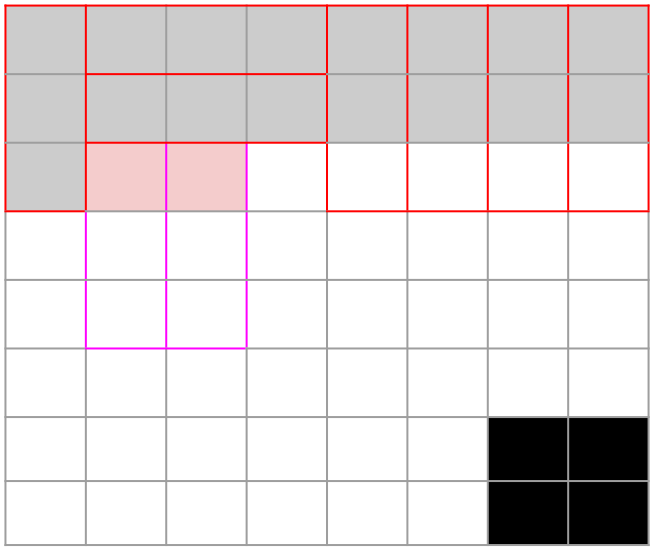


This is not always the most efficient decomposition, but it's very general and flexible.

If we want to extend this region to cover 18 cells, there are two ways to do that.

Adding one tile at a time, we eventually cover the whole specified area.

Dynamic Programming (continued)

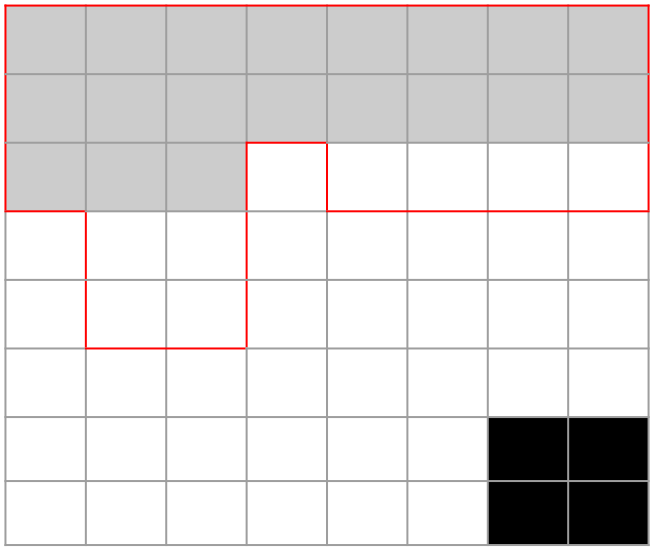


If we add the horizontal tile, then we have already covered the 19th cell.

But, if we add the vertical tile, then there is exactly one way to cover the 19th cell.

We are not looking ahead, so we could even reach a situation where there's no way to cover the remaining cells.

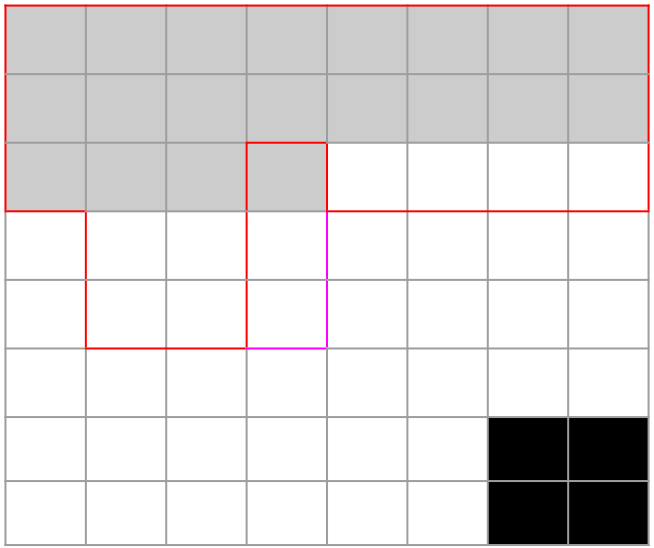
Dynamic Programming (continued)



At each step, we have a list of regions that cover the current subset of cells (plus other cells), and a count of how many ways to tile that region.

For example, for this specified region, the number of ways to tile it is still 3.

Dynamic Programming (continued)



At the next step, there are 19 ways to tile the expanded region.

3 come from adding the indicated tile to the previous region.

The others will be produced using other decompositions.

Some haven’t been generated yet! That’s ok, we will produce them later. They will be extensions of regions we have now.

Dynamic Programming (continued)

- At each step, we have a list of cells that must be covered, and other cells may be covered.
- For each such region, we want to store a count of how many ways that region has been generated.
- If the same region is generated in two different ways, we want to combine the counts. This is the essence of dynamic programming!
- Figuring out which regions are even possible may be a hard problem!
- If there are N cells that might be covered, there might be a lot fewer than 2^N possible regions! So storing the counts as a 2^N long table would be very inefficient.

Dynamic Programming (continued)

- Solution: we store a list of regions, each with the corresponding count.
- This avoids any precomputation: we can simply generate the regions as we go.
- For some tiling problems, there would be more compact representations. But this always works, and isn't much less efficient.
- We need a bitmap of the cells that the region covers, but it doesn't need to include the grey cells, which are always covered.
- Two remaining problems:
  1. Generate new regions from previous regions.
  2. Combine counts when we generate the same region.

Dynamic Programming (continued)

- When we add a new grey cell, there are two possibilities:
  1. Our region already covers that cell.
  2. Our region doesn't cover that cell.
- In case 2, we need to add a new tile.
- If we are adding grey cells top-to-bottom, left-to-right, there's only one way to position each possible tile (its uppermost leftmost cell must cover our spot).
- For each such tile, the one possible way to position it may or may not be legal, i.e., it might extend outside of the area to be tiled, or it might overlap the region that we're trying to extend.

Dynamic Programming (continued)

- So we have an algorithm for generating the new list of regions, given the previous list:
  1. For each region in our current list that already covers the new cell, there's no change to it or its count.
  2. For each region in our current list that doesn't cover the new cell, check each tile in our tileset. If adding that tile is legal, that generates a new region, with the same count.
  3. The union of all of these (region,count) pairs is our new list.
- But this new list may have duplicate regions. We need to combine these, otherwise we don't get the benefit of dynamic programming!

Dynamic Programming (continued)

- If we were storing the counts in an array, it would be obvious when duplicates occur, and easy to add them. But, as stated before, that could be very space-inefficient.
- We could sort the list at each step, which would allow us to find and combine duplicates. But sorting is relatively expensive, especially if we want to do it in-place, without additional memory. (The constraint on the size of tiling problems we can solve is often memory storage, rather than compute time. Doubling the memory is significant.)
- There’s a key trick, which makes this whole process super efficient!

Dynamic Programming (the trick!)

- Let's store our list of regions in "lexicographic order", i.e., if we think of each region as a bit string of what cells it covers, write those out in order.
- Key Observation: If we pick a tile, and add that tile to each region in a lexicographically ordered list, then the list is still lexicographically ordered!
- Deleting some regions from the list (because the addition of that tile is not legal) also preserves lexicographic order!
- Therefore, given our list at time *T*, we can generate *K+1* lists at time *T+1*, one which consists of those regions that already cover the new cell, and one from each possible tile that we might add to the regions that don't.
- The new list is then the merge of these *K+1* lists.

Dynamic Programming (trick continued)

- Merging lists is a very efficient operation.
- At each step, we have a “pointer” into each of the *K+1* lists.
- Whichever pointer points to the lexicographically first region, will be the next one to be added to the list we are generating.
- If several pointers point to the same region, those can be combined, and their counts added.
- Finding the lexicographically first of these *K+1* regions can in fact be done in *O(log K)* time rather than *O(K)* time, using a heap -- a minor but useful improvement. (This algorithm can scale to hundreds or thousands of tiles!)
- We don’t actually need to store the *K+1* lists, just generate them as we go.

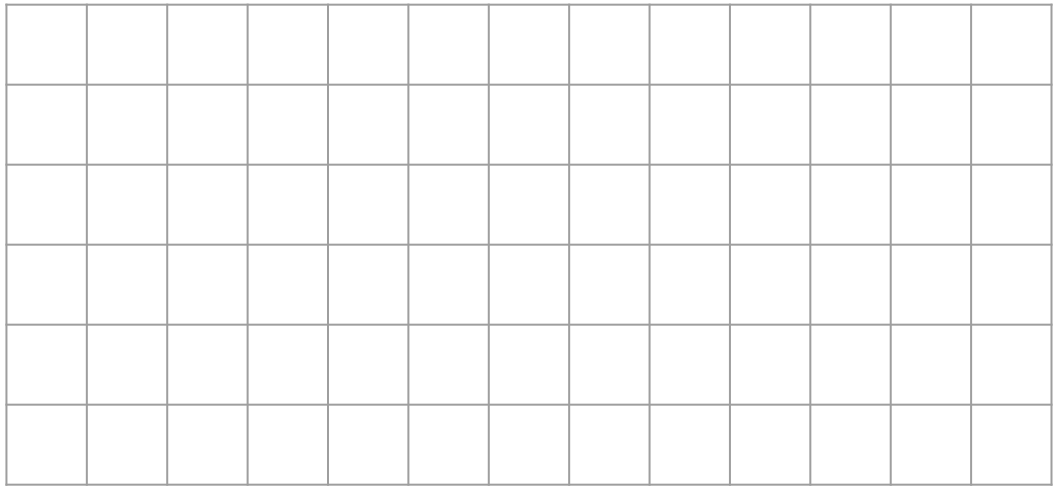
Implementation Details

- Regions are represented by bitstrings. The maximum length of those strings may vary over the course of the algorithm. It's easiest to just store these as variable length byte sequences.
- Counts start out small and may get big. Again, it's easiest to store these as variable length byte sequences.
- Multiprecision arithmetic is, of course, essential. But there's only addition in this algorithm, no multiplication, so that's much easier.
- Sometimes it's useful to compute counts *mod p* or *mod* 2^n, rather than as integers. Or even just in the two-element semiring, if all we care about is whether tilings exist or not.

Even Fancier Implementation Details (Future?)

- It could be significantly more efficient to store the sorted list of regions as delta encoded (i.e., the difference as integers from the previous region).
- It could be possible to overwrite the current list as we generate the new list, reducing the total memory requirement. Once all of the pointers have advanced past a given point in the list, we can free up that memory.
- Running the same algorithm both forward and backward can let us to count tilings that use each tile in each location. (A lazy approach is to just run on the region with that tile deleted. But doing that for all tiles and all locations is more work.)
- The ordering of cells is somewhat arbitrary. It might be possible to predict which ordering is going to be more efficient (in time and/or space).
- We could assign weights to tiles, and it wouldn't be much harder to compute weighted counts (either sum of products of weights, or sum of sums of weights).

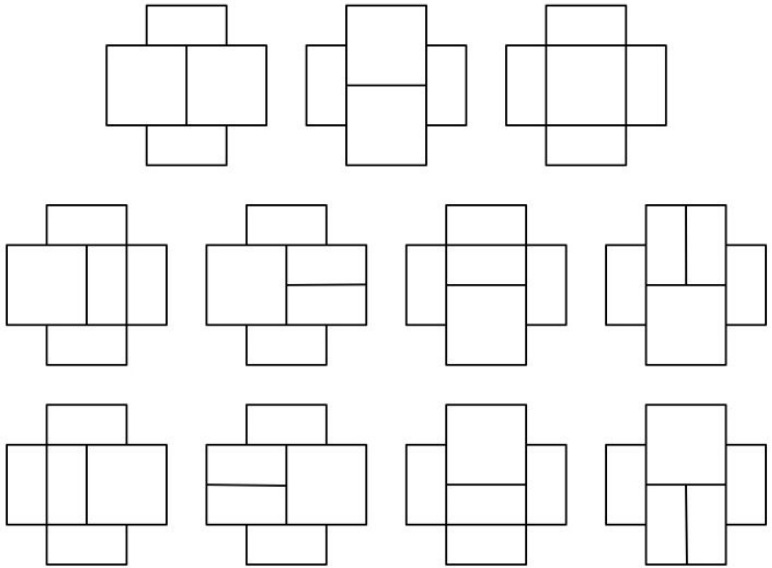
An Example



The 13x6 rectangle has 23494 tilings by 3x1 trominos.

Jim Propp is 23494 days old.

Another Example



Tilings of Aztec diamonds by dominos and square tetrominos (turns out to have lots of 2-adic properties):

n	$M(n)$
0	1
1	3
2	19
3	293
4	10917
5	996599
6	222222039
7	121552500713
8	162860556763865
9	535527565429290907
10	4318205059450240425083
11	85475498697714319842817853
12	41511861754637978889455121442 21

Goals

- Document source.
- Make source code and executable generally available.
- Implement some of the fancy algorithmic features.
- Support alternatives to square grid.
- Support “Exact Cover” problem (replace translation of tiles with arbitrary subsets of cells).

Questions?