Code Jam 2018 - World Finals

Analysis: Two-Tiling

The restriction on tile size limits the problem to just 35 distinct tiles, and therefore "only" 35 * 34 / 2 = 595 possible cases. We might consider whittling away at this number via individual insights — e.g., any case with a 9-cell tile and an 8-cell tile must be impossible, since the least common multiple of 9 and 8 is 72, which is larger than the allowed board size. We may even be tempted to cut out some paper tiles and do some solving by hand, but if so, we will quickly discover that this is harder than it seems, and it is particularly hard to be sure that a case is impossible. We need to either write a Master's thesis on properties of n-ominoes, or take an exhaustive approach.

We cannot simply find all of the ways to tile part or all of the board with each tile, and then intersect those two sets; there could be too many possible tilings to generate (e.g., 2^{64} for the 1x1 tile). Even if we could come up with special-case solutions to get around this issue for enough of the smaller tiles, we would still waste a lot of time enumerating sets that are obviously incompatible with the other tile.

Instead, we can use a backtracking strategy that takes advantage of our constraints, using the placement one type of tile to guide how we place the other type. The strategy does not have to be lightning-fast; we know that every case will appear in the test set, so we can enumerate and solve them all offline and then submit once we have an answer for each one. However, it must be fast enough to generate all 595 answers before the World Finals end!

Let us number the cells of the 8x8 board in row-major order, which is the same order in which our backtracking search will explore. We can translate each tile (and all of its rotations and reflections) around the 8x8 board to exhaustively find all possible placements. Then, we can assign each of those placements to the lowest-numbered cell that the placement uses. For example, one placement of the 2x2 square tile can cover cells 1, 2, 9, and 10 of the board, and the lowest-numbered of those cells is 1, so we assign that placement to cell 1. Now we have a helpful map from each board cell to the tile placements that fit there. Notice that it would be redundant to list our example 2x2 square tile placement under cell 10. For example, by the time our search reaches cell 10, it will have already considered that same placement at cell 1, so there would be no need to consider it again.

Then, starting in the upper left cell and going in row-major order, we try to build a set of cells that can be tiled by both tiles. For each cell c, we decide whether to fill it. If it is already covered by a previous tile placement, we must fill it; if we decide to fill it, and it is not already covered by one or both types of tiles, then we have our search explore all of the tile positions assigned to that cell in our map above.

Since the board is 8x8, we can use the bits of a 64-bit data type to represent each tile's coverage of the board, and we can use bitmasking to quickly see whether desired cells are occupied, or generate one state from an existing state. It is also very quick to check whether two states are equal; if this is ever the case, we stop immediately, step back through our backtracking to figure out exactly how to divide that set up into tiles of each type, and output that tiling. If the backtracking process finishes without finding such a set, then the case is impossible.

This solution turns out to be fast in practice, running in seconds. Intuitively, small tiles are more "flexible" and it is easier to find solutions quickly for cases involving them, whereas large tiles fill up the board's cells rapidly and do not open up as many choices. That is, the recursion tree for the backtracking is wide but short in the case of small tiles, and tall but narrow in the case of large tiles.

This approach does not necessarily find minimal sets of cells, but it can generate some artistically pleasing shapes and tilings, and/or puzzles that really would be fun to solve! Here are a couple examples generated by one of our solutions:

These angular tiles create a solution with a surprising mix of dense and holey areas!

```
aaa.bbb. AAA.BBB.
a..cbcb. A..CCCB.
aaacbcb. ADDDECB.
...ccc.. ...DEC..
..dddeee ..FDEEEG
..d....e ..F....G
..dddeee ..FFFGGG
........ ........
```

We hope you "loved" this problem! Notice that a test case is not necessarily any easier when the two tiles differ in only a very small way.

```
aa...bb. AA...BB.
aaa..bbb AAA..CBB
acacdbdb DDACCCBB
.cccddd. .DDCCEE.
.cceedd. .DDFFEE.
..eefff. ..FFEEG.
..eeeff. ..FFGGG.
....ff.. ....GG..
```