untitled.md

James Wang (jjw6wz)
Tiling Puzzle Solver Using Backtracking

Source code: https://github.com/jameswang14/tiling
Demoed to Siva on 12/13

Implementation and Algorithms

Parsing

To parse the ASCII files I implemented floodfill (BFS) to find the pieces and board. Scan the input files until a non-whitespace character is hit, then floodfill, storing the character and coordinates and replacing the original character in the input with whitespace. For each piece we'll then have a list of tuples in the form of (x, y, character) e.g [(3, 28, @), (4, 28, @)]. To reconstruct the piece, we find the smallest x and y coordinate and shift everything by those values - our piece earlier would become [(0, 0, @), (1, 0, @)]. Iterating through each point we place characters appropriately and put whitespace everywhere else. To find the board, we pull out the piece with the largest area.

Tiling

The backtracking algorithm works by find the first empty cell (from left-right, top-down order) that shouldn't be empty and trying to fit pieces into it (with or without rotation/flipping). When trying to place a piece, we check if it's valid: the piece doesn't go out of bounds, intersect with another piece, or mismatch with the target board configuration. After a valid placement, we then branch off of the new board and update the number of pieces available. If flipping and rotation is allowed, we generate up to 8 branches for each piece. At the start of each branch, we check if the running board configuration matches the target board - if so we then check if it's unique and add it to the solutions if so.

GUI

I built a simple web-based GUI, wrapping the python code into an API using Flask. The GUI is made up of HTML elements, where input names are populated from the server, and flags can be set by the user. The solve request calls the main backtracking code from the server. The server will respond either with all the solutions, one solution (based on the flag), no solutions, or invalid input. The web UI then displays a message or draws the solution using a canvas.

Optimizations

A majority of the optimizations came from fine-tuning termination conditions, since _backtrack() and methods inside of it make up 99%+ of the total run-time. In general, the less branching we had to do, the better performance would be. The biggest optimization came from knowing when to prune the current branch. Because of the way we search for an empty spot, that is left-right and top-down, and the way we place tiles, that is the top-leftmost non-blank tile, if we ever can't fill a spot we can prune since all subsequent tilings will be unable to fill it i.e down the line no tiling placement should fill a hole above and to the left of it.

Symmetry

I noticed that without rotation or flipping, the algorithm ran orders of magnitude faster, which is unsurprising given that run time is exponential, where the exponent is a function of the number of branches. If we can reduce the number of rotations/flips we need to search, we'd get huge performance gains since we'd cut down on the number of branches. The only way we can really do this is checking for symmetry - if a piece is symmetrical, we cut down the number of flips and rotations we need to search down in half. We pre-process the pieces and list which symmetries it has. Then during runtime, we only check rotations and flips if there isn't symmetry.

Board Symmetry and Early Isomorphic Detection

You could also get some optimization from utilizing board symmetry to avoid searching down isomorphic solutions. If placing a piece results in a board configuration that is isomorphic to any board in the current solution set, we can cut the branch since all isomorphic branches would already have been explored. I didn't have time to implement this properly, but it could provide significant speedup since it can cut-down searches by 1/2 in cases where boards have flip and rotational symmetry. (Siva proposed an interesting idea, using a hash to represent board orientation in a similar way to Rabin-Karp's substring search algorithm to make detection super-fast.)

Board Rotation

Our algorithm searches top-down left-right; this means it prefers taller boards as opposed to wider ones. Suppose we were solving a 3x20 board, and we placed a piece on the top-left that would eventually be found invalid e.g there's a unfillable pocket at (1,0). This would only be discovered after trying to fill the entire top row, which is 20 tiles long. Instead, if we can solve a 20x3 board, assuming rotation is allowed, we'd be able to find invalid placements much faster. Testing this on the 3x20 input, this optimization improved runtime very significantly - solving all solutions for the 3x20 board originally took >1000s; now it only takes ~3 seconds.

Python Optimizations and Other Optimizations

Some optimizations were python specific. For example, the built in copy.deepcopy() is extremely slow but was used in the initial implementation. I wrote a custom _deepcopy_matrix() and _deepcopy_3d() that improved speeds by about 100x, a significant improvement especially considering how many times deep copies are needed. I also different python implementations - python 2.7, python3.6, and pypy. Pypy ended up being the fastest, with a 5x speedup over the other two in some cases (2.7 appeared to be marginally faster than 3.6 surprisingly).

When placing a tile, I originally had bounds checking inside the loop i.e if the loop every iterated to be out of bounds then return. Checking this before iteration however, saved a lot of time since it only requires inexpensive mathematical computations and no looping.

I also keep track of two board states, curr and curr_unique, where curr matches the original board (e.g XOXOXO for checkboard), and curr_unique tells you where each piece has actually been placed (e.g 11122). This information is actually redundant because all tiling placements should be legal, but the purpose of curr is to take advantage of python's == operator speed. If we tried to compare curr_unique to board, we have to compare element-wise which I found to be a bit slower.

Benchmark

Puzzle Name	Number of Solutions	CPU time for 1 solution	CPU time for all solution
checkerboard.txt	1558	1.8994410038	2960.47746396
IQ_creator.txt	6	0.023059129715	3.21392679214
trivial.txt	1	0.00162982940674	0.0013039112091
partial_cross.txt	20	0.0825531482697	21.728924036

Puzzle Name	Number of Solutions	CPU time for 1 solution	CPU time for all solution
pentominoes3x20.txt	2	0.613763093948	2.83569097519
pentominoes4x15.txt	368	0.271947145462	98.8304212093
pentominoes5x12.txt	1010	0.398168087006	495.204017878
pentominoes6x10.txt	2339	0.0606160163879	1865.80987287
pentominoes8x8_corner_missing.txt	5027	0.4973599910736084	2904.07937512
pentominoes8x8_four_missing_corners.txt	2170	0.3919868469238281	1898.43961232
pentominoes8x8_four_missing_diagonal.txt	74	0.745181798935	64.9669299126
pentominoes8x8_four_missing_near_corners.txt	188	1.20336699486	687.030138969
pentominoes8x8_four_missing_near_middle.txt	21	0.168095111847	128.746465206
pentominoes8x8_four_missing_offset_near_corners.txt	54	2.29781603813	224.567842007
pentominoes8x8_four_missing_offset_near_middle.txt	126	0.612210035324	114.136837006
pentominoes8x8_middle_missing.txt	65	0.339186906815	273.276759863
pentominoes8x8_side_missing.txt	1288	0.217298984528	918.520232916
test1.txt	48	0.0125780105591	37.4087078571
test2.txt	0	0.0710370540619	0.05770516395
thirteen_holes.txt	2	0.0610740184784	0.647748947144
lucky13.txt	time_out	1.1308248043060303	time_out

Having more solutions seems to heavily impact performance. A large reason for this might be late isomorphism checking (see "Board Symmetry and Early Isomorphic Detection" section)

Instructions to Run

To run: `python -m flask run` and navigate to localhost:5000. To view solutions, go to localhost:5000/solution_viewer