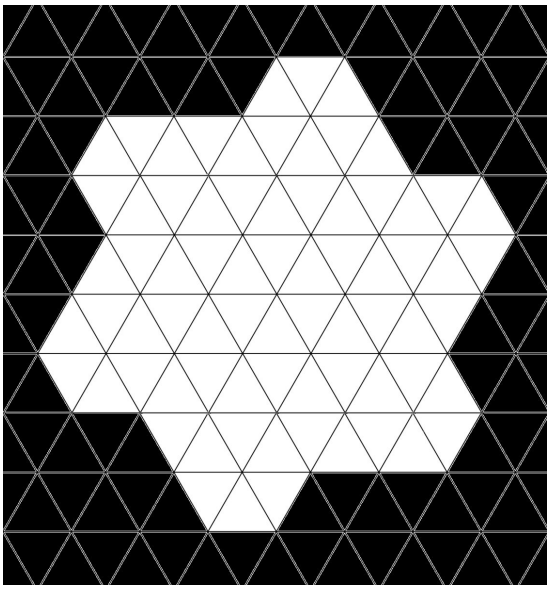
Haley’s puzzle

Solving a puzzle with python.



By Lode Ameije 05/2020, lodeameije@gmail.com.
Puzzle name unknown (please let me know), bought in Croatia.
All files and code: https://github.com/hyperlode/triangular_puzzle_solver.git

The puzzle challenge: Fit all the pieces in the board.

Abstract

An obscure wooden puzzle with 12 unique pieces was begging to be solved. The search space looks too big to solve by hand. A computer program was created to run through the entire search space.

The puzzle has similarities to Pentonimo puzzles. This is however based on a triangular grid, whereas Pentonimo is on a square grid. In the end a solution was found relatively quickly which suggests that there are more than 3000 solutions in this 200 billion possibilities search space.

Intro

I found this puzzle on the coffee table of my good friends' girlfriend Haley. After playing with it for some minutes, I right away felt that this was going to be very hard. But, for some reason, I was triggered. And when I'm triggered, I won't rest before I solve it, or before I haaave to give up for my own sanity. It might be Impossible to solve by hand, but we have these beasts called computers. And I wanted a little challenge. My coworker suggested I make a little document for every project I do. This way I have a tangible result. I tried to go into quite some detail here, in order to show it to non experienced interested friends who are wondering how on earth I spend my free time.
Closest puzzle found on the internet "KINGZHUO Hexagon Tangram". But, this is not it.

I don't like this puzzle

After a week of being into this riddle, I still don't see a reasonable road to manually solving this puzzle in a reasonable time. Physical puzzle solving is all about reducing the search space. Instead of using brute force, one has to come up with some clever tricks to dramatically decrease time to solve the puzzle. That's all there is to puzzle solving. So, for now, I could not find a method to limit things down to a human level. Sure, were some tremendous optimizations, but it's still prohibitively time expensive to find a solution by hand.

It is however very interesting to see that all twelve pieces are unique. They also seem to be the only possible pieces to be made out of six adjacent triangles. The base board is a very nice symmetrical pattern. I'm curious to know who came up with it, and if there are similar puzzles with less or more triangles.

Unless I'm wrong of course and I miss the simple intelligent solution! I'm still very curious!

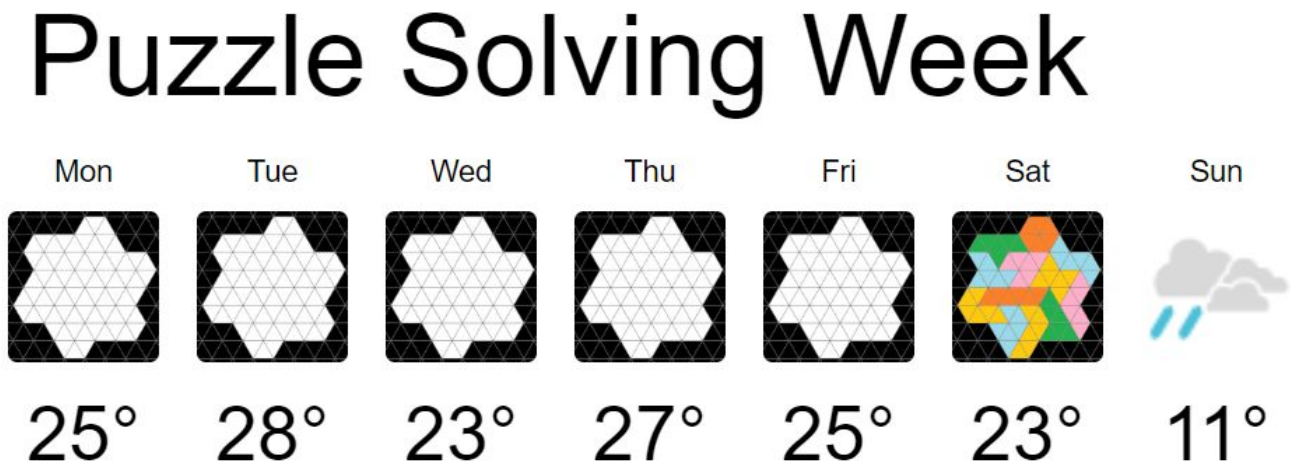


Fig.: I lost a week of my life

Solution

Approach

No way this was going to work by hand. But as it was still hoped that this puzzle was not unbearingly hard. Python was chosen as a programming language. While slower in execution, it was faster to program because of it's more simple syntax and because the author was more experienced in it than superior speed languages like C++.

Setting up the environment

Triangular grid

This puzzle has a triangular grid. As opposed to a square grid. They can be intimidating at first, but once properly orientated, they're not that different from square grids.

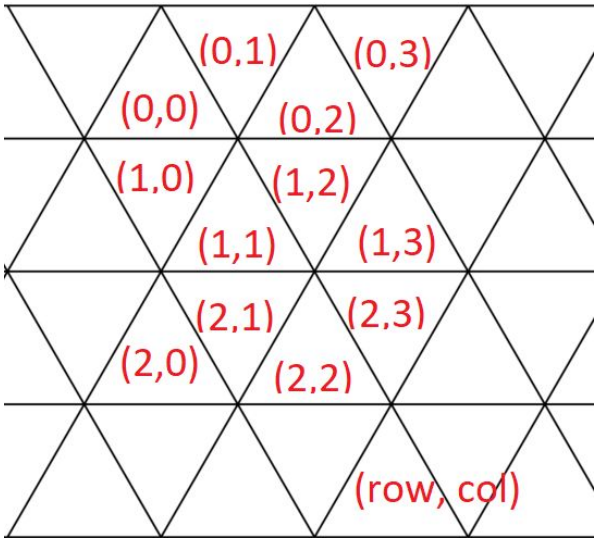


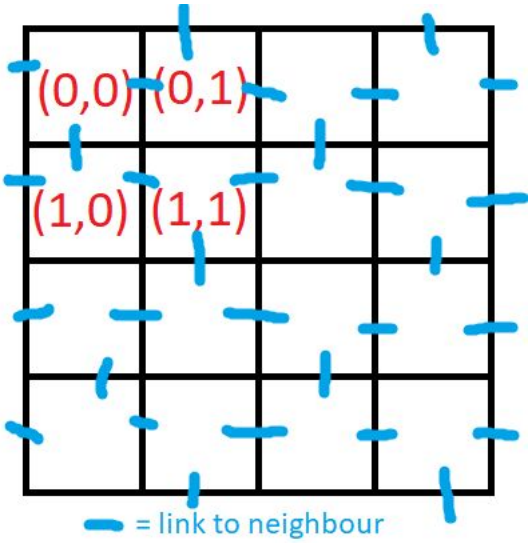
Fig. : Cell coordinates in a triangular grid (row, column). Choosing an upfacing triangle as (0,0) was arbitrary.

There are two kinds of triangles (=cells) in this grid. Upwards facing and downwards facing.
By choice, the (0,0) triangle is an upwards facing one.

Transformations for translating, rotating and mirroring in this grid are a bit tricky at first. It was relevant to set it all up to calculate all the orientations of the pieces and to move the pieces over the board.

This grid in computer memory was represented by a dictionary with each cell "infill" value addressed by its (row, column) identifier.

Accessing neighbours



In order to deal with neighbours, we can represent the grid in a normal matrix. Every cell has a left and a right neighbour. But upward pointing cells have no top neighbour and downward facing ones have no bottom neighbour. This behaviour was actually not needed in this implementation. As all algorithms worked purely by overlaying grids.

Puzzle pieces

The puzzl pieces are all unique.
The pieces all fit on the triangular grid. There is no distinguishable front or back side, so the pieces can be flipped too. Flipping is the same as mirroring (=chiral forms). These flipped pieces have again 6 orientations.
It appears like these are the only pieces that can be found containing 6 adjacent cells in the grid (not checked thoroughly)

For some pieces there are identical forms, for others, there are none.

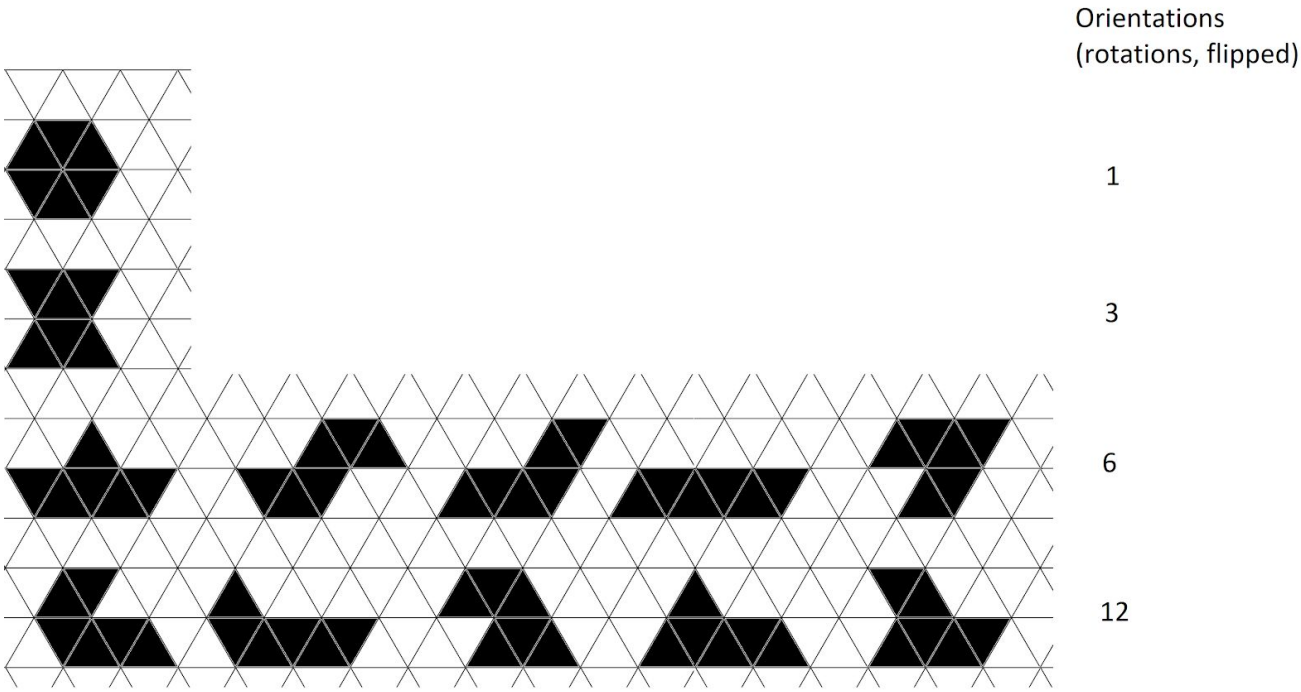
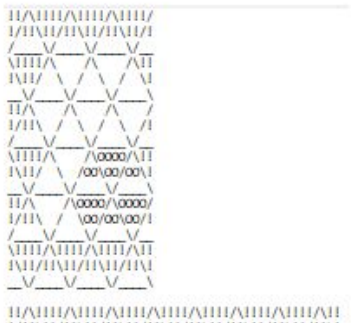
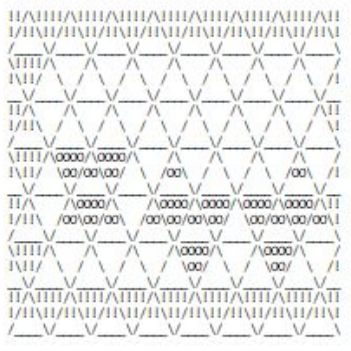
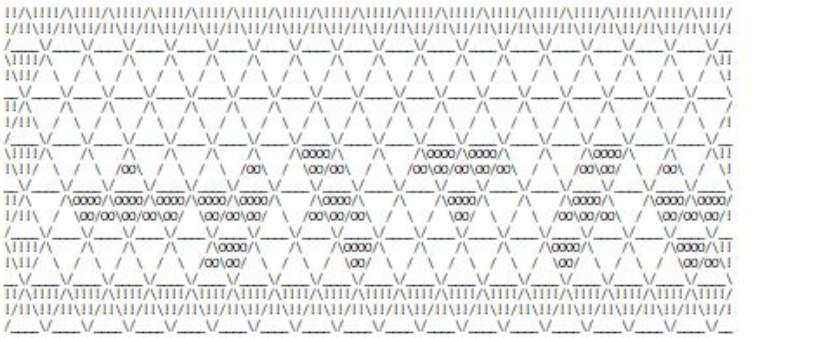
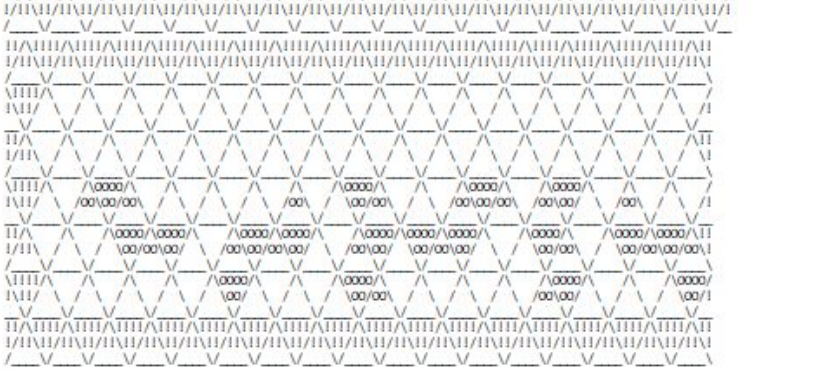


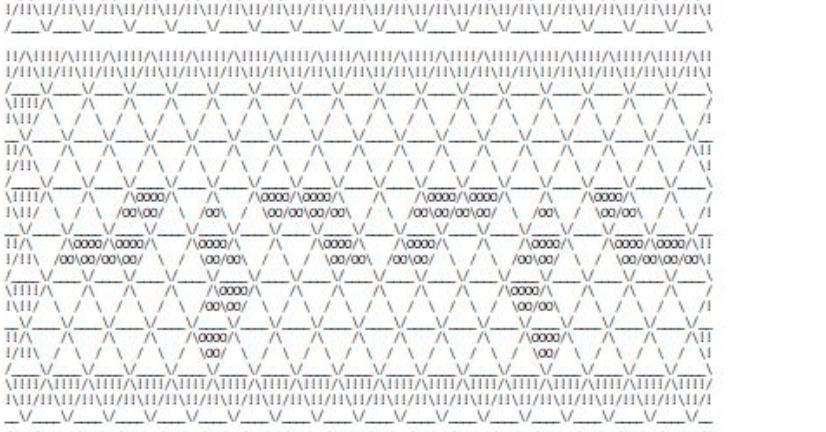
Fig. All Puzzle pieces (top left to bottom right) named: hexagon, hourglass, boat, tectonic (shifted hexagon), boomerang, bar, spaceship, snake, hockeystick, UNKNOWN , mountainview and UNKNOWN

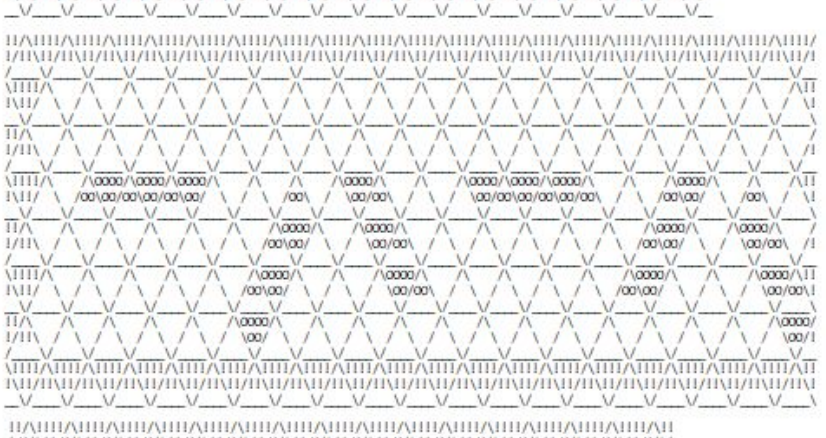












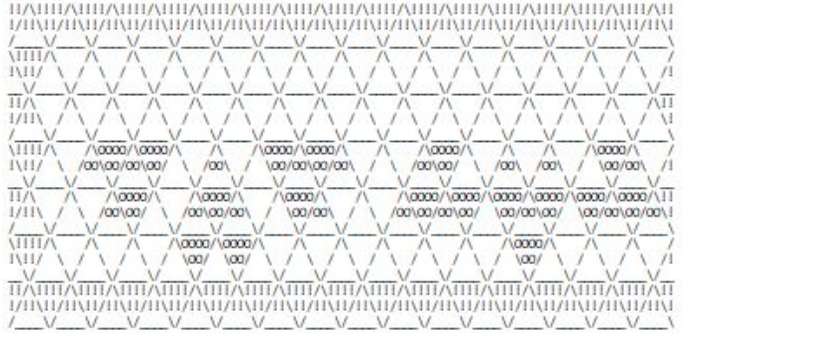


Fig. Puzzle pieces with all their orientations part 1

Fig. Puzzle pieces with all their orientations part 2

The board

The board has a well defined shape in which all pieces fit exactly. The board has rotational symmetry. There is a clear upside. So, no flipping. That makes a total of six orientations.

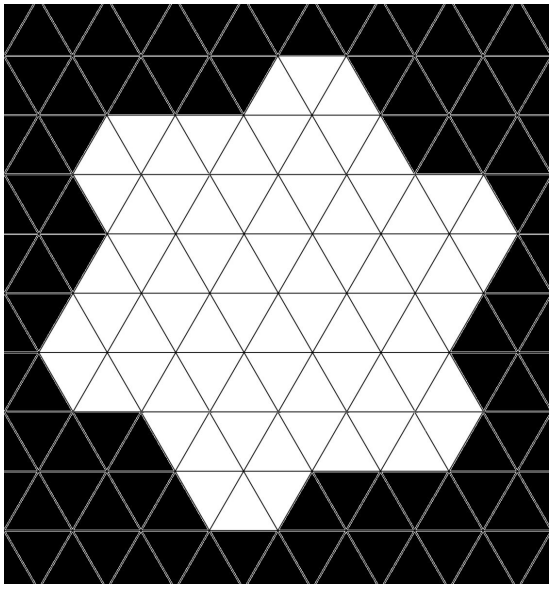


Fig Puzzle playing field. white= open for pieces, black = not open for pieces.
Puzzle board shape is obtained from the puzzle.
The board fits on the same triangular grid as the pieces. In order to create the required shape for the puzzle, the cells that are on the grid, but no border cell values are pre filled with a “CELL_NO_GO” value. When pieces overlap this,

Edge cases

In order to limit edge case time handling when fitting pieces, the board matrix is extended beyond the needed size. The extended border cells have a special “CELL_EDGE” value. When pieces overlap the edge when put down on the board, there is no error thrown. Instead, all overlayed cells values from hoe board with the piece are added up. By the value of the cells (CELL_EDGE + CELL_ON), it’s clear that there was a collision with an edge. This illegal state can than be treated like any other illegal state. No need for border checking or error catching.

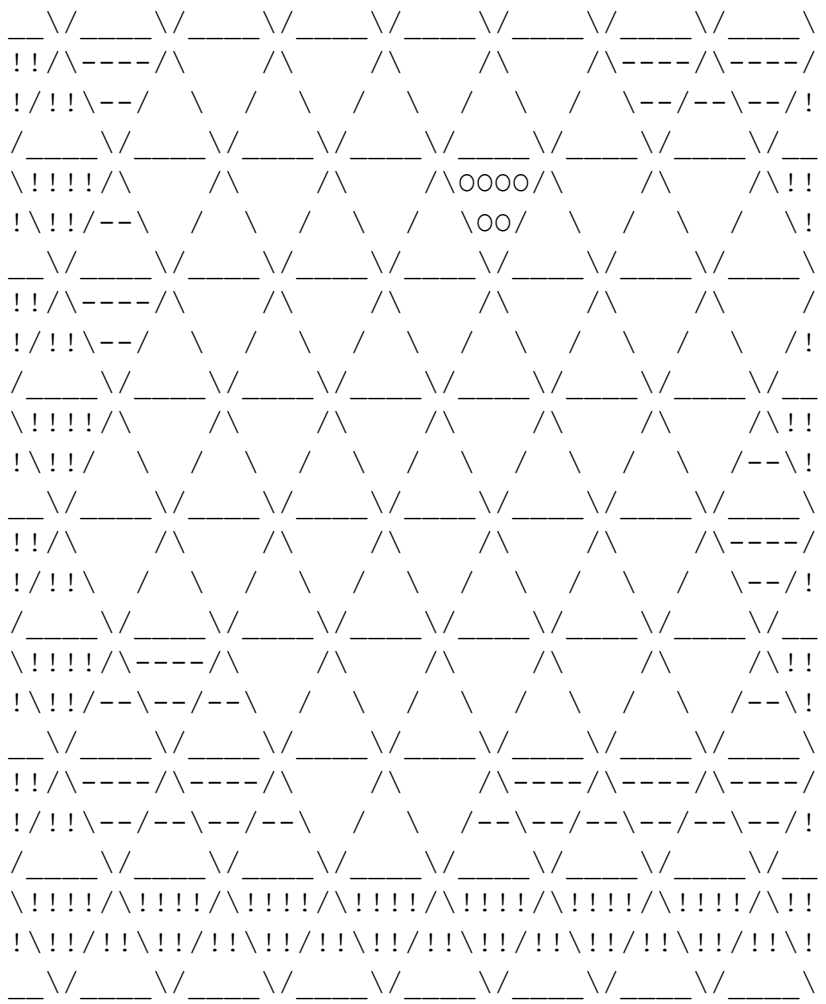
[figure board with cells]

Work was done to visualize the board in the terminal. It took some time, but it for sure paid off during debugging.

[PRINTED board in the terminal]

```
“!” = CELL_EDGE
“ “ = CELL_OFF
“O” = CELL_ON
“-” = CELL_NOGO

!!/\!!!!/\!!!!/\!!!!/\!!!!/\!!!!/\!!!!/\!!!!/
!/!!\!!/!!\!!/!!\!!/!!\!!/!!\!!/!!\!!/!!\!!/!
/____\/____\/____\/____\/____\/____\/____\/__
\!!!!/\----/\----/\----/\    /\----/\----/\!!
!\!!/--\--/--\--/--\--/  \  /  \--/--\--/--\!
```



Solving the puzzle

While there are probably more methods to choose from, this is the used workflow.

Global solution

For every piece

- add piece
- check if board is illegal
- check if board has potential (check for spaces that will never be able to contain other pieces)

Do until all pieces fit on the board.

Adding a piece

Limiting the search space

Unavoidably, we have to put pieces on the board to check them. But which piece? Where do we start? Don't just take a random piece. Remember how pieces can have multiple orientations. Each orientation represents another shape for the computer. Board X is not the same as board X rotated 90 degrees for our way of working (more on that later). So, every orientation is in reality a different shape for the algorithm to work with. Some pieces have more orientations than others. In the solution, every piece is present. We don't know in which orientation though. The more orientations a piece has, the less chance there is for a particaular orientation to actually make it in the final result.
Ideally you want to test the pieces with the most probability of making it in the solution to be tested first.

There is one special piece. the hexagon. We know for sure which form it will take in the solution as it has only one shape, no matter its orientation. After eliminating rotational symmetry, there are only five positions of this piece on the board.
This dramatically reduces our search space.

The logical next step would be to take the next pieces with minimal orientions. This one has 3 forms. It would be possible to search for all possible boards with these two pieces located on [hmm todo]. In reality, this was not done.

The "most probable pieces first" was abandoned for the "easy checking if a board has a future approach.

Setting up the base boards.

Five base boards are created, with the hexagon 'built in'. This not only decreases our search space enormously, it also gets rid of the board's rotational symmetry. This is the single most effective optimization that was found.

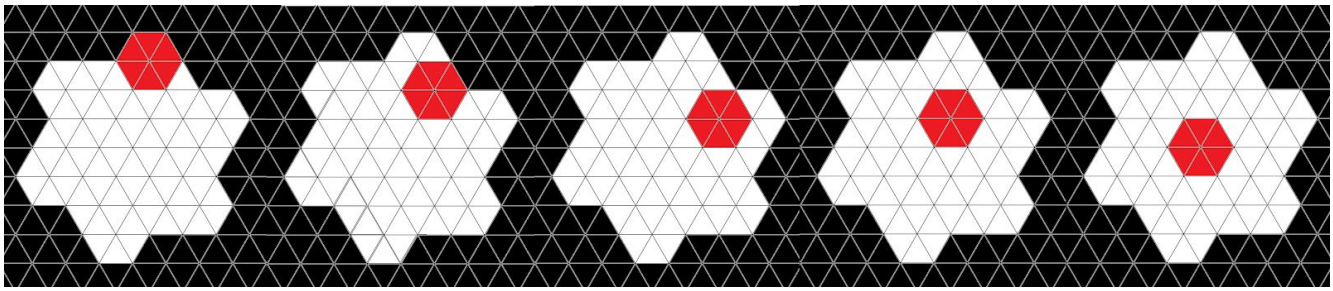


Fig. All possible positions of the hexagon in a solution. Note that for the board with the central hexagon, there is still rotational symmetry.

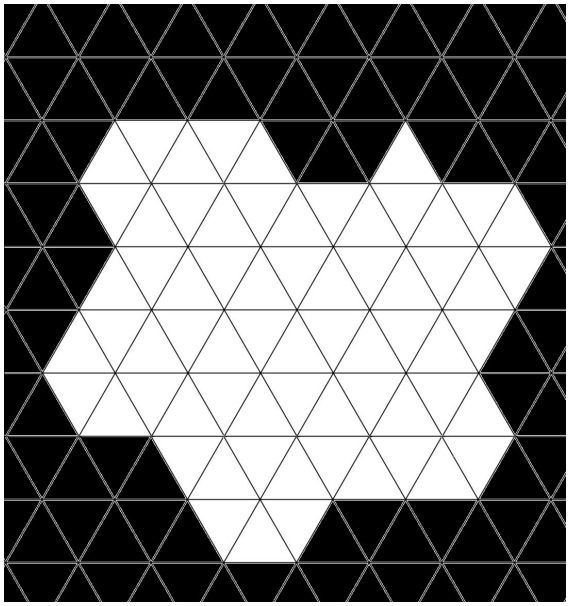


Fig. A baseboard with hexagon shape effectively having become part of the board

Check if board is legal

Adding a piece to the board simply means overlaying the piece grid with the baseboard grid. The overlapping cell values are summed up. Checking for illegal boards simply because a task of looking for specific values in the board. As soon as ‘a cell’ has a non valid value, we can discard the whole board right away. This is very fast.

i.e. CELL_ON + CELL_ON = 1 + 1 = 2 = CELL_PIECE_COLLISION_WITH_PIECE.

Constants used for cell values

```
CELL_EDGE = 8  # for programatorical reasons (to have no exception on the edge of the grid)
CELL_NOGO = 4  # not allowed --> i.e. board edge
CELL_ON = 1
CELL_OFF = 0

CELL_PIECE_COLLISION_WITH_EDGE= 9
CELL_PIECE_COLLISION_WITH_NOGO = 5
CELL_PIECE_COLLISION_WITH_PIECE= 2
```

Collision with edge

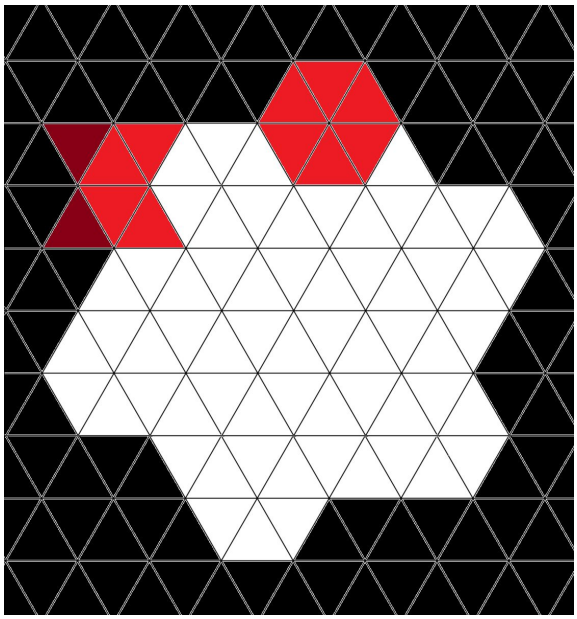


Fig. Illegal! Piece colliding with edge.
collidion cell values are CELL_PIECE_COLLISION_WITH_EDGE or CELL_PIECE_COLLISION_WITH_NOGO

Collision with another piece

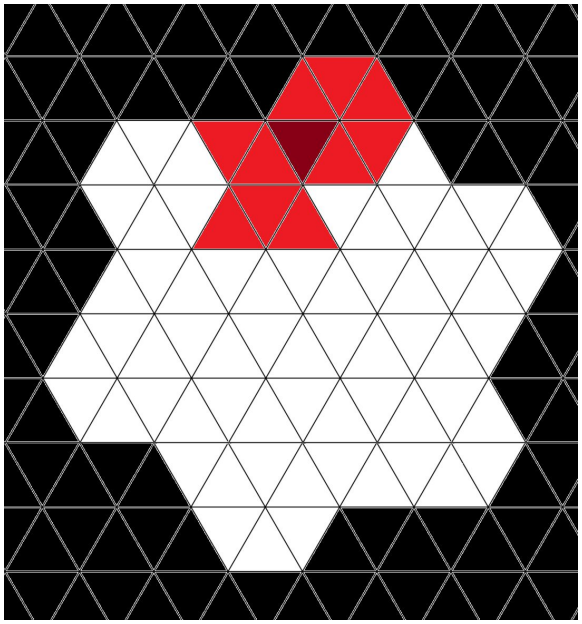


Fig. Illegal! Piece colliding with another piece.

Check if board has a future

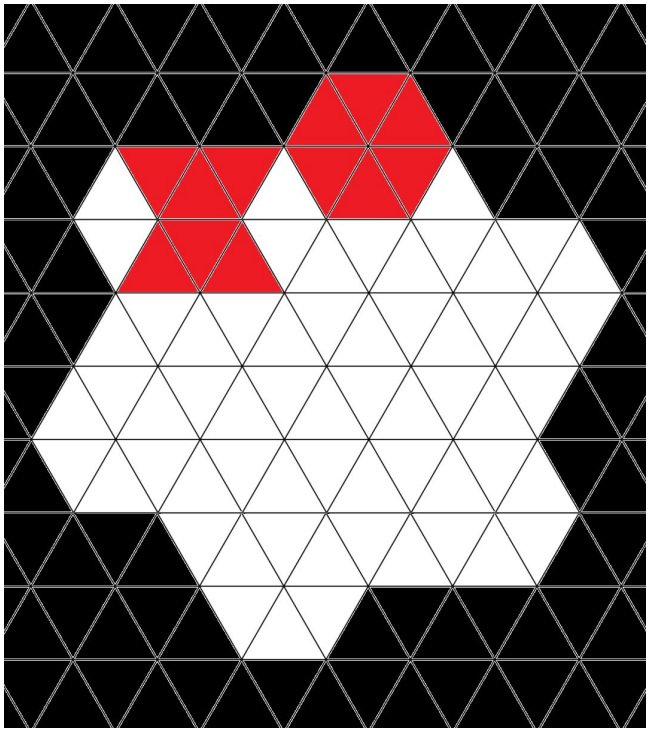


Fig. This board is legal, but has no future. Pieces will never fit the small hole on the West of the hourglass piece. It seems relatively labor intensive to check boards for having a future. We went a solution that does not need this step.

Search algorithm

In order to speed up things vastly, a check needs to be done after each piece is laid on the board to check if this board still has “a future” as a winning candidate.

Instead of using complicated algorithms to check if there are any holes that won’t be able to contain pieces, or margins too small for fitting left over pieces, a simple approach was used.

Set up all possible sequences of pieces. Now check for every sequence if its pieces would fit in the board, starting from the top left to the bottom right. Check every orientation for every piece.

By requiring that the cells of the board are filled systematically from the top left to the bottom right, flawed boards and gaps are very quickly eliminated as unworkable solutions.

Set up sequences

As we will try to fit the pieces in the board from the top left to the top right, we need all the possible sequences of pieces. That’s a permutation.
Imagine there was only one piece. That’s one possibly “sequence”

If there are two pieces (A and B), there are two sequences. AB and BA
If there are three pieces (A, B and C), there are six possible sequences:ABC, ACB, BAC, BCA, CAB and CBA. that's actually 3x2x1=6
If there are four pieces, that's 4x3x2x1 possibilities = 24
…
There are 11 pieces
11! = 11x10x9x...x3x2x1 = roughly 40 million sequences.

We will need to repeat the same pieces sequences for every base board.

Example of a sequence:
[3,6,1,7,2,8,9,11,4,10,5] every piece has a number from 0 to 11. Piece 0 is the hexagon, as it is integrated in the base boards, it is not included in the sequence.

Systematically check if a sequence of pieces is a winner

Take the next piece of the provided sequence
Take top left empty cell of board.
Check if putting down any orientation of the piece covering that empty cell produces a valid board.
If not, try another orientation of the previously laid down piece.
If fitting, repeat for the next piece in the sequence.

This is a textbook example for recursion.

An random example:

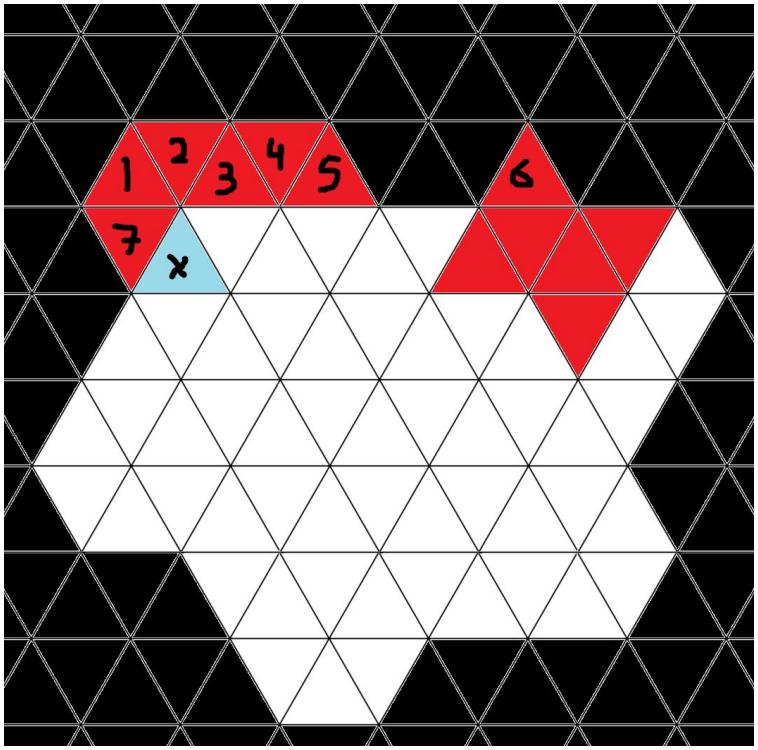


Fig. In the provided sequence, the hockey stick and the hourglass come resp. first and second. cell 1 was the first top left cell. Various orientations of the hockey stick were tried, until if fit. Then the next free cell was searched, 2.3.4.5 were tested but not free. 6 was found free. The hourglass piece was

found to fit in an orientation. next free cell was not 7 but x this is the current free cell for which the next piece in the provided sequence will be added.

Estimation of complexity

A sequence of pieces in a particular order is tried on the board.
We work with 11 pieces (Just remember: there are 12 pieces but 1, the hexagon, is used to make the baseboard).
The number of sequences is a permutation of all the pieces.
That's 11! sequences. That's about 40 million. (39,916,800)
Each piece has different orientations, we need to test them all. Empirically, we saw that about 100 pieces needed to be laid on the board to conclude that this sequence was not going to work.

So, the total search space is:

- 5 base boards
- 40 million sequences per base board
- 100 pieces laid per sequence

which gives us a total of 20 billion pieces to be laid down to search every possible combination.

That's a lot. But it's not impossible. Before any optimization on an i5 processor, single thread, I tested at about 1,500,000 pieces per hour.
which makes about 20 billion / 1.5 million = 13000 processor hours. Seems a lot. But the program was multi processor ready. 10 processors with 8 cores could do the job then in about 150~200 hours.
Also, the software could probably be vastly improved. Something that could be done for fun, but would be a waste of time otherwise.

Solution aftermath

Solved after randomly testing about 65000 sequences.
Which makes for a very rough first estimation of (200,000,000 / 65000=) 3000 solutions.

Fig. SPOILER ALERT! One of what must be many solutions.The creators went out of their way to not provide any hint. Using different kinds of wood to make sure solvers could take no clue.

The “agile” approach of first getting the program into a state where searches can be performed and to refrain from premature optimizations paid off here. In fact, no systematic approach was even executed. As yours truly had to help moving a friend, a welcome covid physical labor distraction, the systematic approach was not yet set up. Before leaving the house, a routine was quickly set up where random sequences are tested. A big hurray was uttered when, upon arrival in the evening, it appeared like a solution was found already.
As such, the part where all 5x 11! sequences are generated, together with a workpool to set multiple computers at work was disregarded. A nice follow up would be to try to run this on a machine in a cloud service.

Future work

As a solution was found, the project ended right there.
There are however interesting side projects possible.

- It would be interesting to check how many solutions there are.
- There is a lot of research on optimizing search algorithms. It would be interesting to see how this can be optimized further.