Solving a Patterned Polyomino Packing Puzzle

Algorithmic Study

Prateeksha Singh

November 22, 2019

Abstract

We present an efficient way to solve a color-constrained assorted multi-sized polyomino piece packing puzzle (named the Dr. Wood Kaleidoscope Classic): A 8 x 8 packing puzzle with numerous possible packings, but which enables wants a specific pattern. arranging a packing that matches a given pattern.
The python implementation of the algorithm can be found at GitHub: www.github.com/pratu16x7/kaleidoscope

Keywords: packing puzzle, edge matching, backtracking, polyominoes

Introduction

Packing Puzzles

Packing puzzles have intrigued folk through decades, both to recreational mathematicians and programming enthusiasts. The Pentomino packing puzzle is the most popular, which requires packing all the 12 pentomino shapes into different sized rectangles. They are a special kind of polyform; Pentominoes is an example of packing with polyominoes, the simplest polyform packing, involving packing together a group of polyominoes which may be same or different in size.

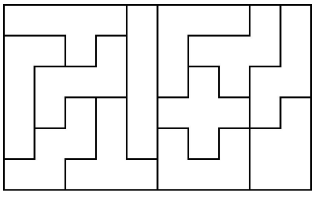


Fig 1.0: A solution to the Pentominoes puzzle

The Kaleidoscope Classic

The puzzle consists of an 8x8 board, over which 18 polyomino pieces are to be arranged: 1 octomino, 10 tetrominoes, 4 triominoes, 1 domino, 2 monominoes. Each piece is colored on both sides and can be chosen to be placed on either. No two pieces are alike. A pattern is a particular arrangement of the pieces, placed with either side chosen, on the 64-celled board, shown without the piece placement. A solution to the pattern is a board that shows a possible packing that the pieces can be arranged for the pattern.

As the pieces set is built to be easy to pack to fit the board in many ways, the challenge is to pack them in specific ways. There are over 200 patterns possible, each having as many as millions or as little as a single solution. It is Dr Wood's first widespread creation and a winner of the Puzzle of the Year by the Australian Games Association in 2004.

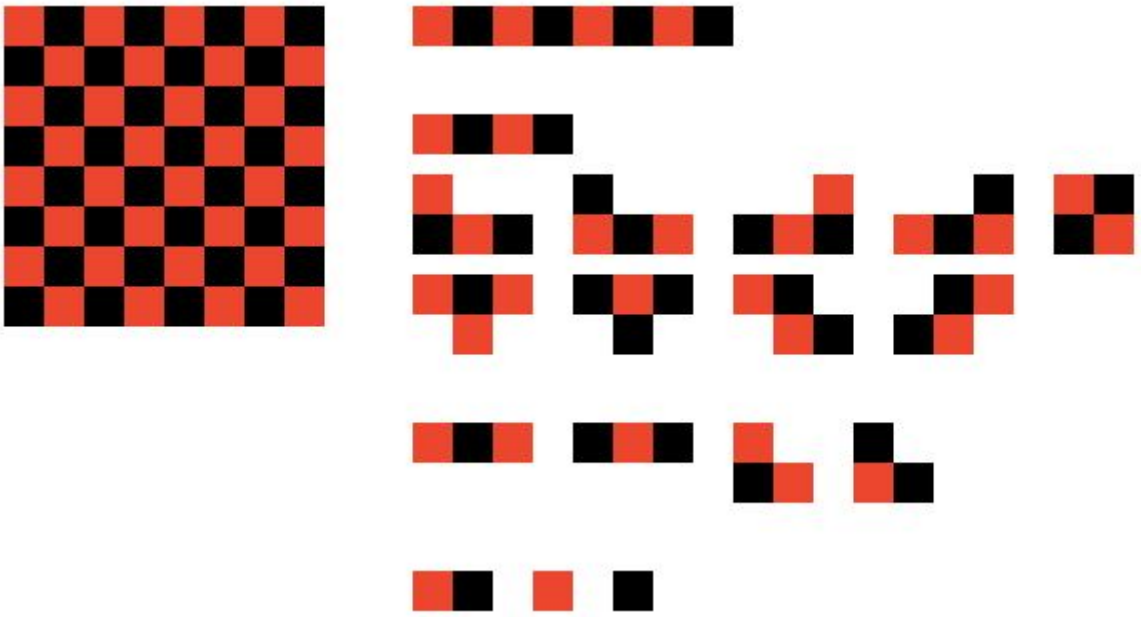


Fig 1.1: The Kaleidoscope Classic: The 8 x 8 board and 18 polyomino pieces. No two pieces are alike.

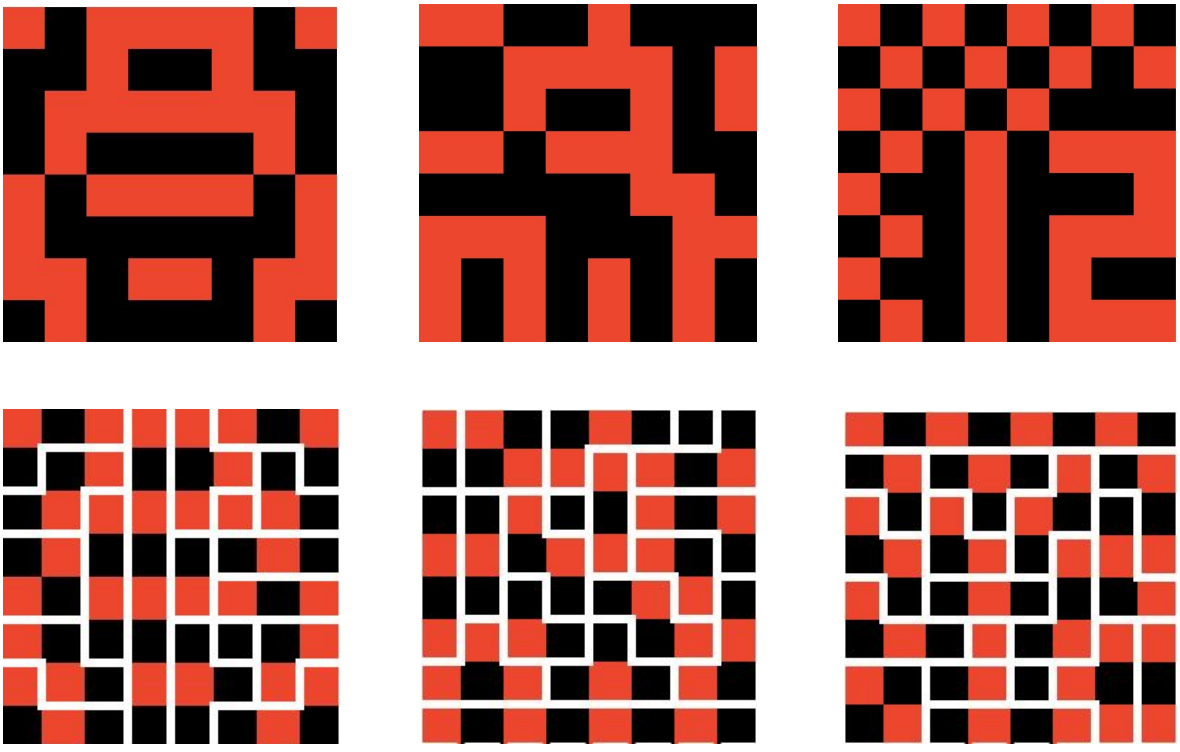


Fig 1.2: Some puzzle patterns and their possible solutions

Problem definition

"*For a given pattern of an 8x8 grid, and the given 18 pieces colored both sides, find a placement of each of the pieces such that the pattern is satisfied.*"

Terms used:

- Board
- Pattern (any grid)
- Cell edge info
- Piece
- Y, X coordinate system of a board
- A solution move: piece, orientation, coord

We propose an algorithm to account for the difference in size and colors of the pieces and take advantage of them to make a typical edge matching algorithm fast.

Blue and Yellow sides are beyond the scope for this analysis but are presented in the 'Future Improvements' section.

Observations & Motivation

The context: Perceived Human difficulty

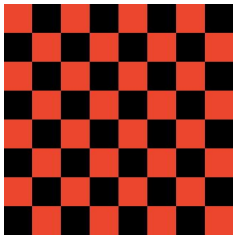
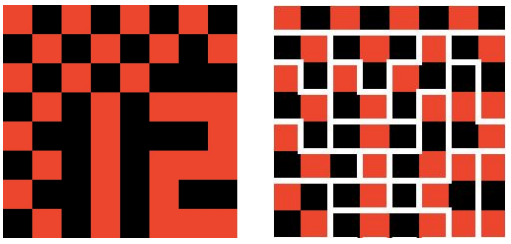


Fig: The checkerboard, the easiest pattern to solve, with billions of solutions

Conventionally, the checkerboard pattern, with alternating colors is considered the easiest to solve. Partly because the board pieces have Conversely, any board that is not the checkerboard is considered harder: the more features a board has, the harder it is to solve. However, we present some observations that show that this is not the case.

The solution as an increase in cell edges

Looking closely at a solution of a given pattern, we observe that the placements of all the 18 pieces give us additional information apart from the cell colors: edges. In the same vein, a problem pattern is represented with zero edges, and with every piece placement, we add to the edge information and move closer to the solution (maximum edges).



Zero edges 65 edges

Fig: Increase in edges from pattern to solution (excluding the 32 trivial border edges)

Trivial edges

In any pattern, the borders of the board are implicitly edges, which serve as a starting point to place the pieces.

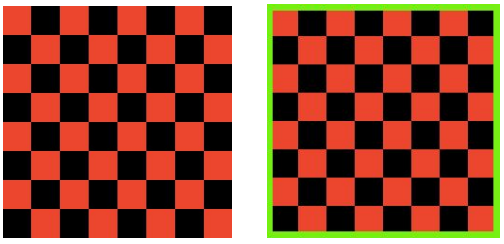


Fig: . The 32 implicit edges in any pattern

Adding to Trivial edges (color adjacency edges)

We observe that all pieces have alternating colors, and hence continuous color regions (same-colored adjacent cells) indicate the presence of a divide, across which two pieces have been placed.

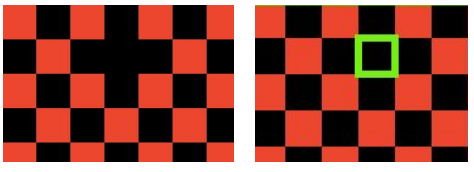


Fig: (small window) implied edges from same colored adjacent cells

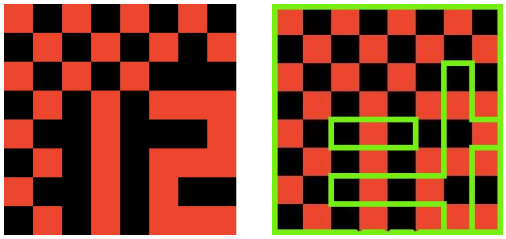
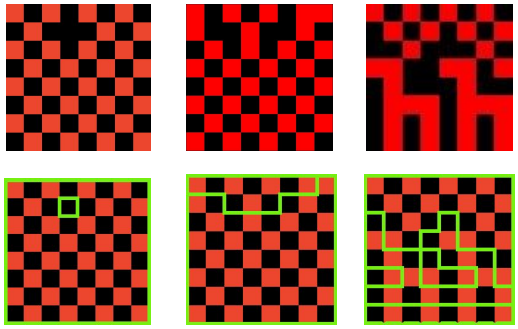


Fig: 1. (small window) implied edges from same colored adjacent cells

Checkerboard as a base pattern

And any attempt to tweak the checkerboard pattern introduces *some* secondary adjacent color edges.



[fig]Row 1: Incr in features,Row 2: Increase in edges

Fig 2.1: Edge info in any board different from checkerboard pattern

And now combining all of the above, the following follows,

1. a solution is progression to more edge info
2. The checkerboard is easiest to solve (billions of solutions)
3. In any pattern that is not a checkerboard, there is bound to be some non-trivial edge info

=> it should be relatively easier to solve with built in info. Or at the very least, there should be a way to do it.

The least featureful checkerboard, easiest to solve. Any intent to make a different pattern will involve putting in same colored cells adjacent. This in turn adds additional edge information to the trival 32 board edges. Since checker board gives least edge info and is the easiest to solve, it is my contention that with more edge information at hand, it can only get easier to find the solution.

A closer look at why Checkerboard is easiest

Let's see why the checkerboard is easy to solve, and others are perceived 'harder':

- because fo its flexibility: there are greater number of posible next moves at every step due to it's genrality => hence many different permutations of possible pieces-position , i.e. solutions

Having some edge already in place deprives us of choice, but the constraints is what gives us less possible options, thereby helping us select best piece with more confidence and move towards a solution quickly.

Having established this, what we now need is a way to actually get possible options and a way to find out what defines a good position among others.

Proposed Algorithm

Set of moves tried, and until board has no remaining holes and no remaining pieces, or no moves

1. Define territory (Separate holes) by adjacent same color cell edges.
2. Check if any holes have single piece solution, and place them
3. See if the magic wand any of the holes (practically done in the first step)
4. While pieces remaining:
   a. Make a size progression for each hole by cell count, wrt available pieces.
   b. Look for the densest windows across holes. Rank by edge/cell density.
   c. Fit pieces to each hole, rank each window-hole combination by:
      i. edges matched and crookedness heuristic of pieces.

ii. span (only small_wand, and theoretically magic wand)
iii. Select the winner, but keep a set of other next best moves

d. If no moves:
i. Try a different sized piece, according to a different progression
ii. If still no moves, backtrack to parent and try it's sibling

Before going into each step in detail, let's introduce a few techniques.

Techniques for Implementation

Grid representation

Patterns, solutions, and pieces can be represented as matrices, with each cell carrying three pieces of information: color, coordinates and edge information (calculated later). This makes it easier to perform comparisons between each piece with a particular region of the board to place them, as well as rotating the piece grids to check if they fit.

Edge scoring heuristic-based search

As multiple piece-placements (moves) may be valid at any given point of the puzzle, we can define a heuristic specifying the denser edge regions in a pattern to be filled first, and another to score pieces based on their 'crookedness'. Tetronominoes form a majority, are chosen first wherever possible, leaving the smallest pieces for the end.

Backtracking and Tree Pruning

As it is customary to solve packing puzzles with a backtracking algorithm, a puzzle such as the Kaleidoscope having different sized pieces has many opportunities for tree pruning at earlier stages to reach the solution quickly.

Detailed Stages

Selecting the piece set

Usually, red and black combination boards have an even balance of 32 red and 32 black cells, which implies that the red monomino is to be placed as is. However, it can also be flipped to its black side to create different patterns, which have a 31 red + 33 black count:

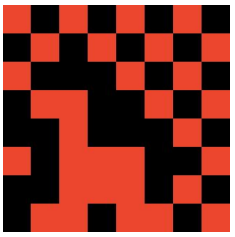


Fig 4.1: The Seal

Red Count = 31, Black Count = 33

Fig 4.2: The Red Monomino and its flip black side

While solving such a pattern, it is necessary to confirm this at the very beginning, to define the piece set with the suitable side of each piece in general. This is even more relevant in patterns with Blue and Yellow sides of the pieces, mentioned in the 'Future Improvements' section.

Finding edges as Islands (holes) territory

As we know that finding edges in a pattern brings us closer to the solution. We can look for same-colored adjacent cells in order to find the initial non-border edges. There are two ways to do this:

1. ***Fill algorithm***: Maintain an untraveled and traveled cells list. Initially, all the cells are untravelled. Randomly select a cell from untravelled and add all its connecting cells (up, down, left, right) to the traveled list. If any of them have the same color as the cell, an edge exists in that direction. Repeat by selecting another cell.

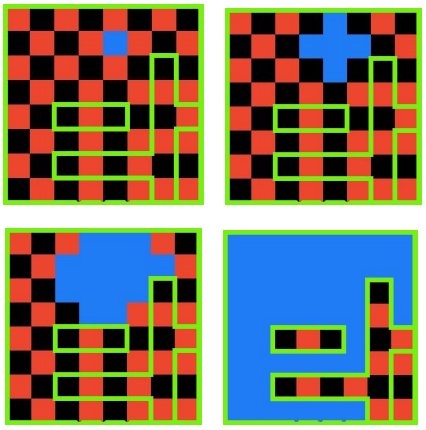


Fig: Fill algorithm to find regions

2. ***Finding islands*** (efficient): Maintain a list of islands. Scan the pattern row-by-row one by one, and compare each cell with its joining cells in the same row, as well as the row before it to mark the edges. If they don't share their color, they are on the same island. At the end of the parsing check the number of island regions formed.

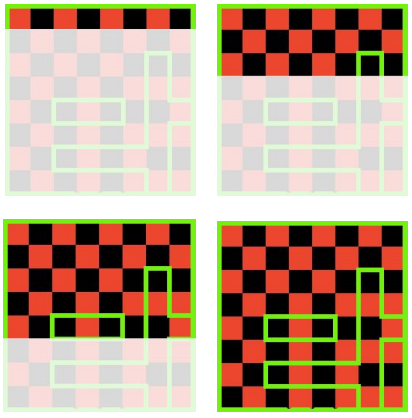


Fig: The efficient island approach
Successive island count: 1, 2, 3, 4

The next step is to explode dismantle the regions so that we can focus and prioritize each closed-off region separately.

Implementation detail: Each hole preserves its relative offset from the whole board, it's most top-left coordinate so that we can trace every piece offset that is first calculated wrt to the region can be used to calculate its absolute offset from the board.

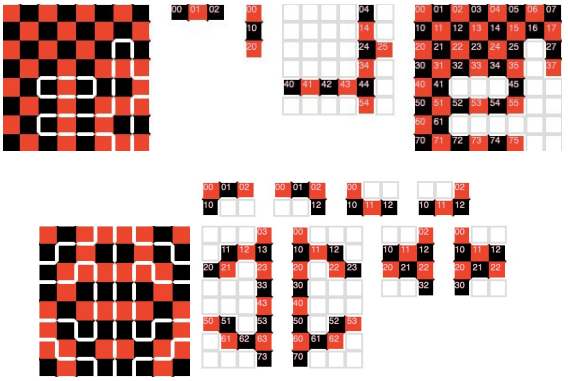


Fig: Puzzles is its simplified form, regions each revealing a new set of edges.

Targeting constrained areas first

Most constrained places, the ones with the most edges, should be solved first. The smallest regions are the most constrained, usually having a trivial solution as a single piece, followed be areas in the bigger holes having areas of high [edge density], that can only possible house few pieces (The algorithm to do this and edge density are covered in the next section).

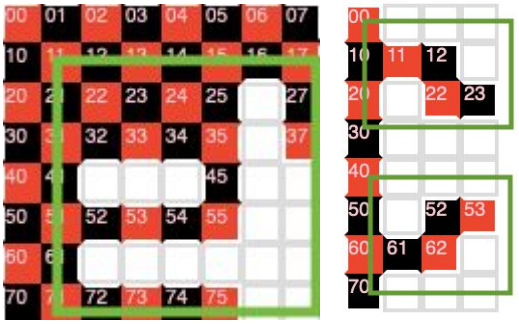


Fig: Constrained areas with most edges per area.

Triviality

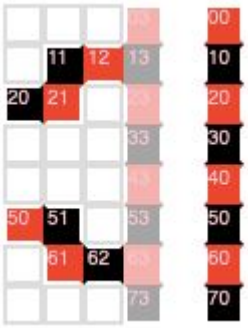
In some cases, there are regions (holes) formed that can be solved with a single piece. We'll call these trivial solutions. These moves are non-negotiable. As such they are placed at this stage and do not form branches in the backtracking tree.

- trivial pieces (no branch)

Trivial solutions or without,

- octomino (best chance, but first branch)

This is also the stage to place the magic wand, as only a few islands can have it. This is necessary, as the octomino is the piece that has the most freedom, and needs the most leeway. As subsequent stages will decrease both, The start stage is the only one that satisfies this requirement.



In general, larger pieces are preferred first. As this is our first move, if there are multiple possible positions, This would mark the first fork in our backtracking tree.

Hole count progression

Hence, calc size progressions for all holes, not accurate but still: we try to look at the cell count of the hole and make an educated guess as to the sizes of the pieces that will fit the hole. Acts as the primary constraint, which is a reasonable guess at the start and gets accurate towards the end constraint.

For example, if the cell count is 17:

17 = 4 + 4 + 4 + (5)

However, in order to do this efficiently for all given holes, we need to how many pieces of a given size are available, in order to prioritize the **biggest first**. Let's look at the count distributions for the remaining pieces:

Considering magic wand out of the picture, we have

Tetrominoes (size: 4): 10

Triominoes (size: 3): 4

Domino: 1

Moniminos (size 2): 2

This sets some Goals: in general,

- 4 first,
- available pieces,
- try bigger before smaller
    - corollary: 2 and 1 sizes are easy to fit but very rare, so reserve use for last

We start with tetrominoes until 8/6/5/3 cells left (7 is just 4 + 3, where 3 = 3 or 3 = 2 + 1, and 4 is), after which, there are multiple progressions possible,

5 = 4 + 1, 5 = 3 + 2

6 = 3 + 3, 6 = 4 + 2

8 = 4 + 4, 6 = 3 + 3 + 2

and we keep a provision to try each of them separately if one doesn't work out. This has to be continuously tallied based on the hole state (as the cell count changes), and available pieces, if every sized piece of the progression is available. These form the necessary conditions to decide the next expected piece count while choosing a move.

Get next move

The repeating step after every move.

Consists of 2 or 3 best choices:

- Choosing the next hole, congested
- Choosing the next window

This depends on:

- overall density, densest window of all, but if a hole is smaller complete it first
- that means windows across all holes by default, but if some hole small then windows only to it
- some examples
- keep updating progression once done

The Sliding Window

To efficiently find potential candidate pieces that can fit in a hole, we take the help of windowing. Windowing involves moving over all the sides of a

hole and collecting mini piece-sized grids. Such a window has characteristics like cell count, edge count, and open edge count. A row has hole and window pos, actual win, stats.

Window sizes

Just like the magic wand, if the small_wand is in available pieces needs possible positions. Special window 4x1. For all other pieces, the window sizes is 3x2 can fit all of them.

[fig]: 3x2 window, 4x1 windows, and their rotated versions as well

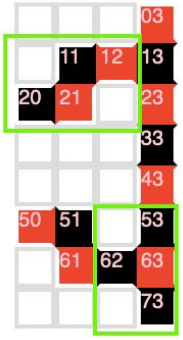


Fig 4.3: 2x3 and 3x2 windows

Window selection thresholds

As our goal is to target the most edges, we try to get The small wand has to be placed in a long position, that doesn't cause it to jut out. In order to do that, it should generally be placed flush against a border, also having an end covered. Hence, 4x1 windows have to satisfy a special constraint, touched on at least one end, and at least one whole side i.e 5 edges

Implementation detail

Instead of capturing every window initially and calculating its edge count, we can quickly find the coord for the most edged windows.

3x2 Windows

We first make a grid mapping of all counts, then map it over for counts for our window size. Grid, grid with edge counts, and a grid with horizontal and vertical gradients. Then we can select the ones with the highest score.

4x1 Windows

Parse every row in the pattern to see if the row has at least 4 cells. If it does, select the starting point, if more that 4 select each of the start and end.

Selecting from the thresholded pieces by distribution

We select the highest edges count window to test out which pieces fit them the best and measure the match of each.

Turning windows to Moves

Each possible window is then evaluated or . In 3x2 windows, whether all if its cells can be filled by a piece, there can more than one piece . Thus we generate window piece pairs. And give each a score, based on

[fig]: 2-3 examples deciding a piece based on a 3x2 hole

This is also the part where the next expected piece count is considered. A piece is only selected if it matches the next expected count.

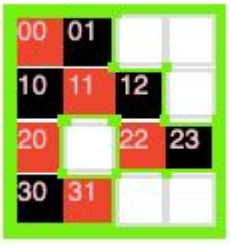
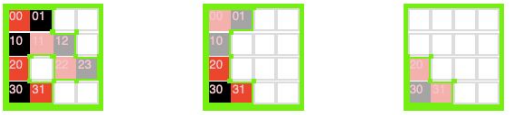


Fig: A hole example with 10 cells, demonstrating 4 + (3 + 3) rather than 4 + (4 + 2)

In this example, the following moves are deduced.



Scoring moves: Edge Matches and other factors

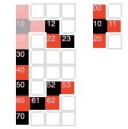
Based on the cells covered and number of edges the window has, and the cells, the bestness of a move (a piece fill) is defined by:

- edges overlap between the piece and window
- Rareness/ Crookedness / Trouble-to-fit of the piece

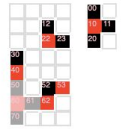
Edge Overlap

Matched edges count + no_of_total window edges

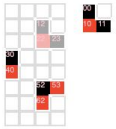
Matched edge count is the number of those piece edges (greater set) that are also window edges.



8 | 8 | 10 + 0.52 + = 12.52



8 | 8 | 10 + 0.52 + = 12.52



8 | 8 | 8 + 0.26 + = 12.26

Bonus deviation heuristic / reservation

Pieces like the Z and T have more trouble to fit and hence are assigned an additional score for their crookedness, for them to be picked up faster and filled in as soon as possible. In case of examples of tied scores, or cases where there is only a slight difference, the crooked pieces will be considered due to their bonus, just enough score to break tie or one less edge.

Hence, the final score is given by,

Score = Matched edges + crooked piece bonus

Other factors:

Ideally, the move should not break the hole into two parts. One exception to this would be that the resulting two holes have a <3 sized hole that is part of the size progression of the hole and present in its available pieces.

Making a move

After considering the scores, the top 3 choices are then chosen in order to be the forks for this stage of the game, and the first piece is chosen for the current move. The other pieces will be used to back track later. Current hole state, selected piece forks, resultant state from first piece are calculated.

Implementation detail: Placing a piece

In order to make a move, we have to merge the selected piece with the hole at its selected coord. Once the piece is placed, the new hole is created by:

- Nulling the piece cells
- Establishing edges at the piece open edges

No moves available

There are a couple of steps to follow:

1. Change piece size progression:
2. If no progressions left, Backtrack to previous moves and select the next best sibling.
3. If no siblings left, back track to parent and check its siblings
4. Repeat step 3

Once we have made a move, the get next move cycle is repeated.

Backtracking

While making a move, we also save the next best pieces at every stage for later. These form the nodes in a backtracking tree. At any given stage where there are no moves even after trying a different count progression, the last piece is backtracked to first try its sibling and proceed to its parent and continue the search if it fails.

Practical Walkthroughs

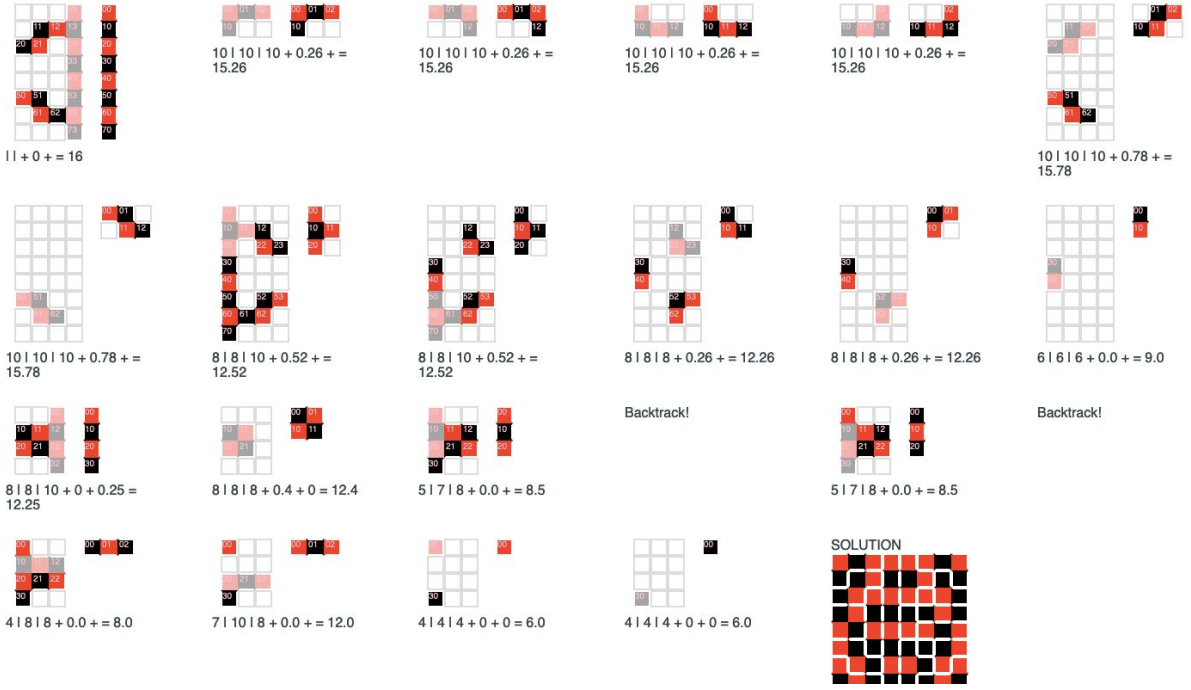


Fig 4.4: The Car Pattern solution: Move progression with edge scores and backtracking

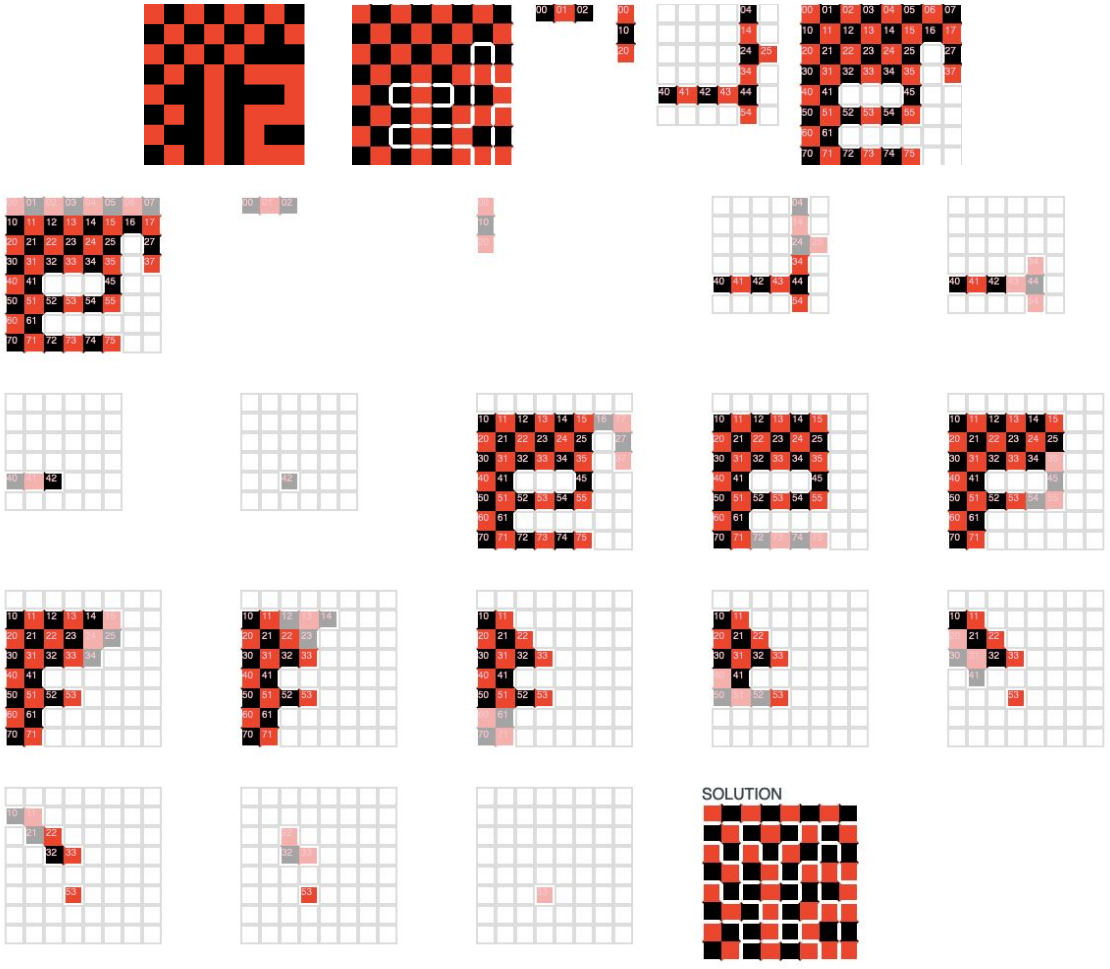


Fig 4.4: The '12' Pattern solution: move progression

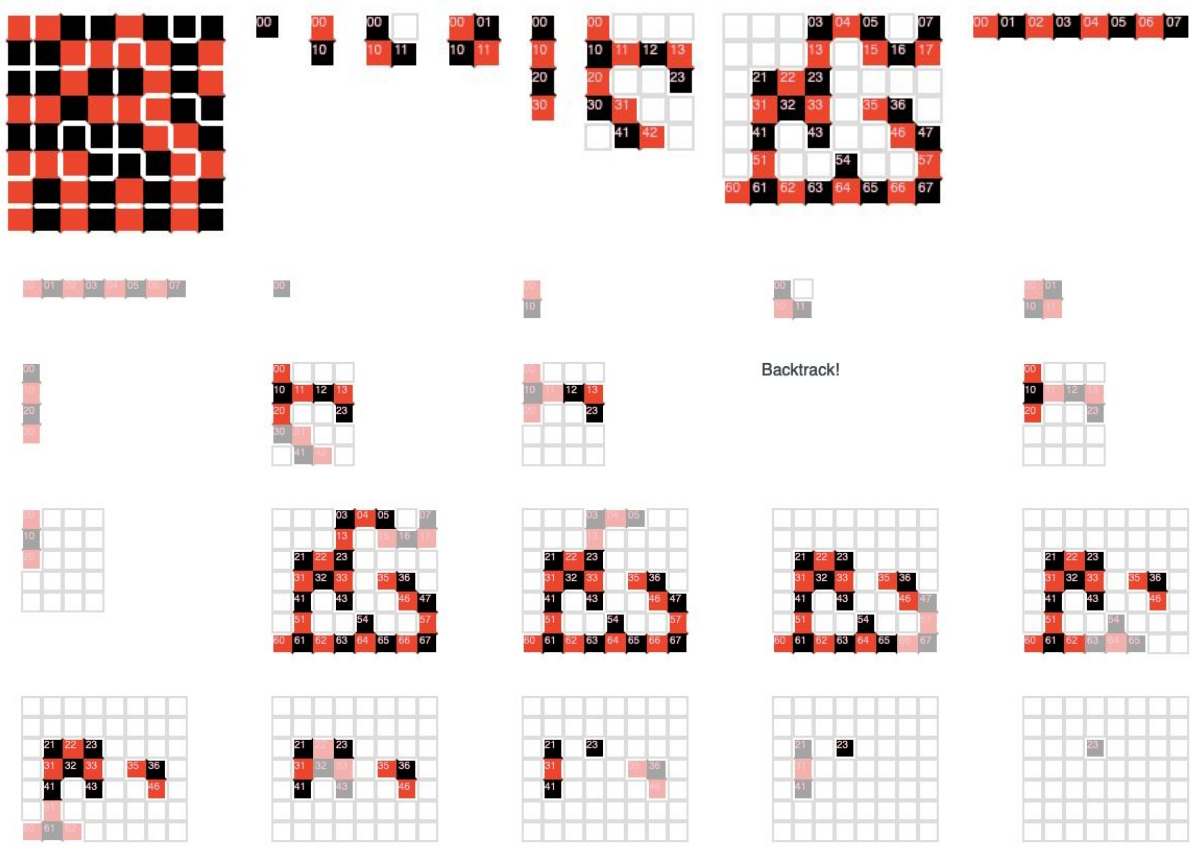


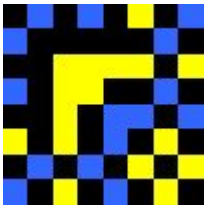
Fig: The No Single Squares solution: Move progression with edge scores and backtracking

Implementation, Performance Tests

The previous page shows two different solution progressions for the Car and 12 patterns. The Car pattern has a higher degree of initial fragmentation into holes, however it still requires some backtracking in the final stage to fit in the triomino. The 12 board on the other hand, despite having less number of holes, is able to find a fitting piece at every step without having to backtrack at any point.

Future Improvements

We have considered piece possible position based judgement only in the case of small wand. However, this will become highly important in the case of flipped blue and yellow pieces, because these piece side are less homogenous due to two colors taking turns, and there are few possible places for each piece to go.



Combining both types of branching, we can come up with a robust approach to solve black and red, blue and yellow, and even hybrid Kaleidoscope boards.

Conclusion

The algorithm makes use of finding islands, edge matching, piece size and crookedness heuristics and backtracking to find the best fitting pieces. We have seen that performing every possible technique

to prune the backtracking tree leads to a faster solution, and achieves performance quite better traditional edge matching, taking advantage of the unique nature of the Kaleidoscope puzzle.

References

- Edge-matching: https://erikdemaine.org/papers/Jigsaw_GC/paper.pdf
- Lew Baxter. Polyomino puzzles are NP-complete. Message on Eternity Yahoo! Group. http://groups.yahoo.com/group/eternity/message/3988
- Knuth Dancing Links: https://www-cs-faculty.stanford.edu/~knuth/preprints.html (P159)
- 3Backtracking: (Something) James R. Bitner and Edward M. Reingold. Backtrack programming techniques.Communications of the ACM, 18(11):651–656, 1975.
- Superforms: http://puzzlezapper.com/aom/mathrec/polycover.html
- Polyform: https://en.wikipedia.org/wiki/Polyform