CMPT 417

Tetromino Packing

Jacob Patenaude - 301203788

Problem

Given a 4x4 grid and a number of rectangle, square, L-shaped, and T-shaped tetrominoes, determine if there exists a way to arrange the tetrominoes such that the board is filled without any overlapping pieces. E.g. there is a packing when there are four rectangles:

1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4

Chosen Language

I chose to write the specifications for the problem in the IDP System by DTAI Research Group as the logical statements involved were very close to the theoretical second-order logical formulae we have been using in class. For writing the IDP specification, I made extensive use of the online IDE.

Motivation/Goals

I was interested in the tetromino packing problem as it seemed to have an obvious, but not necessarily trivial, extension –making the board size bigger. My goal was to, after solving the given problem, create a generic way of generating a satisfying model for a given number of each tetromino and to have a way to model the performance of finding that solution so that I could see how the performance changed for arbitrary inputs and even, hopefully, for arbitrary board sizes. Finally, due to the visual nature of solutions to the problem, I wanted to create a visual output of solutions to quickly and cleanly display and check solutions.

Initial Solution (Vocabulary and Structure)

To start the vocabulary, I gave each of the four blocks that would be used in the packing an id from 1 to 4. Then, to describe the placement and type of tetrominoes within the board, I used five associations:

I. The type of a piece out of rectangle, square, T-piece, or L-piece (referred to as constants “R”, “S”, “T”, “L”).
II. The rotation of a piece out of 0, 90, 180, or 270 degrees clockwise (referred to as constants “ROT_0”, “ROT_90”, “ROT_180”, “ROT_270”).
III. The reflection of a piece (simply a boolean of whether or not a piece is reflected).
IV. The location of a piece’s anchor position –I gave each of the tetrominoes a special cell to anchor a piece onto the board, given its rotation and reflection. E.g. for a rectangle, I anchored the piece by its bottommost cell (when viewing a rectangle vertically).
V. The block that occupies the cell at a given position in the board. The actual solution to the problem is this list of block positions as it directly describes the packing which corresponds to the solution to the given inputs (if the inputs give a satisfiable structure).

These were implemented as relations:

```
BlockType(Block, Type)
Rotated(Block, Rotation)
Reflected(Block)
Located(Block, Index, Index)
Has(Index,  Index, Block)
```

Where “Block” corresponds to a block’s idea, “Type” to one of the type constants, and “Rotation” to one of the rotation constants.

From here it was a simple matter of defining the input as the number of each tetromino: nR, the number of rectangles; nS, the number of squares; nT, the number of T-pieces; and nL, the number of L-pieces as the instance structure.

Lastly, I needed to write the theory which would allow IDP to derive the packing of the board from everything given above.

Initial Solution (Theory)

The simplest part of the theory was giving some constraints on the inputs. I knew that I needed to ensure that the given numbers of each tetromino corresponded to the actual number of occurrences of that block within the solution and that I needed to restrict the input relations to actually contain valid entries. That gave me the following restrictions:

```
// All blocks have a type and the number of each type is correct
!b[Block] : ?1 t[Type] : BlockType(b, t).
#{ b[Block] : BlockType(b, "R") } = nR.
#{ b[Block] : BlockType(b, "S") } = nS.
#{ b[Block] : BlockType(b, "T") } = nT.
#{ b[Block] : BlockType(b, "L") } = nL.

// All cells have a block
!x[Index], y[Index] : ?1 b[Block] : Has(x, y, b).

// All blocks have a unique location, reflection, and rotation
!b[Block] : ?1 x[XIndex], y[YIndex] : Located(b, x, y).
!b[Block] : ?1 r[Rotation] : Rotated(b, r).
```

From here, the theory became much more tedious and challenging. For each of the possible block types and rotations and for both reflected and not reflected blocks, I needed to define which cells would be occupied by a block relative to its anchor position in general terms. This gave me over a dozen statements that roughly all had the form:

```
!b[Block], x[Index], y[Index] : // For all blocks and cells
        BlockType(b, "R") & // If the block is a rectangle
        Located(b, x, y) & // And the block is anchored to the cell
        Rotated(b, "ROT_0")  // And the block is not rotated
        // No mention of reflection as a rectangle reflected is the same!
=> // Then
        (y + 3 =< 3) & // The top of the rectangle is not outside the board
        Has(x, y + 0, b) & // And the rectangle occupies all the cells above it
        Has(x, y + 1, b) &
        Has(x, y + 2, b) &
        Has(x, y + 3, b).
```

And at last a solution was had! As an aside, I was initially getting incorrect solutions where the locations of blocks made no sense. I realised it was due to the provided notes having the various indexing constraints (e.g. (y + 3 =< 3) above) on the left side of the implication, but the solver was simply making the implication false and putting the pieces wherever it wanted (within the other given constraints). Though I am curious if with a different set of constraints having the indexing occur before the implications could still give correct results, for me making this change was enough for IDP to spit out valid solutions for the given inputs.

At this stage, calling IDP for a single set of inputs took about 0.28 seconds.

Next Steps

Now that I could determine if a set of inputs corresponded to a satisfiable model and print out what that model was:

```
procedure main(){
    printmodels(modelexpand(TetrominoPacking, Packing))
} // TetrominoPacking is the theory and Packing the structure
```

I wanted to automate running the task so that I could check for packings for all possible inputs nR, nS, nT, nL for a 4 x 4 board. This amounted to finding all integer compositions of 4 of length 4 (E.g. (4, 0, 0, 0) for 4 rectangles or (1, 1, 1, 1) for one of each block type). Admittedly, I copied optimised code for doing this from [a gist](). Now that I could iterate over all inputs, I needed a way to programmatically call IDP. To do this I added the IDP binary to my system's path (to allow calling it as idp <filename> from any directory) and used Python's subprocess.call utility to invoke that call (adding the "nowarnings" flag to keep my terminal clean from the warning that IDP was auto-completing the given structure as that was exactly the goal!). Finally, I needed a way to set the inputs repeatedly in the IDP file itself (main.idp in my case). To do this, I created a template file with all of the vocabulary and theory and then repeatedly copied the files contents and added the current iteration's inputs and output the results to my IDP file and that was it!

It turns out that there are 35 compositions of the integer 4 of length 4 (when 0s are included) so at about 0.3 seconds per IDP call and with some processing time, running through all of the inputs and determining the satisfying model (if any) took about 10 seconds.

Now that I had all of the valid inputs for a 4 x 4 board, I was interested in displaying them so that I could see the packing which made each board satisfiable and verify its correctness. To accomplish this, I scraped the stdout output of the printmodels call in my IDP main procedure by changing my use of subprocess.call to subprocess.check_output and then parsed that output to generate a JSON (JavaScript Object Notation) string containing all of the details about each iteration. This allowed me to generate a very basic HTML file with the output (as JSON) hard-coded in under a <script> HTML tag. The reason I elected to do this, was that I could now generate the output using JavaScript which, in addition to being very familiar to me, meant that I didn't need to re-run my IDP script to view changes to, for example, my CSS styling.

Final Generalisations

Now that I could view all of the packings for a 4 x 4 grid, I had two final goals for generalising the problem:

1. Generate packings for all valid board sizes ranging from 2 x 2 up to some specifiable maximal size (including non-square sizes such as 2 x 4)
2. Add the final Z-piece tetromino (for all its reflections and rotations)

The first problem was overall quite straightforward, simply change the type Index to be two separate types XIndex and YIndex, and also allow specifying what the maximum values for those indices were (simply the number of rows/columns decremented by 1 for 0-indexing) and what the maximum block id would be. In general, for an N x M board, there are N x M / 4 tetrominoes since each tetromino occupies 4 units of area and there are N x M available area units in such a board. This meant I was now generating all integer compositions for the number of blocks of length 4, rather than simply the 4 blocks I was using before.

The second problem was slightly more challenging as, in addition to supporting the new nZ input for the number of Z-pieces, I needed to add the theory for the cell-occupation of such a piece given its anchor location! This (like the L-piece had been) was extra annoying as a Z-piece, unlike a rectangle, is different for all 4 rotations and both reflections (reflected and not). Additionally, I now needed to generate integer compositions of length 5 to accommodate the new block type.

Now that I had ways of generalising the problem, I utilised Python's getopt utility to take in some command line arguments to allow for modifying, for example, the maximal board size or whether the Z-piece should be included. The final options available from the command line are copied and formatted below:

Usage:	
-n <numRows> or --numrows <numRows>	Specifies the height of the board (Overridden by --boardsize)
-m <numColumns> or --numcolumns <numColumns>	Specifies the width of the board (Overridden by --boardsize)
-b <maxSize> or --boardsize <maxSize>	Specifies the maximum board size. Will run through all possible heights and widths up to <maxSize> x <maxSize> (Default 4)
-r or --reflections	Include reflections (Default: False)
-z or --ztype	Include Z-pieces (Default: False)
-h or --help	Shows this help message

However, I noticed a new problem. While adding the separate x and y indices did not have a significant effect on performance, the additional constraints for the Z-piece did have a noticeable effect, even when the Z-pieces were not included. At the 4 x 4 board size level, the average time spent making an IDP call increased from about 0.28 seconds to about 0.36 seconds. I realised that even though the Z-piece constraints did not need to be fully evaluated when Z-pieces were not included (as they obviously only apply to Z-pieces), simply enumerating them all was resulting in an unnecessarily long grounding that was impacting performance. By modifying my templating to comment out the Z-piece constraints when they were not needed, the previous performance was restored.

Performance Improvements

I am confident that there exists a modification of my every block and cell theories that would improve performance; however, I was unable to find it. One thought I had was to assert some facts about rectangles, squares, and T-pieces to take advantage of their symmetries (e.g. asserting that all squares should not be rotated or reflected); however, the effect was actually slightly unfavourable. Additionally, I tried combining the for every block and cell theories of each block type into one larger theory; however, this also had a slightly unfavourable effect. It is unclear whether any combination of these changes would have a beneficial effect at larger board sizes.

Conclusions

Todo after running more iterations!