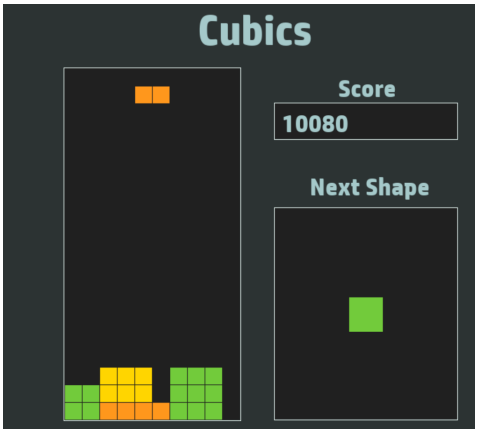
Project for the LAAI course – Tetris

Matteo Rossi Reich - matteo.rossireich@studio.unibo.it
Lorenzo Tribuiani - lorenzo.tribuiani@studio.unibo.it

April 2023



Abstract

The main aim of this project is to develop an automatic solver for a simplified version of the game *Tetris* using *Constraint Programming*. Since *Tetris* is a complex game to handle only with CP a simplified version is proposed where all the block are resticted to be cube or rectangle with a minimum shape of 1x1 and a maximum one of 3x3

1 Introduction

This section of the report is dedicated to an introduction over the *Tetris* problem, particularly in the formulation proposed, its modelling and formalization. In order to formulate the series of mathematical constraints needed for the resolution of the problem a formalization of the problem into a more structured and solid mathematical model is needed. In this particular case two different representation of the problem are used: a *Pivot* representation and a *Matrix* representation.

Shared parameters The following parameters are used by both the representations:

- $field^w$: The width of the level board
- $field^h$: The height of the level board

1.1 Pivot representation

In the pivot representation each block is represented by the x and y coordinates of the bottom-left corner and the *width* and *height* of the block itself.

Inputs

- n: The number of blocks already placed on the level board.
- $blocks^x$, $blocks^y$: Which represent the list of all the *bottom-left* x coordinates and the *bottom-left* y coordinates of the already placed blocks. Since this variable is defined as an array, $block_i^x$ is referred as the bottom-left x coordinate of the $i-th$ block.
- $blocks^w$, $blocks^h$: Which represent the list of all the widths and heights of the already placed blocks. Again, since this variable is defined as an array, $blocks_i^w$ is referred as width of the $i-th$ block.
- $priorities$: which is a vector of weights

Variables

- h: Which represents the height of the plate of all the blocks including the ones to be placed.
- $widths$: The list of the widths of the two blocks which have yet to be placed. Since this element is defined as an array, $widths_i$ is referred as the width of the $i-th$ block.
- $heights$: The list of heights of the two blocks which have yet to be placed.. As well as the $widths$ notation, $heights_i$ is referred as the height of the $i-th$ block.
- X, Y: Which represent the list of all the *bottom-left* x coordinates and the *bottom-left* y coordinates of the blocks to be placed. Again, since this variable is defined as an array, X_i is referred as the bottom-left x coordinate of the $i-th$ block.
- $rotations$: the binary vector which represents whether the new block have to be rotated or not.
- $widths^a$, $heights^a$: is a vector indicating the actual width and height of the block in case it is rotated.

1.2 Matrix representation

The matrix representation uses a $field^w$x$field^h$ binary matrix to represent the level board and all the blocks placed. All the indexes occupied by a block are set as one, zero otherwise.

Variables

- $field$: A binary matrix which represents the field with the blocks. It has value one where there is a block, zero otherwise.

This representation makes easier to compute the value of the following variables:

- $full\_rows$: the number of consecutive filled rows to be eliminated.
- $longest\_row$: the length of the longest non-filled row.
- $unreachables$: the number of holes which cannot be filled because they are blocked. i.e. there is a block on top of the hole.
- pos_y_full: is a vector deifying for each X the Y at which a block can be placed. (depends on the field state)

All those elements are sufficient to formalize the problem, in particular what is wanted is to define a model to produce the positions X and Y of all the incoming blocks avoiding h to grow bigger than $field^h$.

1.3 Variables domains

Since h, X, Y, $full\_rows$, $longest\_row$ and $unreachables$ are the variables of the problem, a definition of the **domain** of each of them is needed. Before starting a general assumption is needed:

Given the formulation of the problem is assumed that all the variables and values are natural numbers.
Here we list the domain of all the variables;

- h: [1,...,$field^h$]
- pos_x: [1,...,$field^w$]
- pos_h: [1,...,$field^h$]
- $full\_rows$: [0,...,$field^h$]
- $longest\_row$: [1,...,$field^w - 1$]
- $unreachables$: [0,...,$field^w * field^h$]
- pos_y_full: [1,...,$field^w$]

1.4 Pygames

The game itself was borrowed and adapted from the Github project of Arafat , it is created using the Python libray Pygames. To interact with MiniZinc which selects the moves to make, the MiniZinc Python package was used.

2 CP

Constraint Programming (CP) is a paradigm for solving *combinatorial problems* based on the declaration of *constraints* on the feasible solutions for a set of decision variables. This paradigm is implemented by various **constraint programming languages**, in particular, in the development of this particular project, **MiniZinc** constraint modelling language is used. MiniZinc is a *free and open source* constraint modelling language that allows an *high level* and *solver-indipendent* modelling for combinatorial problems.

2.1 Constraints

The following section is dedicated to the mathematical formalization of the fundamental constraints needed for the resolution of the Tetris problem.

2.1.1 Board Boundary Constraints

The main purpose of those constraints is to impose the new blocks to be placed inside the board without overlapping the boundary given. Formally:

$$X_i + widths_i \leq field^w \quad \forall\, i \in [1, 2] \tag{1}$$

$$Y_i + heights_i \leq field^h \quad \forall\, i \in [1, 2] \tag{2}$$

Basically, those constraints impose the *right-bottom* corner of each block to be inside the board. The *left-bottom* corner of each block needs no constraint since, by definition, is already constrained to be in [0,...,$field\_w - 1$].
One more improvement could be done to this constraint considering that only the circuits placed on the boundary needs to be checked, Formally:

$$R_x = |X_1 + widths_1, X_2 + widths_2, ..., X_n + widths_n|$$

$$R_y = |Y_1 + heights_1, Y_2 + heights_2, ..., Y_n + heights_n|$$

$$max(R_x) \leq field^w \tag{3}$$

$$max(R_y) \leq field^h \tag{4}$$

2.1.2 Non Overlapping Constraint

The *non overlapping constraint* imposes all the circuits to be placed inside the plate in such a way that none of them overlaps any other circuit. Formally we have:

$$\begin{aligned}(X_i + widths_i \leq X_j \ \vee \ X_i \geq X_j + widths_j) \wedge \\ \wedge \ (Y_i + heights_i \leq Y_j \ \vee \ Y_i \geq Y_j + heigths_j), \\ \forall \ i, j \in [1, ..., n]\end{aligned} \tag{5}$$

2.1.3 Cumulative constraints

The *Cumulative Scheduling Constraint* or *Scheduling under resource constraint* is a constraint often used in scheduling problems. This constraint imposes that, given a set of tasks T, each described by a starting point s, a duration d and a resource consumption r, at each point in time the cumulated resources of the set of tasks that overlaps that point to be strictly smaller than a certain value R imposed. So, it's possible to use this constraint considering that:

- the starting point s is the X coordinate of each block (Y coordinate for width constraint).
- the duration d is the *width* of each block (*height* for width constraint).
- the resource usage r is the *height* of each block (*width* for width constraint).
- the resource usage limit R is the height of the board h (w for width constraint).

so, Formally:

$$\sum_{i|(X_i \leq u \leq (X_i + widths_i} heights_i \leq h \quad \forall \ u \ in \ widths \tag{6}$$

$$\sum_{i|(Y_i \leq u \leq Y_i + heights_i} widths_i \leq w \quad \forall \ u \ in \ heights \tag{7}$$

2.1.4 Rotation

All the constraints introduced until now have no care about rotation. In order to preserve the usability of the models in case circuits' rotation is allowed, a new variable needs to be introduced: *rotation*. This variable is defined as an *array* of boolean values and $rotation_i$ describe whether or not the $i-th$ circuits has been rotated. In addiction to this variable a rule that put in relationship this variable with the height and the width of the circuits needs to be introduced, in particular:

$$widths^a = [widths_i \cdot (1 - rotations_i) + heights_i \cdot rotations_i, ...] \quad \forall \ i \ \in [1, ..., n]$$

$$heights^a = [heights_i \cdot (1 - rotations_i) + widths_i \cdot rotations_i, ...] \quad \forall \ i \ \in [1, ..., n]$$

So, $widths^a$ and $heights^a$ represents the new *widths* and *heights* array. Based on the value of $rotation_i$ the values of $widths_i$ and $heights_i$ could be swapped in $widths_i^a$ and $heights_i^a$.

2.1.5 Relative Y

It constrains the new blocks to be placed on top of the older ones.

$$Y_i = pos\_y\_full_{X_i} - heights_i^a \forall \ i \ \in [1, ..., n] \tag{8}$$

2.1.6 Block Order

This constrains the second block not to be placed under the first one.

$$\begin{aligned} X_2 \in [X_1, ..., X_1 + widths_1] \vee \\ X_2 + widths_2 \in [X_1, ..., X_1 + widths_1] \\ \implies Y_1 \leq Y_2 \end{aligned} \tag{9}$$

2.1.7 Field Initialization

This constrains constructs a binary matrix where each position occupied by a block is set to one.

$$\begin{aligned} field_{i,j} = 1 \iff (\exists k \in [1, ..., n] \mid i \in [blocks_k^x, ..., block_k^x + blocks_k^w) \wedge \\ j \in [blocks_k^y, ..., block_k^y + blocks_k^h)) \vee \\ (\exists k \in [1, ..., 2] \mid i \in [X_k, ..., X_k + widths_k^a) \wedge \\ j \in [Y_k, ..., Y_k + heights_k^a)) \\ \forall\ i\ \in [1, ..., n],\ in [1, ..., n] \end{aligned} \tag{10}$$

2.1.8 Search Strategies

MiniZinc offers different possible search strategies according to the type of problem it has to deal with. After some test, since the problem stands on array search, a **sequential search** has been chosen together with an **integer search** (according to the assumption made before) for each element. The optimization goal is to maximize the number of full rows and the longest row while minimizing the number of unreachable tiles. At the same time a priorities vector is used to weight the total maximization formula, which could be expressed as follows:

$$\begin{aligned} priorities_1 \cdot full\_rows - \\ priorities_2 \cdot h + \\ priorities_3 \cdot longest\_row - \\ priorities_4 \cdot unreachable) \end{aligned}$$

3 Conclusions

In this report are included the listings of the miniZinc code, while the python code is available on github. In the following table are summed some test on game of the algorithm developed. The *step per second* field indicates the number of times per second the block is moved down which is directly dependant on the *level* field. All those test has been executed with the following set of weights: [18, 10, 3, 10], the number of completed rows has the highest priorities while the height and the number of unreachable boxes has the same priorities.

level	step per second	score
1	2	7320
1	2	12910
1	2	9960
2	4	6800
2	4	5630
3	6	2620
3	6	1950
4	8	140

As can be seen the algorithm has good performances on the overall. The main downside is the execution time, which is too slow and reduces the performances on higher levels. Some improvements could be made by reducing the search space and considering the symmetry of the problem and by fine tuning the vector of weights according to the best performances.

4 MiniZinc Code

Inputs and Variables :

```
include "globals.mzn";
include "diffn.mzn";

%---FUNCTIONS---%

%return the index of the last element in an array
function var int: find_last(array[int] of var int: x, var int: el)=
    max( [ if x[i] == el then i else 0 endif | i in index_set(x)]);



%---INPUTS---%
int: field_w = 10;                                  %width of the field
int: field_h = 20;                                  %height of the field
int: n;                                             %number of already existing blocks
array[1..field_h, 1..field_w] of var 0..1: field;   %a binary matrix representing the
    field


set of int: blocks = 1..n;
set of int: new_blocks = 1..2;
array[blocks] of int: blocks_x;                     %x position of blocks
array[blocks] of int: blocks_y;                     %y position of blocks
array[blocks] of int: blocks_w;                     %blocks width
array[blocks] of int: blocks_h;                     %blocks height



%---VARS---%
array[1..field_w] of var int: pos_y_full;           %relative y of blocks

array[new_blocks] of int: widths;                   %width of the blocks to place
array[new_blocks] of int: heights;                  %height of the blocks to place

array[new_blocks] of var bool: rotations;           %rotations array

%definition of width and hegiths according to the rotations vector
array[new_blocks] of var int: actual_widths = [(widths[i]*(1-rotations[i]) + heights[i]*
    rotations[i]) | i in new_blocks];
array[new_blocks] of var int: actual_heights = [(widths[i]*rotations[i]) + heights[i
    ]*(1-rotations[i])| i in new_blocks];

var 0..field_h: full_rows;                          %number of completed rows
var 0..field_w: longest_row;                        %length of the longest consicutive
    row
var 0..field_w*field_h: unreachable;                %number of unreachable boxes
var 1..field_h: h;                                  %height

array[new_blocks] of var 0..field_w-1: pos_x;       %position x of the blocks to place
array[new_blocks] of var 0..field_h-1: pos_y;       %position y of the blocks to place

array[1..4] of int: priorities = [18,10,3,10];        %multiplicative weights
```

Board Boundaries Constraint (3),(4):

```
% BOARD BOUNDARIES
constraint max(i in new_blocks)(pos_x[i] + actual_widths[i]) <= field_w;
constraint max(i in new_blocks)(pos_y[i] + actual_heights[i]) <= h;
```

Non Overlapping Constraint (5):

```
% NON OVERLAPPING
constraint diffn(blocks_x++pos_x, blocks_y++pos_y, blocks_w++actual_widths, blocks_h++
    actual_heights);
```

Cumulative Constraint (6),(7):

```

% CUMULATIVE
constraint cumulative(blocks_y++pos_y, blocks_h++actual_heights, blocks_w++actual_widths
    , field_w);
constraint cumulative(blocks_x++pos_x, blocks_w++actual_widths, blocks_h++actual_heights
    , h);
```

Field Initialization (10):

```
%FIELD INITIALIZATION
constraint forall(
              i in 0..field_w-1,
              j in 0..field_h-1)(
    exists(block in blocks)(
        ((i in blocks_x[block]..blocks_x[block]+blocks_w[block]-1
        /\j in blocks_y[block]..blocks_y[block]+blocks_h[block]-1)))
        \/
    exists(block in new_blocks)(
        (i in pos_x[block]..pos_x[block]+actual_widths[block] - 1
        /\j in pos_y[block]..pos_y[block]+actual_heights[block] - 1))
        <-> (field[j+1, i+1] = 1)
);
```

Relative Y (8):

```
%REALTIVE Y
constraint forall(i in new_blocks)(pos_y[i] = pos_y_full[pos_x[i] + 1] - actual_heights[
    i]);
```

Block order (9):

```
% BLOCK ORDER
constraint (pos_x[2] in pos_x[1]..pos_x[1]+widths[1] \/
            pos_x[2]+widths[2] in pos_x[1]..pos_x[1]+widths[1]) ->
            pos_y[1] < pos_y[2];
```

Variable assignment :

```
% VARIABLE ASSIGNMENT
constraint pos_y_full = [find_last(field[..,i], 1)| i in 1..field_w];

constraint let {array[1..field_h] of var int: lenghts = [count(field[j,..],1) | j in 1..
    field_h]} in longest_row = max(i in 1..field_h)(lenghts[i]);

constraint full_rows = sum([count(field[i,..], 1) div 10 == 1| i in 1..field_h]); %
    number of completed rows

constraint unreachable = sum(
    [j - k+1 | i in 1..field_w, j in 1..field_h-1, k in 1..field_h-1 where j>=k /\ field
    [j+1,i]==1 /\ (field[k-1,i]==1 \/ k==1) /\ count(field[k..j, i], 0) == j-k+1]
);
```

Search Strategy :

```
%---SEARCH STRATEGY---%
solve :: seq_search([
    int_search(pos_x, first_fail, indomain_min),
    int_search(rotations, first_fail, indomain_min)])
    maximize (priorities[1]*full_rows + priorities[3]*longest_row - priorities[2]*h -
    priorities[4]*unreachable);
```