Tetris Filling Problem

B10901016 Yan Sheng Qiu

November 27, 2023

1 Problem Statement

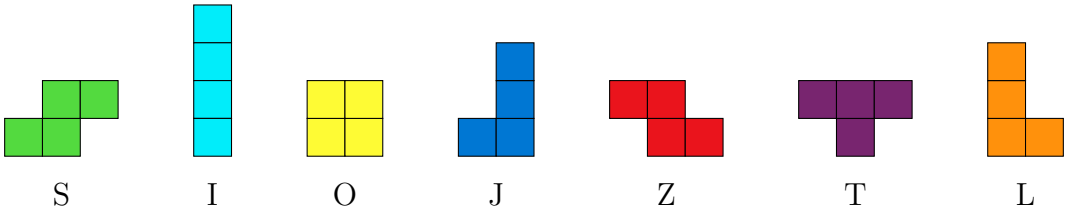


Figure 1: Tetris Blocks

Given a $m \times n$ rectangle board, and a subset of Tetris blocks, find a way to fill the board with the blocks. The blocks can be rotated, but cannot be flipped or overlapped. The blocks can be placed anywhere on the board, but cannot be placed outside the board.

2 Solution

This problem can be modeled as a SAT problem, and we can use SAT solver, called MiniSAT, to solve this problem. In the following subsections, we will introduce the encoding of the problem and the implementation of the algorithm.

2.1 Board and Cells

Cell in the i-th row and j-th column of a board is represented by a_{ij}.

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}

Figure 2: A 4×5 board example.

Actually, necessarily, we need to add "margins" with a width of 3 blocks to the board, for the convenience of the encoding the boundary conditions. The board with margins is shown below.

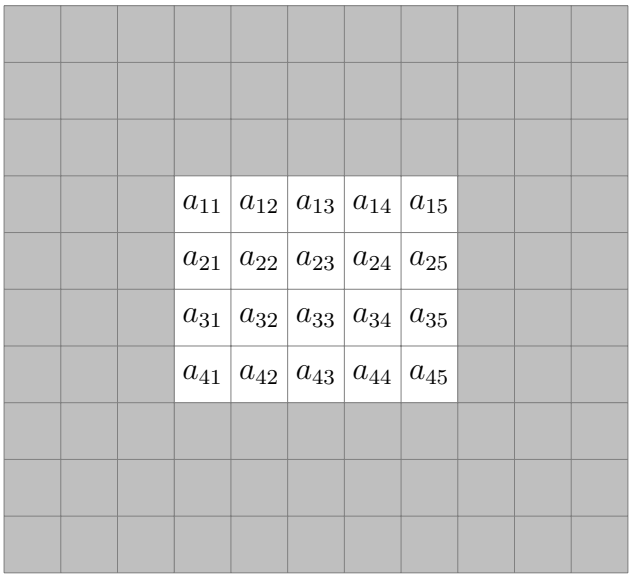


Figure 3: A 6×8 board example with margins.

Note that the original number of the row and the column is preserved. For a margin cell, we still use a_{ij} to denote it, so representation like $a_{(-2)(-1)}$ is used, indicating the cell is on the -2-nd row and the -1-st column.

2.2 Blocks and Cell states

To mark a occupied cell with the information of the block occupying it, each cell has a member array **Cell States**, containing binary variables, to represent the state of the cell. For example, if the cell is in kth state, the kth variable in the array is set to 1, and the others are set to 0. We use $a_{ij}[k]$ to denote the kth variable in the array of a_{ij}.

To define the states of the cells, first we dive into **the four rotation states of an L block**.

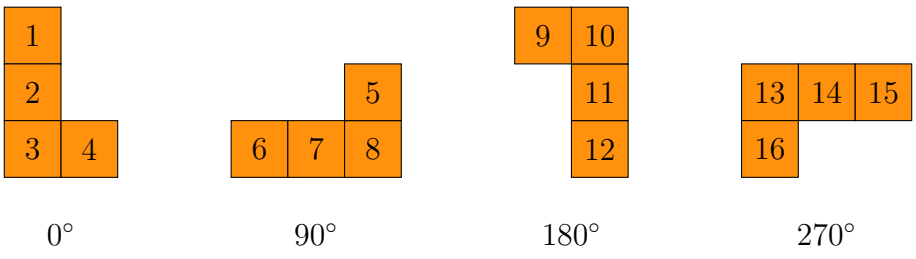


Figure 4: The four rotation states of an L block, labeled by the cell states.

Starting from the rotation state which bears the resemblance to the alphabet shape, we label the states of the cells strictly from top to bottom and from left to right. When the cells in current rotation state are all assigned with states, we can rotate the block by 90 degrees counterclockwise, and assign the states to the cells using the same rule again. We repeat this process until the block is rotated by 270 degrees counterclockwise. Note that after the cell state assignment in a block ends, for example in the L block we end with 16, the next block will start with 17.

The rotation states of the other blocks are defined in the same way. Specifically, note that not all the blocks have four rotation states. The rotation states of the blocks are listed below.

Block	Number of roatation states
S	2
I	2
O	1
J	4
Z	2
T	4
L	4

Table 1: The number of rotation states of the blocks.

Also, note the order of the blocks in the cell state assignment is randomly determined by the author, "S, I, O, J, Z, T, L".

2.3 Variables

For a subset of the Tetris blocks, let the total number of cell states be N_s. Thus, each cell in the board contains N_s binary variables, and given the board size $m \times n$, the total number of variables $N_v = (m+6)(n+6)N_s$ (Do not forget the margin cells!).

2.4 Clauses

Three types of clauses are considered in this solution, regarding respectively the **Uniqueness Conditions**, the **Placement Conditions** and the **Boundary Conditions**.

2.4.1 Uniqueness Conditions

The **Uniqueness Conditions** are used to ensure that each cell is occupied by at most one block. The clauses are as follows:

$$\bigwedge_{i=1}^{m}\bigwedge_{j=1}^{n}\left[\left(\bigvee_{k=1}^{N_s} a_{ij}[k]\right)\wedge \bigwedge_{(k_1,\ k_2)\in\left\{(u,\ v)\middle|(u,\ v)\in\left(\{t\}_{t=1}^{N_s}\right)^2,\ u<v\right\}} \left(\neg a_{ij}\left[k_1\right]\vee\neg a_{ij}\left[k_2\right]\right)\right].$$

Note the first part of the clause is used to ensure that the cell is occupied by at least one block, and the second part is used to ensure that the cell is occupied by at most one block.

2.4.2 Boundary Conditions

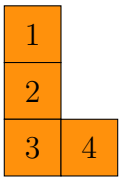
The **Boundary Conditions** are simple, which is "All margin cells are empty". The clauses are as follows:

$$\bigwedge_{(i,\ j)\in\{(u,\ v)|a_{uv}\text{ is a margin cell}\}}\ \bigwedge_{k=1}^{N_s}\neg a_{ij}[k]$$

To put it in words, all cell state variables of the margin cells are set to 0.

2.4.3 Placement Conditions

The **Placement Conditions** are used to ensure that each block is placed on the board. It is rather hard to write down a general formula, so we will use the L block as an example.



$0°$

Figure 5: The L block in the first rotation state.

The clauses are as follows:

$$\begin{aligned}
&\left[(\neg a_{ij}[1]\wedge a_{(i+1)j}[2])\wedge(\neg a_{ij}[1]\wedge a_{(i+2)j}[3])\wedge(\neg a_{ij}[1]\wedge a_{(i+2)(j+1)}[4])\right]\\
\wedge&\left[(\neg a_{ij}[2]\wedge a_{(i-1)j}[1])\wedge(\neg a_{ij}[2]\wedge a_{(i+1)j}[3])\wedge(\neg a_{ij}[2]\wedge a_{(i+1)(j+1)}[4])\right]\\
\wedge&\left[(\neg a_{ij}[3]\wedge a_{(i-2)j}[1])\wedge(\neg a_{ij}[3]\wedge a_{(i-1)j}[2])\wedge(\neg a_{ij}[3]\wedge a_{i(j+1)}[4])\right]\\
\wedge&\left[(\neg a_{ij}[4]\wedge a_{(i-2)(j-1)}[1])\wedge(\neg a_{ij}[4]\wedge a_{(i-1)(j-1)}[2])\wedge(\neg a_{ij}[4]\wedge a_{i(j-1)}[3])\right]
\end{aligned}$$

Note ach subclause is an implication of form $a_{ij}[k_1] \rightarrow a_{(i+s_i)(j+s_j)}[k_2]$, where $(s_i,\ s_j)$ is the shift to the cell in the relative position, namely, "if the cell is in state k_1, then the cell in the relative position is in state k_2". We generate clauses like this for all rotation states of all blocks in the subset.

To sum up, we use the **Uniqueness Conditions** to ensure that each cell is occupied by at most one block, the **Boundary Conditions** to ensure that all margin cells are empty, and the **Placement Conditions** to ensure that each block is placed on the board. These clauses are fed into the MiniSAT solver, and the solution is obtained.

3 Implementation

The implementation of the solution is written in C++. The file structure is as follows:

```
├── LICENSE
├── Makefile
├── README.md
├── bin
│   └── TetrisSAT
├── doc
│   └── TetrisSAT.pdf
├── outputs
│   └── output.png
└── src
    ├── AuxFuncs.cpp
    ├── AuxFuncs.h
    ├── Cell.h
    ├── Clauses.cpp
    ├── Clauses.h
    ├── IO.cpp
    ├── IO.h
    ├── SATtest.cpp
    ├── TetrisData.cpp
    ├── TetrisData.h
    ├── TetrisSolver.cpp
    ├── TetrisSolver.h
    ├── lodepng
    │   └── ..
    ├── main.cpp
    ├── minisat
    │   └── ..
```

Figure 6: File structure of the project.

3.1 Cell.h, Cell.cpp

The `Cell.h` and `Cell.cpp` files define the necessary functions regarding cells.

3.2 Clauses.h, Clauses.cpp

In `Clauses.h` and `Clauses.cpp` files, three types of clauses are generated, namely, the **Uniqueness Conditions**, the **Placement Conditions** and the **Boundary Conditions**.

3.3 IO.h, IO.cpp

The `IO.h` and `IO.cpp` files define the functions regarding input and output. Package `lodepng` is used to read and write `png` files.

3.4 TetrisData.h, TetrisData.cpp

The `TetrisData.h` and `TetrisData.cpp` files define the shapes, rotation states, and the colors of the Tetris blocks.

3.5 TetrisSolver.h, TetrisSolver.cpp

The `TetrisSolver.h` and `TetrisSolver.cpp` files redefine the SAT solver, which is MiniSAT in this case.

3.6 AuxFuncs.h, AuxFuncs.cpp

The `AuxFuncs.h` and `AuxFuncs.cpp` files define the auxiliary functions, including the functions to generate the clauses.

3.7 main.cpp

The `main.cpp` file summarizes the whole process of the program. The clasues are combined into a CNF (Conjunctive Normal Form) formula, and the formula is fed into the MiniSAT solver. If the formula is satisfiable, the solution is obtained, and the board is drawn using `lodepng` package. CPU time is recorded before and after the solution process of the MiniSAT solver.

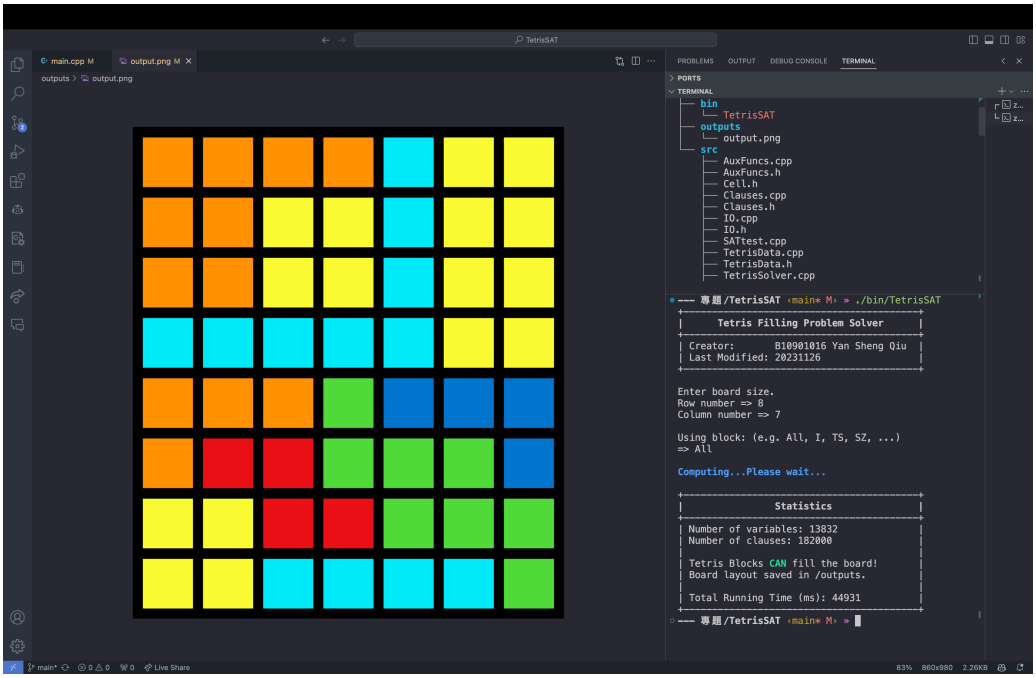


Figure 7: The solution of the 8×7 board with the subset of blocks "S, I, O, J, Z, T, L".

4 Experimental Results

We use the subset of blocks "S, I, O, J, Z, T, L" to fill the boards with different sizes, from 5×5 to 15×15. The experimental results are shown in the table below, with the unit being microsenconds. Note if the total number of cells is not divisible by 4, we skip the measurement.

m \ n	5	6	7	8	9	10	11	12	13	14	15
5	-	-	-	27750	-	-	-	51192	-	-	-
6	-	22228	-	35526	-	49987	-	66766	-	67636	-
7	-	-	-	44375	-	-	-	84220	-	-	-
8	26002	35317	43987	53299	63883	77271	90097	102070	117240	133058	148925
9	-	-	-	65382	-	-	-	126076	-	-	-
10	-	50533	-	76884	-	108679	-	150066	-	201125	-
11	-	-	-	92296	-	-	-	193371	-	-	-
12	52284	69039	87036	105384	127987	149709	173821	203871	231921	259510	295343
13	-	-	-	119340	-	-	-	230530	-	-	-
14	-	88098	-	135169	-	191260	-	260026	-	339192	-
15	-	-	-	152305	-	-	-	296178	-	-	-

Table 2: The experimental results of different sizes of boards.

5 Observations

- With all blocks used, all the boards with size $m \times n$ where $m \geq 5$ and $n \geq 5$ are completely fillable (Actually, intuitively, we can fill them with simple O blocks.).
- The experimental time compelixity is about $O(mn)$.
- There are always various kinds of blocks in the solution.

6 Future Work

- The experimental time complexity is about $O(mn)$. Which conditions can be added to make the time complexity lower?
- Are there any other ways to model this problem?
- Can we fill the board with sufficient size (e.g. $m \geq 5$, $n \geq 5$) completely without O blocks and I blocks?