Internship report
The Complexity of *Tetris*

Anita Dürr, Supervisor: Michail Lampis

January - June 2020

Abstract

This report presents different time complexity results on decision problems concerning the famous game *Tetris*. As a first contribution, we show that the Tetris Clearing (with rotation) problem stays NP-complete in the strong sense even by making an additional constraint on the game rules. We also use dynamic programming technique to solve a special case of the problem. In a second part, we give a polynomial algorithm that can be used to solve the Domino Surviving with rotation problem with *2-tris*. We additionally make an approximation of the optimal playing solution to the Domino Clearing with rotation problem. Finally, we prove that the Domino Surviving without rotation problem is NP-complete.

Contents

1 Formal rules of the *Tetris* game **4**

2 Solving the Tetris Clearing problem **5**
2.1 Strong NP-Completeness . . . 5
2.1.1 Tetris Clearing is in NP . . . 6
2.1.2 Reduction from 3-Partition to Tetris Clearing . . . 6
2.1.3 Proof of equivalent instances . . . 7
2.2 Dynamic Programming approach . . . 11
2.2.1 A simpler partition problem . . . 11
2.2.2 Tetris Clearing without holes . . . 13

3 *Domino*: a variant of *Tetris* **16**
3.1 Surviving with rotation . . . 16
3.1.1 Identify reachable rows . . . 17
3.1.2 Fixing a piece : use of a support . . . 18
3.1.3 Completing a hole . . . 20
3.1.4 Clear a row . . . 25
3.2 Clearing with rotation . . . 26
3.3 Surviving without rotation . . . 26

Bibliography **28**

Introduction

As part of the research work to be done in CPES 3, I chose to work with Michail Lampis[1] on the complexity of the game *Tetris*. In the original game, a player has to move different geometric objects (called tetrominoes) that are falling into the playing field. Completed lines disappear and the player can continue to fill the vacated space. The game ends when the player loses, that is when an object gets stuck at the top of the field. The player can technically never win, but the objective is to maximize his or her score, which increases as rows are cleared. Part of the game difficulty stems from the uncertainty on the type of piece that will fall as well as the speed of falling. In this work however, we are interested in an offline version of *Tetris*, where the sequence of falling pieces is given in advance. The formal rules of such a game are described in Section 1.

This game version was already studied in many papers and proved to be NP-complete (see below for the definition) in [3] by Demaine et al. and in [1] by Breukelaar et al.. We focus on this problem under different constraints in Section 2, proposing another way to do the reduction (in 2.1) and a dynamic programming algorithm (in 2.2.2).

Further work was done by Demaine et al. in [4] on variants of the game using other set of pieces. As some of the problems using domino pieces stayed open, we try to close them in Section 3. Section 3.1 pursue the idea that the DOMINO SURVIVING WITH ROTATION problem belongs to P, while we give an approximate solution to the DOMINO CLEARING WITH ROTATION problem in 3.2. In the final Section 3.3, we give a NP-completeness proof of the DOMINO SURVIVING WITHOUT ROTATION problem, inspired by a proof made in [4].

This study topic was a pretext to familiarize myself with notions of theoretical computer science, especially regarding the field of computational complexity. I give here definitions of those basic notions.

Decision problem A decision problem is a problem that can be answered by YES or NO. Typically whether or not a given instance verifies certain properties. As an example consider the problem where a graph is given as input and the question is whether or not there exists a cycle in it. The set of decision problems can be grouped into different classes regarding their computational complexity.

- **P** The class of decision problems that are *solvable* in polynomial time (in the size of the instance) by any (reasonable) computational model, for example a Turing machine.
- **NP** The class of problems that are *verifiable* in polynomial time by any computational model.
- **NP-hard** The class of problems to which *all* NP problems can be *reduced* in polynomial time.
- **NP-complete** The class of problems that are both in NP and NP-hard.

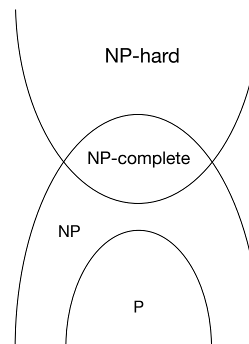


Figure 1: Diagram for P, NP, NP-complete, and NP-hard set of problems (under the assumption that P $\neq$ NP).

We specify the following vocabulary:

Solve Given an instance, decide whether or not it verifies the properties of the problem. For our example, decide whether or not there is a cycle in the given graph.

[1]assistant professeur at LAMSADE, Université Paris Dauphine PSL

Verify Given an instance and a proof of membership, verify that this instance verifies the properties of the problem according to this proof. In our example, a proof of membership of the given graph to the set of graph containing a cycle would be a set of edges. To verify this problem boils to verify that the given edges do form a cycle in the given graph.

Reduction Reducing a problem X to a problem Y is to give an algorithm A that solves X using an algorithm B which solves Y. The reduction is said in polynomial time if all parts of A except B have a polynomial complexity. As the algorithm A could ask B to solve only part of the instances of Y, we say that Y is at least as difficult than X (see Figure 2).

As the reduction is transitive, when proving that a problem is NP-hard, it suffices to reduce from a problem already proved to be NP-hard. The first problem to be proved to be NP-hard by reducing from all NP problems is the 3-SAT problem.

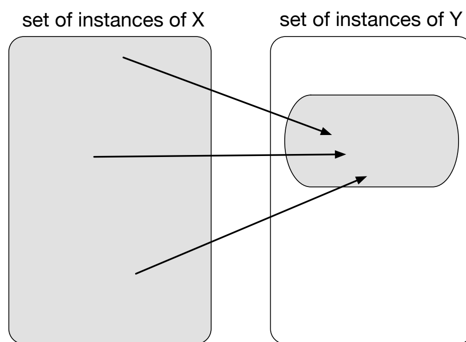


Figure 2: The idea of the reduction is to reduce the set of instances of the problem X to a subset of instances of the problem Y. Both sets should be equivalent in the sense that a YES-instance of X should be reduced to a YES-instance of Y and equivalently for NO-instances.

NP-complete in the strong sense. We additionally say that a problem is NP-complete in the **strong sense** if it remains NP-complete even if all input parameters written in unary notation[2] are bounded by a polynomial in the input size. The problem is NP-complete in the weak sense if the reduction that establishes its hardness uses binary numbers with a number of bits polynomial in the input size. Hence if the numbers are written in unary it would take exponential time. So strong NP-complete problems are even more "difficult" as weak NP-complete problems.

[2]In unary notation, the size needed to write a number is the number itself and not only its *log* as in binary notation.

1 Formal rules of the *Tetris* game

For the sake of completeness, we recall here the formal definition of *Tetris* given in [3, Section 2] by Demaine et al. on which we based our following work.

Gameboard. The gameboard is a grid of size $(h \times w)$ indexed from top-to-bottom and left-to-right. The (i, j)-th *gridsquare* (or *cell*) is either *empty* (*unoccupied*, *free*) or *filled* (*occupied*). In a legal gameboard, no row is completely filled (intuitively because it would clear) and there is no completely empty row that lies below any filled gridsquare. The grid is delimited by always-filled cells.

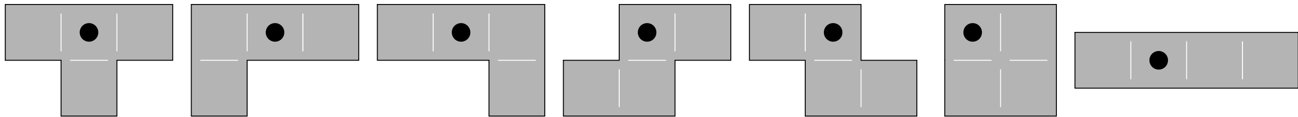


Figure 3: The seven *Tetris* pieces also called *tetrominoes*: T, L, J, S, Z, O and I. Their center gridsquare is marked with a dot.

Game pieces. The *Tetris* pieces are exactly the polygons that can be created by joining four unit gridsquares edge to edge. The *2-tris* piece is the only polygon created by joining two unit gridsquare. The pieces and their centers are shown respectively in Figure 3 and Figure 13. A *piece state* is a 4-tuple, consisting of:

1. a *piece type* (T, J, L, S, Z, O or I if playing the *Tetris* version ; in the *2-tris* version the piece type doesn't need to be specified as there is only one available) ;
2. an *orientation* (0°, 90°, 180°or 270°) : the number of degrees clockwise from the piece's base orientation (given in Figure 3 and Figure 13) ;
3. a *position of the piece's center* on the gameboard (chosen from $[\![1, h]\!] \times [\![1, w]\!]$). The position of an O is the location of the upper-left gridsquare of the O ;
4. the *value fixed or unfixed*, indicating whether the piece can continue to move.

When a piece arrives on the board, it is in its *initial piece state*, that is, the piece is in its base orientation, unfixed, and the highest gridsquare is placed into row 1 and the center into column $\lfloor w/2 \rfloor$.

Rotation model. We will only focus on the so-called *instantaneous rotation model.* From a piece state P, a piece can rotate to a piece state P' with an angle $\theta \in \{90°, -90°\}$. The rotation is illegal if P' occupies a filled gridsquare. In that case, $P = P'$. Otherwise, if $P = (t, o, (i, j), f)$ then $P = (t, (o + \theta) \mod 360°, (i, j), f)$.

Legal moves. For a piece that is fixed, no more moves are legal. The following moves are legal for a piece in the state $P = (t, o, (i, j), unfixed)$, with current gameboard B:

1. a *clockwise rotation* ($\theta = 90°$)
2. a *counterclockwise rotation* ($\theta = -90°$)
3. a *slide to the left* by one column: this is legal if the cells to the left of the piece are empty in B and the new piece state is $P' = (t, o, (i, j - 1), unfixed)$
4. a *slide to the right* by one column: similarly, the new piece state is $P' = (t, o, (i, j + 1), unfixed)$

5. a *drop* by one row: this is possible if all of the cells below the piece are empty in B and the new piece state is $P' = (t, o, (i+1, j), unfixed)$

6. a *fix*: this can be done if at least one gridsquare below the piece is filled in B and the new piece state is $P' = (t, o, (i, j), fixed)$

Clearing rows. The result of a fix for a piece P in a gameboard B is a new gameboard B' which is initially B with the cells of P filled. Additionally, if some row r in B' has every gridsquare filled, then row r is cleared. This means that for each row $r' \leqslant r$ in B', r' is replaced by row $r'-1$ of B'. Row 1 of B' is completely empty. This can be done multiple times so that multiple rows are cleared by fixing a single piece. A piece arrives on the board only after the previous one is fixed and the clearing process ended. Finally, if in B' the next piece's initial state is blocked by filled cells from entirely entering the gameboard, the game ends and the player *loses* or *doesn't survive*.

Tetris problems. In this document, we consider different problems all related to the above described rules. For each problem, we are given an initial gameboard partially filled and a finite ordered sequence of game pieces (tetrominoes or dominoes). The goal is either to clear the entire board (remove all occupied cells) or to survive (avoid stacking any piece above the upper limit of the board). Furthermore, we will consider both models where the piece can be rotated according to the above rule or not. In this second case, the pieces can only be moved to the left or right.

2 Solving the Tetris Clearing problem

In this section we are interested in the problem variant using *Tetris* pieces that can be rotated and where the goal is to clear the entire board. As the non rotating version will not be discussed here, we will refer to this problem only as Tetris Clearing problem.

Tetris Clearing

Input: A legal *Tetris* gameboard of size $(h \times w)$, potentially containing filled cells, and an ordered sequence of m *Tetris* pieces $p_1, p_2, \ldots, p_m$.

Question: Is it possible to play in such a way that the gameboard is left empty in the end ? Rotations are allowed.

While Demaine et al. showed in [3] that this problem is NP-complete in the strong sense, Breukelaar et al. gave a simplified reduction in [1]. In the next section, we make an additional restriction on the game rules. In our version, if a piece slides to the left or right while the cells below are empty, the piece is also moving downwards. With this kind of diagonal movement, both previous proofs become infeasible. However, the reduction made in 2.1 shows that even with this constraint, the problem stays NP-complete.

In 2.2 we focus on another restriction on the initial gameboard configuration. Studying dynamic programming techniques, we came up with an algorithm solving the Tetris Clearing problem in the case of an initial gameboard containing no *inner hole*.

2.1 Strong NP-Completeness

To show that the Tetris Clearing problem with diagonal movements is NP-complete, we first show that it is in NP before doing a polynomial reduction from the 3-Partition problem that was proved by Garey and Johnson in [5] to be NP-complete in the strong sense.

2.1.1 Tetris Clearing is in NP

Clearly, if we are given a sequence of moves in addition to the input, we can efficiently verify that playing those moves leads to the clearing of the entire board. In fact, all the algorithm has to do is to execute the moves and look at the result. The length of the sequence of moves is polynomial in the number of arriving pieces, so the verification can be done in polynomial time in the size of the input.

2.1.2 Reduction from 3-Partition to Tetris Clearing

The 3-PARTITION problem is the following:

3-PARTITION
Input: A sequence of positive integers $x_1, x_2, \ldots, x_{3n}$ such that $\sum_{i=1}^{3n} = B$, with $B/4n < x_i < B/2n$ for all $i \in [\![1, 3n]\!]$.
Question: Is it possible to partition $\{x_1, x_2, \ldots, x_{3n}\}$ into n subsets of the same sum ?

Note that for a YES-instance, the sum of the integers of each subset is equal to B/n. We can also prove that if such a partition is possible, each subset has necessarily size 3.

Proof. Let's call $A_1, A_2, \ldots, A_n$ the n subsets. Since $B/4n < x_i < B/2n$ for each x_i, by summing over the integers of any subset A_j ($j \in [\![1, n]\!]$), we also have:

$$\frac{B}{4n} \cdot Card(A_j) < \frac{B}{n} < \frac{B}{2n} \cdot Card(A_j)$$

Multiplying the left-hand inequality by $\frac{4n}{B}$ we get $Card(A_j) < \frac{B}{n} \cdot \frac{4n}{B} = 4$; while the right-hand inequality gives us $\frac{B}{n} \cdot \frac{2n}{B} = 2 < Card(A_j)$. Hence $2 < Card(A_j) < 4$ so $Card(A_j) = 3$, and this for any $j \in [\![1, n]\!]$. □

The reduction consists of defining an equivalent TETRIS CLEARING instance to any given 3-PARTITION instance. We will first construct from given $x_1, x_2, \ldots, x_{3n}$ and B a *Tetris* gameboard and a sequence of *Tetris* pieces. We then prove that the 3-PARTITION instance is a YES-instance if and only if the corresponding TETRIS CLEARING instance is a YES-instance. With the restriction made on the game rules where horizontal movement sometimes become diagonal, we decide to use pieces that will only fall down. Horizontal movement will only be made at the very beginning in order to place the pieces in the desired columns.

Nevertheless, even with this restriction the general idea of the reduction remains the same as both proofs in [3] and [1]. We construct n *buckets* corresponding to the n partitioning subsets for the 3-PARTITION problem. For each integer x_i we chose a sequence of pieces so that they must be placed into the same *bucket*. We also manage to have a legal 3-partition for $x_1, x_2, \ldots, x_{3n}$ exactly when the pieces in each *bucket* have the same height. The last column of the gameboard forms an empty *chimney* which prevents any row from being cleared until all piece sequences corresponding to the x_i are placed. The entire board can be cleared using the last pieces of the sequences and finally completing the *chimney*.

Tetris board The initial gameboard constructed from the 3-PARTITION instance is shown in Figure 4. It has width $4n + 1$ and height $4 \cdot B/n + 28 + \lfloor w/2 \rfloor$. The top $\lfloor w/2 \rfloor$ rows are needed to rotate the pieces and moving them into the right columns before entering the *buckets*. Each *bucket* consists of three adjacent columns that are empty except for the bottom two rows as shown in Figure 4. All *bucket*s are separated by a filled column. We call the last empty column a *chimney*. This board can be constructed by using n O-pieces for the bottom of the n buckets, and $n \cdot (B/n + 7)$ stacked I-pieces for the n filled columns.

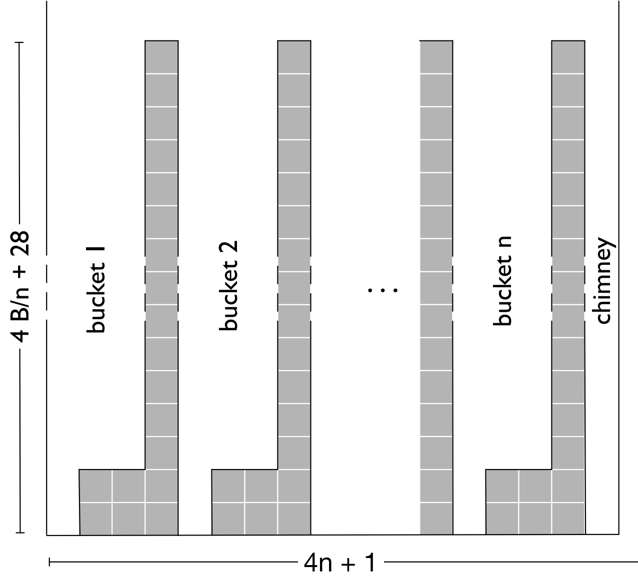


Figure 4: The initial gameboard in the reduction from 3-PARTITION to *Tetris*.

Tetris pieces The sequence of *Tetris* pieces consists of sequences of pieces corresponding to each x_i from the 3-PARTITION instance, followed by additional pieces used to close the *buckets* and the *chimney*. For each x_i, we begin with a J-piece, then take the sequence (T, T, Z) x_i times and end with (T, T, O, Z, L). These pieces are given for x_1 to x_{3n} in this order. After the pieces corresponding to x_{3n}, we have the J-piece $2n$ times (to close the *buckets*) and $B/n + 7$ I-pieces (to close the *chimney*).

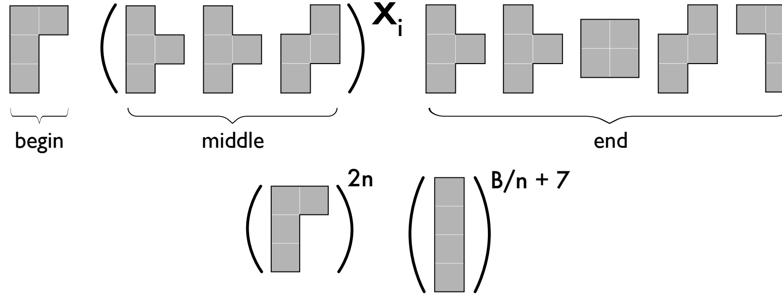


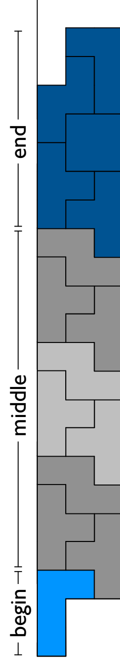
Figure 5: The piece sequence in the reduction from 3-PARTITION to *Tetris*.

Polynomial construction The construction of the gameboard uses $n + n \cdot (B/n + 7) = B + 8n$ pieces and the sequence of pieces is of length $\sum_{i=1}^{3n}(1 + 3 \cdot x_i + 5) + 2n + B/n + 7 = 3B + 20n + B/n + 7$. So the total number of pieces is linear in $n + B$. Moreover, the generation of the sequence can be done with a single loop in linear time and each piece is described by a constant size symbol. So the construction of the *Tetris* instance takes polynomial time in $n + B$.

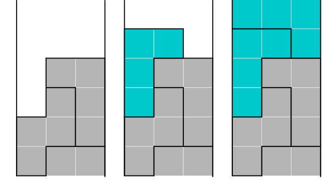
2.1.3 Proof of equivalent instances

We now have to prove that the 3-PARTITION instance and its corresponding *Tetris* instance are equivalent. The easiest direction to show is that for a YES-instance of 3-PARTITION, we can clear the entire gameboard. Let's consider that there is a partitioning of the x_i's into sets $A_1, A_2, ...A_n$ so that for all $j \in \{1, 2, \ldots, n\}$ $\sum_{a_i \in A_j} a_i = B/n$. To clear the board, we place all pieces associated with the set A_j in the j-th *bucket* of

the gameboard, as shown in Figure 6a. Once all the pieces associated to the x_i's are placed, each *bucket* is entirely filled, except for the top. We can then use the $2n$ J-pieces to close all *bucket*s by placing them on top, as shown in Figure 6b. At this point, the left part of the board is entirely filled. We can then use all remaining I-pieces to fill in the *chimney*, each time clearing the bottom four lines.



(a) Fill in a *bucket* with the pieces associated to the x_i's.



(b) Closing a *bucket* with 2 J-pieces.

Figure 6: Clear the board when the 3-PARTITION instance is a YES-instance.

To show the other direction, we proceed in several steps. Let's consider a YES-instance of the TETRIS CLEARING problem, that is we succeeded in clearing the board given the initial gameboard configuration and the ordered sequence of pieces described above.

First notice that every line of the $(4n+1) \times (4B/n+28)$ bottom rectangle of the board contains at least a filled cell, so every line of this rectangle needs to be completed in order to clear the board. Furthermore, the volume occupied by the pieces is independent of whether the given instance is a YES or NO instance and, resulting from the above clearing using the same pieces, we know that the pieces fit exactly the volume of the empty cells in the rectangle in question. The explicit volume of the pieces is (with s(α) the volume of the sequence of pieces α) :

$$
\begin{aligned}
V_{pieces} &= \sum_{i=1}^{3n} [s(begin) + s(middle) \cdot x_i + s(end)] + s(J) \cdot 2n + s(I) \cdot (B/n + 7) \\
&= \sum_{i=1}^{3n} [4 + 12x_i + 20] + 4 * 2n + 4(B/n + 7) \\
&= 12n + 12 \sum_{i=1}^{3n} x_i + 60n + 8n + 4B/n + 28 \\
&= 80n + 12B + 4B/n + 28 \quad \text{(since } \sum_{i=1}^{3n} x_i = B \text{)}
\end{aligned}
$$

On the other hand, the volume of the empty cells within the $(4n+1) \times (4B/n+28)$ rectangle is :

$$\begin{aligned} V_{board} &= s(bucket) \cdot n + s(chimney) \\ &= (3(4B/n+28) - 4) \cdot n + 4B/n + 28 \\ &= (12B/n + 80) \cdot n + 4B/n + 28 \\ &= 80n + 12B + 4B/n + 28 \end{aligned}$$

Hence the volume of the empty cells in the rectangle is exactly the volume occupied by all the pieces. Thus, in order to clear the board, all the pieces need to be placed in the $(4n+1) \times (4B/n+28)$ rectangle and no piece should stand above the $(4B/n+28)$-th line, or an empty cell would remain in the rectangle, making the clearing impossible.

We then notice that the unique piece that goes in the *chimney* without standing above the limit is an I-piece. This is trivially verified by placing all disposable pieces in the *chimney*. Using the same volume argument as above, we deduce that all I-pieces must go in the *chimney* and that all pieces coming before the first I-piece are used to entirely fill the *buckets*. This means that no row is cleared before the first I-piece arrives.

Let's show that the sequence of pieces corresponding to a given x_i have to be put in the way described in Figure 6a. We will show that the pieces corresponding to x_1 have to be placed in the same *bucket* in a manner that, at the end of it, the bottom of this *bucket* has exactly the same shape as the bottom of other empty *buckets*. By induction, it will follow that the pieces corresponding to any x_i have to be placed into the same *bucket*.

We consider the sequence (J, (T, T, Z) $\times$ x_1, T, T, O, Z, L) arriving in this order. Figure 7 shows that T, O, Z or L pieces cannot go into an empty *bucket* without leaving an empty or unreachable cell. A cross means that the cell is unreachable and a dot means that the cell is only reachable with an I-piece. Since all available I-pieces are used to close the *chimney*, those cells would remain empty. Hence, once the J-piece is placed in a given *bucket*, all following pieces have to be put into this same *bucket* and cannot be split into different ones.

To show that there is only one possible configuration for the pieces, we try all possible configurations for each arriving piece in Figures 8, 9 and 10. For each of them, there is only one way to place it without leaving empty cells. Since the sequence of *middle*-pieces leaves the bottom of the bucket unchanged, it can be repeated as often as we want.

So to sum up, a necessary condition to clear the board is to place the pieces corresponding to an x_i into the same *bucket* using the configuration of Figure 6a. This leaves the top of each *bucket* with empty cells. The only remaining pieces before the I-pieces are the $2n$ L-pieces. Once again, there is trivially only one possible way (shown in Figure 6b) to place them in order to *close the buckets*.

Finally, we need to show that each *bucket* contains exactly 3 series of pieces corresponding to the x_i's. We count $(4x_i + 8)$ filled lines by the series of pieces corresponding to x_i's (we don't consider the first two cells filled by the *begin* piece and compensate by considering that the two cells left empty by the *end* pieces are filled). So if A_j is the subset of x_i's for which the corresponding pieces were placed in the j-th *bucket*, the number of filled lines is $4\sum_{x_i \in A_j} x_i + 8 \cdot Card(A_j)$. On the other hand, a *bucket* contains $(4B/n + 24)$ lines filled with x_i pieces (we don't consider the two-line bottom notch and compensate by considering that the two first cells filled with the *closing* J-pieces are filled with x_i pieces). We deduce that $4B/n + 24 = 4\sum_{x_i \in A_j} x_i + 8 \cdot Card(A_j)$ and dividing by 4 and subtracting $2 \cdot Card(A_j)$ we obtain:

$$\sum_{x_i \in A_j} x_i = \frac{B}{n} + 6 - 2 \cdot Card(A_j)$$

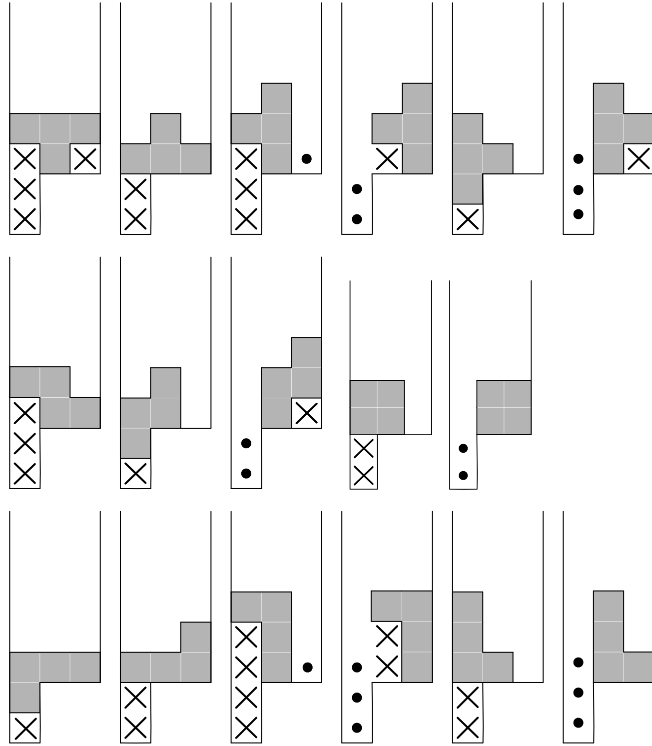


Figure 7: All possible configurations to place respectively a T, Z, O and L piece at the bottom of an empty *bucket*. A cross indicates an unreachable cell because it is shadowed by other pieces and a dot indicates an unreachable cell because only a I-piece could reach it.

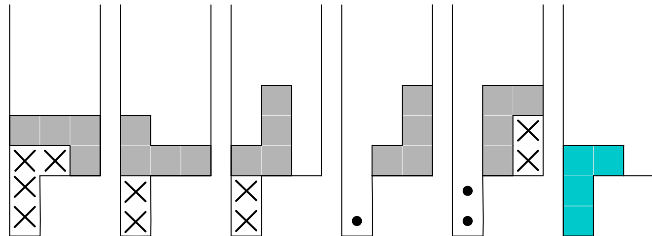


Figure 8: There is only one way to place the *begin* J-piece.

So if $Card(A_j) < 3 \Leftrightarrow Card(A_j) \leqslant 2$, then $\sum_{x_i \in A_j} x_i > B/n$. But since $\frac{B}{n} \cdot \frac{1}{4} < x_i < \frac{B}{n} \cdot \frac{1}{2}$ for each i, we also have by summing over A_j: $\sum_{x_i \in A_j} x_i < \frac{B}{n} \frac{Card(A_j)}{2} \leqslant B/n$, contradicting the above inequality. Hence $Card(A)$ has to be greater equal than 3. Using the same reasoning, we prove that if $Card(A_j) > 3 \Leftrightarrow Card(A_j) \geqslant 4$, then $\sum_{x_i \in A_j} x_i < B/n$ from the first equation and $\sum_{x_i \in A_j} x_i > B/n$ from the second one. Therefore $Card(A_j) = 3$. This means that in order to clear the board, each *bucket* was filled with exactly 3 sequences corresponding to x_i's. Furthermore $\sum_{x_i \in A_j} x_i = B/n$ for all *buckets* corresponding to A_j.

This shows that in order to fill the board, the pieces corresponding to an x_i are necessarily placed into the same *bucket* and that each *bucket* contains pieces corresponding to exactly 3 values of $\{x_1, x_2, \ldots, x_{3n}\}$ that sum up to B/n. Hence if the Tetris Clearing instance is a yes instance, by clearing the board a player finds a partition answer to the 3-Partition problem.

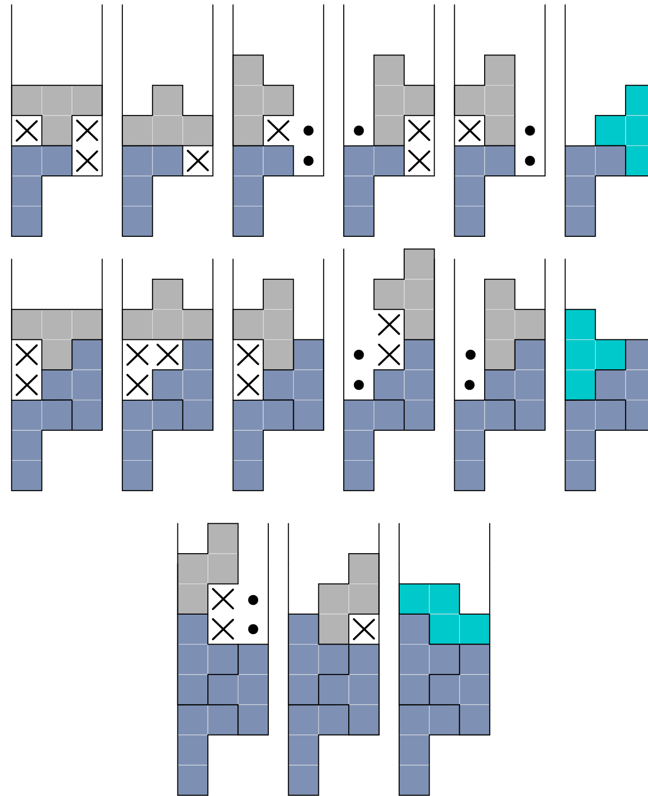


Figure 9: Placing the *middle* pieces.

This finishes the proof that the TETRIS CLEARING problem is NP-complete in the strong sense. □

2.2 Dynamic Programming approach

Dynamic programming is a technique that comes very handy, especially in combinatorial tasks. Reading Chapter 6 of [2] and solving some of the exercises, I devised a dynamic programming algorithm for a partition problem that I describe in 2.2.1. This inspired me to use a dynamic programming algorithm (in 2.2.2) for a special case of the TETRIS CLEARING problem. Dasgupta et al. present in [2, Chapter 6] dynamic programming as an

> algorithm paradigm in which a problem is solved by identifying a collection of subproblems and tackling them one by one, smallest first, using the answers to smaller problems to help figure out larger ones, until the whole lot of them is solved. [The subproblems should verify] a key property that allows them to be solved in a single pass : there is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to "smaller" subproblems, that is, subproblems that appear earlier in the ordering.

2.2.1 A simpler partition problem

The following partition problem is part of [2, Exercice 6.25]. Instead of partitioning the integers into n subsets as in the 3-PARTITION problem, the task is here to find only 3 partitions. To avoid any confusion with the 3-PARTITION problem, we name it the PARTITION IN 3 problem.

PARTITION IN 3
Input: A sequence $x_1, x_2, \ldots, x_n$ of positive integers.
Question: Is it possible to partition $\{x_1, x_2, \ldots, x_n\}$ into 3 subsets J, K, L of the same sum ?

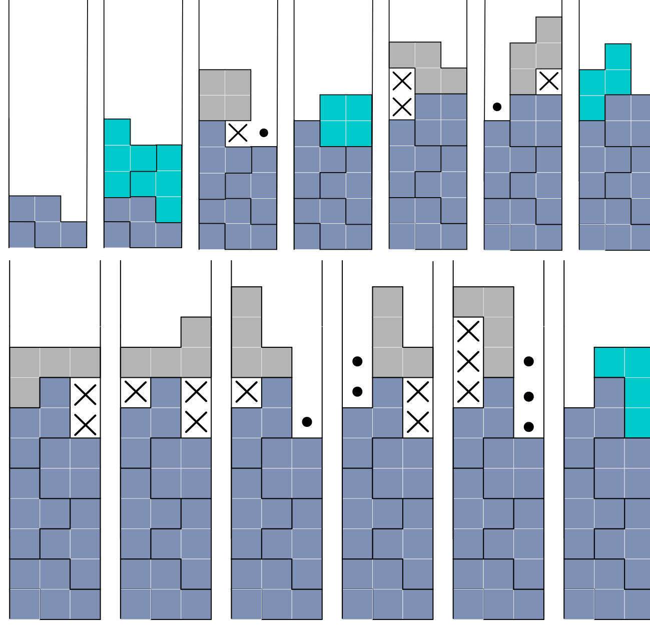


Figure 10: Placing the *end* pieces. To place the first two T-pieces is similar to place the two T-pieces from the *middle* sequence in Figure 9.

Note that for a YES-instance, $\sum_{x_i \in J} x_i = \sum_{x_i \in K} x_i = \sum_{x_i \in L} x_i = \frac{1}{3}\sum_{i=1}^{n} x_i$. Let's call $t := \frac{1}{3}\sum_{i=1}^{n} x_i$.

Resolution. Define $A(i, s_1, s_2, s_3)$ the subproblem answering whether or not there exists a partition of the first i integers $x_1, x_2, \ldots, x_i$ into J, K, L such that $\sum_{x_i \in J} x_i = s_1$, $\sum_{x_i \in K} x_i = s_2$ and $\sum_{x_i \in L} x_i = s_3$. The answer to all the subproblems $A(i, s_1, s_2, s_3)$ form a four-dimensional table. Our final objective is to solve $A(n, t, t, t)$.

For this, we need to express $A(i, s_1, s_2, s_3)$ in terms of smaller subproblems. Consider x_i the last integer used in the partition. x_i can be put either into J, K or L. This increases the sum of the partition containing x_i by x_i, while the sum of the two other partitions remain unchanged. Hence the relation between the subproblems is:

$$\begin{aligned} A(i, s_1, s_2, s_3) = {} & [(x_i \leqslant s_1) \wedge A(i-1, s_1 - x_i, s_2, s_3)] \\ & \vee [(x_i \leqslant s_2) \wedge A(i-1, s_1, s_2 - x_i, s_3)] \\ & \vee [(x_i \leqslant s_3) \wedge A(i-1, s_1, s_2, s_3 - x_i)] \end{aligned}$$

where each clause corresponds to assign x_i to respectively J, K and L. Finally, initialize $A(i, s_1, s_2, s_3)$ with:

$$\begin{aligned} A(0,0,0,0) &= \text{True} \\ A(0, s_1, s_2, s_3) &= \text{False} \quad \forall s_1, s_2, s_3 > 0 \\ A(i, 0, 0, 0) &= \text{False} \quad \forall i > 0 \end{aligned}$$

Each cell of the table of answers takes a constant time to compute, so this dynamic programming algorithm runs in time $O(nt^3)$. If $t = poly(n)$, the algorithm is polynomial. It can furthermore be improved to a $O(nt^2)$ time complexity: the size of the last partition L can be deduced from the size of J and K ($s_3 = \sum_{j=1}^{i} a_j - s_1 - s_2$).

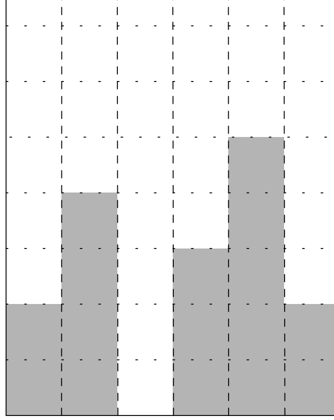


Figure 11: A *Tetris* gameboard without *inner hole*. The height of the six columns are respectively $T_1 = 2,\ T_2 = 4,\ T_3 = 0,\ T_4 = 3,\ T_5 = 5,\ T_6 = 2$. Notice that at least one column has the height 0 or rows would clear.

2.2.2 Tetris Clearing without holes

In a *Tetris* gameboard, we call *inner holes* empty cells for which there exists filled cells above. If we consider a *Tetris* gameboard without any inner hole, each column can be described only by its *height*: the level of the upper filled cell. An example of such a board is given in Figure 11. This constraint on the initial gameboard configuration allows us to use a dynamic programming algorithm to solve the TETRIS CLEARING (without inner hole) problem.

Algorithm. For this, we define the subproblem $A(i, t_1, t_2, \ldots, t_w)$ to be whether or not it is possible with the first i pieces to reach the gameboard configuration where the column j has height t_j for each $j \in [\![1, w]\!]$. To initialize the subproblems, call $T_1, T_2, \ldots, T_w$ the height of the columns in the initial gameboard and define:

$$\begin{aligned} A(0, T_1, T_2, \ldots, T_w) &= \text{True} \\ A(0, t_1, t_2, \ldots, t_w) &= \text{False} \quad \forall j \in [\![1, w]\!] \quad t_j \neq T_j \\ A(i, T_1, T_2, \ldots, T_w) &= \text{False} \quad \forall i > 0 \end{aligned}$$

Alike before, to express $A(i, t_1, t_2, \ldots, t_w)$ in terms of smaller subproblems, we consider all possible placings of the last piece p_i. However, for simplicity we would like to have only increasing t_j's. Therefore we virtually consider that completed rows never clear. In the normal case of a clearing, the heights t_j would all decrease by the number of cleared rows, but the height difference would stay unchanged. Hence virtually keeping the completed row in the gameboard doesn't changes the game rules. We just consider that the total height of the non-clearing board is greater than h. In fact, the board configuration has to verify at any time $\max t_i - \min t_i \leqslant h$. Our goal is to solve the subproblem $A(m, t, t, \ldots, t)$ for any $t \geqslant 0$.

If p_i is the last piece to be placed, for each of its possible piece type we consider all possible positions for which p_i can be placed on top of the board without leaving any *inner hole*. For each of those possibilities, we need as before to verify that the t_j's stay positive. The complete definition of $A(i, t_1, t_2, \ldots, t_w)$ is given in Equation 1. Illustration of cases 1b and 1e are given in Figure 12.

Complexity. This proves that there exists an algorithm for the TETRIS CLEARING WITHOUT INNER HOLE problem. However as the subproblems A form a $w + 1$-dimensional table, the complexity of the algorithm is exponential in w. Hence the problem remains nevertheless NP-complete. This was predictable

as the previous reduction uses a gameboard without holes and that can be won without making any. However, according to the result of [9] and the previous reduction, this complexity is close to optimal.

$$
\begin{aligned}
A(i,t_1,t_2,\dots,t_w) = (\max t_i - \min t_i \leqslant h) \quad \wedge & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(1a)}\\
(p_i == I)\wedge \quad & \bigvee_{j\in[\![1,w]\!]} \left\{ A(i-1,t_1,\dots,t_j-4,\dots,t_w)\wedge(t_j\geqslant 4) \right\} \quad \text{(1b)}\\
& \vee \bigvee_{j\in[\![1,w-3]\!]} \left\{\begin{array}{l} A(i-1,t_1,\dots,t_j-1,t_{j+1}-1,t_{j+2}-1,t_{j+3}-1\dots,t_w)\\ \wedge(t_j=t_{j+1}=t_{j+2}=t_{j+3})\\ \wedge(t_j\geqslant 1)\end{array}\right\} \quad \text{(1c)}\\
\vee (p_i == O)\wedge \quad & \bigvee_{j\in[\![1,w-1]\!]} \left\{\begin{array}{l} A(i-1,t_1,\dots,t_j-2,t_{j+1}-2,\dots,t_w)\\ \wedge(t_j=t_{j+1})\\ \wedge(t_j\geqslant 1)\end{array}\right\} \quad \text{(1d)}\\
\vee (p_i == L)\wedge \quad & \bigvee_{j\in[\![1,w-2]\!]} \left\{\begin{array}{l} A(i-1,t_1,\dots,t_j-2,t_{j+1}-1,t_{j+2}-1\dots,t_w)\\ \wedge(t_j=t_{j+1}=t_{j+2})\\ \wedge(t_j\geqslant 2)\end{array}\right\} \quad \text{(1e)}\\
& \vee \bigvee_{j\in[\![1,w-1]\!]} \left\{\begin{array}{l} A(i-1,t_1,\dots,t_j-1,t_{j+1}-3,\dots,t_w)\\ \wedge(t_j=t_{j+1})\\ \wedge(t_j\geqslant 3)\end{array}\right\} \quad \text{(1f)}\\
& \vee \bigvee_{j\in[\![1,w-2]\!]} \left\{\begin{array}{l} A(i-1,t_1,\dots,t_j-1,t_{j+1}-1,t_{j+2}-2,\dots,t_w)\\ \wedge(t_j=t_{j+1}=t_{j+2}-1)\\ \wedge(t_j\geqslant 1)\end{array}\right\} \quad \text{(1g)}\\
& \vee \bigvee_{j\in[\![1,w-1]\!]} \left\{\begin{array}{l} A(i-1,t_1,\dots,t_j-3,t_{j+1}-1,\dots,t_w)\\ \wedge(t_j=t_{j+1}+2) \qquad\qquad \wedge(t_j\geqslant 3)\end{array}\right\} \quad \text{(1h)}\\
\vee (p_i == J)\wedge \quad & \bigvee_{j\in[\![1,w-2]\!]} \left\{\begin{array}{l} A(i-1,t_1,\dots,t_j-1,t_{j+1}-1,t_{j+2}-2\dots,t_w)\\ \wedge(t_j=t_{j+1}=t_{j+2})\\ \wedge(t_j\geqslant 2)\end{array}\right\} \quad \text{(1i)}\\
& \vee \bigvee_{j\in[\![1,w-1]\!]} \left\{\begin{array}{l} A(i-1,t_1,\dots,t_j-1,t_{j+1}-3,\dots,t_w)\\ \wedge(t_j=t_{j+1}-2)\\ \wedge(t_j\geqslant 1)\end{array}\right\} \quad \text{(1j)}\\
& \vee \bigvee_{j\in[\![1,w-2]\!]} \left\{\begin{array}{l} A(i-1,t_1,\dots,t_j-2,t_{j+1}-1,t_{j+2}-1,\dots,t_w)\\ \wedge(t_j=t_{j+1}+1=t_{j+2}+1)\\ \wedge(t_j\geqslant 2)\end{array}\right\} \quad \text{(1k)}\\
& \vee \bigvee_{j\in[\![1,w-1]\!]} \left\{\begin{array}{l} A(i-1,t_1,\dots,t_j-3,t_{j+1}-1,\dots,t_w)\\ \wedge(t_j=t_{j+1})\\ \wedge(t_j\geqslant 3)\end{array}\right\} \quad \text{(1l)}
\end{aligned}
$$

$$\vee\ (p_i == S) \wedge \bigvee_{j\in[\![1,w-2]\!]} \left\{ \begin{array}{l} A(i-1, t_1, \ldots, t_j - 1, t_{j+1} - 2, t_{j+2} - 1 \ldots, t_w) \\ \wedge\ (t_j = t_{j+1} - 1 = t_{j+2} - 1) \\ \wedge\ (t_j \geqslant 1) \end{array} \right\} \tag{1m}$$

$$\vee \bigvee_{j\in[\![1,w-1]\!]} \left\{ \begin{array}{l} A(i-1, t_1, \ldots, t_j - 2, t_{j+1} - 2, \ldots, t_w) \\ \wedge\ (t_j = t_{j+1} + 1) \\ \wedge\ (t_j \geqslant 3) \end{array} \right\} \tag{1n}$$

$$\vee\ (p_i == Z) \wedge \bigvee_{j\in[\![1,w-2]\!]} \left\{ \begin{array}{l} A(i-1, t_1, \ldots, t_j - 1, t_{j+1} - 2, t_{j+2} - 1 \ldots, t_w) \\ \wedge\ (t_j = t_{j+1} = t_{j+2} + 1) \\ \wedge\ (t_j \geqslant 2) \end{array} \right\} \tag{1o}$$

$$\vee \bigvee_{j\in[\![1,w-1]\!]} \left\{ \begin{array}{l} A(i-1, t_1, \ldots, t_j - 2, t_{j+1} - 2, \ldots, t_w) \\ \wedge\ (t_j = t_{j+1} - 1) \\ \wedge\ (t_j \geqslant 2) \end{array} \right\} \tag{1p}$$

$$\vee\ (p_i == T) \wedge \bigvee_{j\in[\![1,w-2]\!]} \left\{ \begin{array}{l} A(i-1, t_1, \ldots, t_j - 1, t_{j+1} - 2, t_{j+2} - 1 \ldots, t_w) \\ \wedge\ (t_j = t_{j+1} - 1 = t_{j+2}) \\ \wedge\ (t_j \geqslant 1) \end{array} \right\} \tag{1q}$$

$$\vee \bigvee_{j\in[\![1,w-1]\!]} \left\{ \begin{array}{l} A(i-1, t_1, \ldots, t_j - 3, t_{j+1} - 1, \ldots, t_w) \\ \wedge\ (t_j = t_{j+1} - 1) \\ \wedge\ (t_j \geqslant 3) \end{array} \right\} \tag{1r}$$

$$\vee \bigvee_{j\in[\![1,w-2]\!]} \left\{ \begin{array}{l} A(i-1, t_1, \ldots, t_j - 1, t_{j+1} - 2, t_{j+2} - 1 \ldots, t_w) \\ \wedge\ (t_j = t_{j+1} = t_{j+2}) \\ \wedge\ (t_j \geqslant 2) \end{array} \right\} \tag{1s}$$

$$\vee \bigvee_{j\in[\![1,w-1]\!]} \left\{ \begin{array}{l} A(i-1, t_1, \ldots, t_j - 1, t_{j+1} - 3, t_{j+2} - 1 \ldots, t_w) \\ \wedge\ (t_j = t_{j+1} - 1) \\ \wedge\ (t_j \geqslant 2) \end{array} \right\} \tag{1t}$$

Figure 12: Placing a vertical I in column 3 makes $A(1, 2, 4, 4, 3, 5, 2)$ *True* because condition 1b is verified with $j = 3$. If on the contrary the first piece is a J, placing it in column 1 and 2 makes $A(1, 5, 5, 0, 3, 5, 2)$ *True* because condition 1e is verified with $j = 1$.

3 *Domino*: a variant of *Tetris*

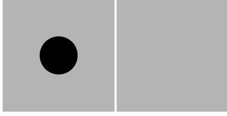


Figure 13: The *2-tris* game piece (or domino) which center is marked with a dot.

In this section we are dealing with *2-tris* related problems. *2-tris* is a variant of *Tetris* where all pieces have size 2: there is only one possible piece, a domino (see Figure 13). The only difference in the game rules to *Tetris* is the set of game pieces. Paper [4] shows that the DOMINO CLEARING WITHOUT ROTATION problem is NP-complete, yet the DOMINO CLEARING WITH ROTATION and the DOMINO SURVIVING with and without rotation problems are left opened by the authors. We attempt respectively in Section 3.1 and Section 3.2 to prove the polynomial complexity of the DOMINO SURVIVING WITH ROTATION and the DOMINO CLEARING WITH ROTATION problems. Finally, we prove that DOMINO CLEARING WITHOUT ROTATION is NP-complete in 3.3.

3.1 Surviving with rotation

DOMINO SURVIVING WITH ROTATION
Input: A 2-tris gameboard of size $(h \times w)$ partially filled with blocks and m 2-tris pieces.
Question: Is it possible to place all the pieces without any of them standing above the upper limit of the board ? Rotations are allowed.

In the paper [4], the authors have the intuition that the DOMINO SURVIVING WITH ROTATION problem is in P. They prove that if the top row of the board is empty, then the player can survive an arbitrarily long sequence of dominoes. The proof relies on the fact that in this case the player could manage to fill every empty cell of the second line by placing vertical pieces until this line completes and clears itself. The eventually filled cells from the first line would drop one level, leaving the first line empty again.

However we found a mistake in this proof. There are two gameboard configurations with an empty first row where it is not possible to complete the second row. Due to the instantaneous rotation model, if the only empty cell of the second row is the rightmost cell, the arriving horizontal domino, which has its center to the left, cannot be rotated to fill in the cell. If another cell of the row is also empty, it can be used to rotate the domino in a position where its center is on the right, allowing the filling of the rightmost cell. Yet this is not possible if the only other empty cell of the row is the leftmost cell. So in configurations where the only empty cells of the second line is only the rightmost cell or only the rightmost and the leftmost cells, the second row cannot be as easily cleared as described by Demaine et al.. In fact it cannot be cleared at all because if the rightmost cell can be *filled from below*[3], then the whole third line would have to be empty, and this is not a legal gameboard configuration.

Hence in addition to, as Demaine et al. summarized, proving that determining whether the top row can be cleared is in P and to handle the case when the top row cannot be cleared, we also need to find an efficient way to avoid the above problematic configuration and to handle the case where they are not avoidable. As this mistake was noticed belatedly, we don't handle this case in this document. Instead, we only present a procedure to determine in polynomial time whether the top row can be cleared.

We consider in the initial gameboard that the *initial state* is legal, ie. that the cells $(1, \lfloor w/2 \rfloor)$ and $(1, \lfloor w/2 \rfloor + 1)$ are empty, or the first arriving piece would be blocked and the game would end. In this case it is not possible to place any piece, so trivially it is impossible to clear the first line.

In the following, we simplify the description of a piece state introduced in Section 1. Instead of using a 4-tuple of the *piece type*, its *orientation*, the *position of its center* and the value *fixed* or *unfixed*, we will

[3]see Section 3.1.3 to understand how to fill in a cell from below.

reduce this definition to the couple *(position, orientation)*, as there is only one piece type possible and as we will only consider unfixed pieces.

General idea. Deciding if the first row can be cleared is equivalent to deciding if there is any row that can be cleared. Indeed by clearing a row, all rows above would drop by one level, leaving the first one completely empty. So the idea is to consider each row and to verify if it is possible to clear that row. As we will show later, fixing pieces into a given row i might require completing cells in lower rows. So it could happen that a lower row clears before the row i was filled. This would change the configurations around row i. To avoid this difficulty, we will look at each row from bottom to top (i.e. in a decreasing order of i). By doing so, when checking if a given row i can be cleared, we will know for sure that lower rows will not clear.

The first step of our algorithm is to identify all *reachable* cells in the board. This will allow us to filter rows where each empty cell can be reached from the origin. However, it doesn't suffice that an empty cell is reachable from the origin, it also needs to be possible to fix a piece into that cell. So we will look at each piece independently and determine a way to fix it, using *supports*. The empty cells will then be grouped into *holes* and it will be determined if it is possible to fix pieces in the hole all together. Finally, once it was verified that every hole can be completed independently, we'll determine if they can be completed concurrently.

3.1.1 Identify reachable rows

If it is possible to move a piece from the *initial state* to an *ending state*, then this *ending state* is said to be *reachable*. By extension, a cell is *reachable* if there exists a reachable piece state that covers it. A *reachable row* is a row for which every empty cell is reachable. To identify the later ones, we will first identify all reachable piece states in the given gameboard.

Identify reachable piece states Let's consider the directed graph of all possible moves. The vertices are 3-tuples (i, j, θ) where i and j represent the row and column of the piece's center and $\theta \in \{0, 90, 180, 270\}$ the orientation of the piece. There is an arc from (i, j, θ) to (i', j', θ') if there is a legal move from the first piece state to the second one in the given gameboard. A move can be a clockwise rotation, a counterclockwise rotation, a slide to the left, a slide to the right or a drop by one level. If the move is legal, this corresponds to an arc from (i, j, θ) to respectively $(i, j, \theta + 90°)$, $(i, j, \theta - 90°)$, $(i, j-1, \theta)$, $(i, j+1, \theta)$, $(i+1, j, \theta)$. We don't consider here the *fix* move because it would only add self-loops.

We run a *Breadth First Search* (BFS) algorithm to determine the shortest path from the *initial state* to any piece state. If no path was found, then the piece state is simply not reachable. This algorithm has a time complexity $O(w \times h)$, i.e. linear in the size of the board.

Identify reachable cells. The reachable cells are those for which there exists a reachable piece state that covers it. To cover a cell (i, j), there are 8 possible different piece states (see Figure 14). Hence to determine if a cell is reachable, it suffices to verify if at least one of the states is reachable by looking into the result table of the BFS algorithm. This is done in constant time if the table is pre-computed. So identifying all reachable cells of the board takes time $O(w \times h)$.

Identify reachable rows. We now simply parse the entire board row by row and memorize all rows whose empty cells are all reachable. Such a parsing will take another $O(h)$ time complexity, so $O(w \times h)$ in total.

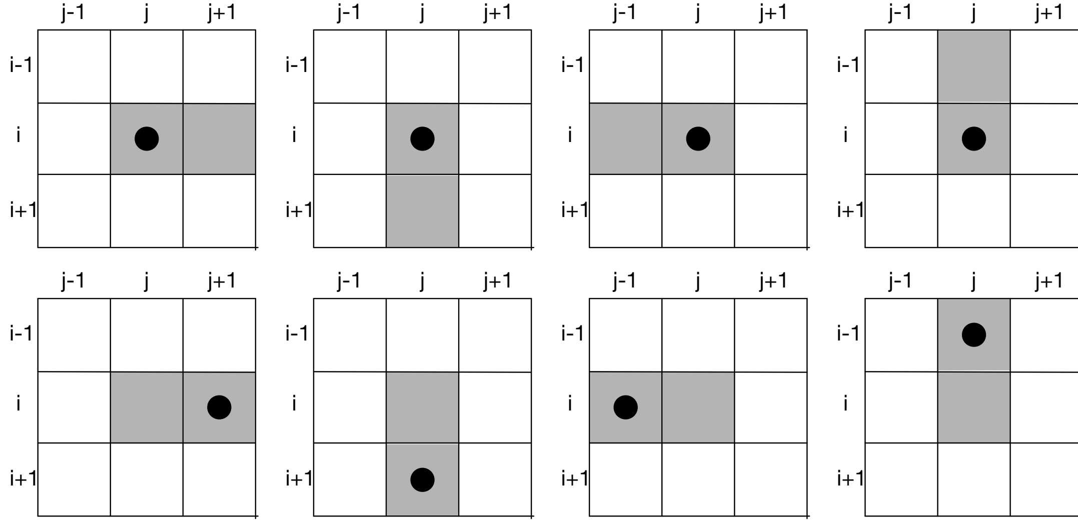


Figure 14: There a 8 different piece states that cover the cell (i, j). The four orientations where the piece center is on (i, j): $(i, j, 0)$, $(i, j, 90)$, $(i, j, 180)$ and $(i, j, 270)$; and the four ones where the piece center is not on (i, j): $(i, j+1, 180)$, $(i+1, j, 270)$, $(i, j-1, 0)$, $(i-1, j, 90)$.

3.1.2 Fixing a piece : use of a support

In the previous section we filtered the reachable rows, but we still need to verify if it is possible to fix the pieces in the empty cells. Indeed a piece can only be fixed if at least one gridsquare below is filled. Otherwise it may be possible to achieve it artificially by fixing pieces in lower rows. Those pieces form together a *support* for the piece we aim to fix. Before describing the possible supports, let us introduce the following vocabulary.

Hole. We define a *hole* as the maximal sequence of adjacent empty cells. They can be aligned horizontally or vertically. A vertical hole is characterized by its column j, the row i of its highest empty cell and its size s, the number of empty cells. Formally, a *vertical hole* is a tuple $(i, j, s)_V$ where:

1. the cells (t, j) for $t \in \{i, i+1, \ldots, i+s-1\}$ are empty ;
2. the cells $(i-1, j)$ and $(i+s, j)$ are filled.

Similarly, a horizontal hole is characterized by its row i, the column j of its leftmost empty cell and its size s, the number of empty cells. Formally, a *horizontal hole* is a tuple $(i, j, s)_H$ where:

1. the cells (i, t) for $t \in \{j, j+1, \ldots, j+s-1\}$ are empty ;
2. the cells $(i, j-1)$ and $(i, j+s)$ are filled.

Overlapping If there is a non empty vertical hole below a piece, a support is needed to fix it. There are many possible supports for a given piece state. However, since we need to be able to fix the pieces simultaneously, it will be important to use supports that don't overlap. Figure 15 gives an example of two support that overlap. To prevent this as much as possible, we choose to use supports that only fill in cells in the same columns as the piece we seek to fix. The support will use a cell from another column only if it is not possible otherwise.

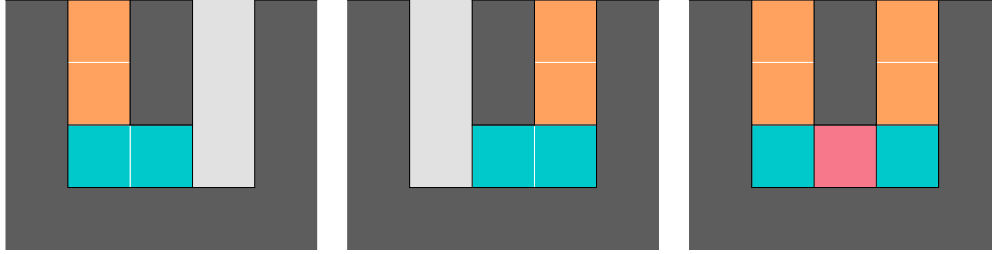


Figure 15: The orange pieces are the one we aim to fix. The used supports are depicted in blue. The two acceptable supports overstep into the same column and are not simultaneously legal : they overlap on the red cell.

Horizontal pieces are always fixable. Let's consider a piece state for which the piece is horizontal (i.e. $\theta \in \{0, 180\}$) and that covers the cells (i, j) and $(i, j+1)$. If both gridsquares below are empty, the piece would fall. Eventually, it will stop when reaching a filled gridsquare below it. We can repeat this operation until we fixed the horizontal piece on cells (i, j) and $(i, j+1)$. Hence horizontal piece states are always fixable. Notice also that such a *support* only fills in cells in columns j and $j+1$. We give an example of such a support in Figure 16.

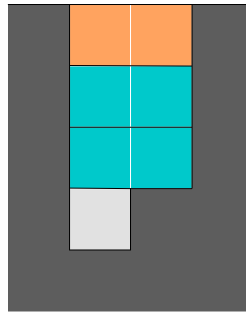


Figure 16: Example of a horizontal piece state (orange) that needs a support (blue).

Supports for vertical pieces. To fix a vertical piece (i.e. $\theta \in \{90, 270\}$) that covers the cells (i, j) and $(i+1, j)$, we can similarly let vertical pieces drop until the desired state is reached. However, since each vertical piece uses two rows, this is possible only if there is an even number of empty cells below $(i+1, j)$. Formally if the vertical hole $(i+2, j, s)_V$ has a size s even. If this is not the case, then we need to use at least a horizontal piece in the support. Let's consider the highest horizontal piece. Since it is the first, all above pieces stand vertically. This is only possible if the horizontal piece is in a row of same parity as i. So if the vertical hole $(i+2, j, s)_V$ has an odd size s, it should be possible to fix a horizontal piece that will cover some cell (i', j) where i' has same parity as i and $i+2 \leqslant i' \leqslant (i+2)+s-1$ (cell (i', j) belongs to the hole). Furthermore, since horizontal pieces are always fixable, it suffices to verify that the state is reachable by looking in the BFS table, which takes constant time. Finally, determining which support a vertical piece has is done in $O(h)$ time. Observe that such a support uses a horizontal piece that oversteps out of the column j to the left or to the right.

To differentiate the supports, we say that it is a **good support** if the hole $(i+2, j, s)_V$ has an even size. In this case, the support only uses cells of the columns j which makes it easier for the next steps. If the hole $(i+2, j, s)_V$ has an odd size, we call the support an **acceptable left** (respectively **right**) **support** if it uses a cell from the column $j-1$ (resp. $j+1$). Figure 17 illustrates the three different kinds of supports for vertical pieces.

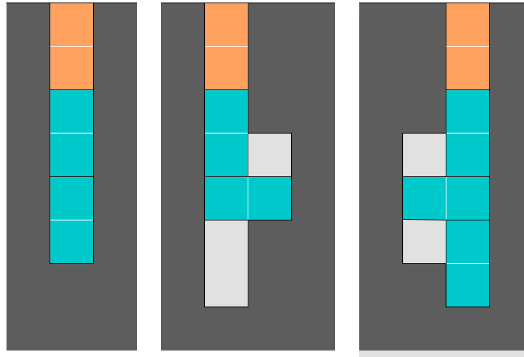


Figure 17: Example of (from left to right) a good support, an acceptable right support and an acceptable left support.

3.1.3 Completing a hole

The above section deals with fixing a piece independently of other pieces to be fixed. We now group the empty (fixable) cells into *horizontal holes* and determine if is possible to complete each of them independently of other holes. However, as previously, to avoid overlapping between two holes, we will prefer to complete a hole using only its columns. We differentiate therefore between **good holes**, that can be completed using only their columns, and **bad holes**, for which it is necessary to use another column. Recall that the above described supports cover in the worst case only the directly adjacent column. Hence if we use only those supports, an overlapping between two holes can occur if they are separated by a single column.

To characterize the good and bad holes, we need to find an explicit way to complete them that will ensure for a bad hole that there is no way to use only its columns. We proved earlier that horizontal pieces can always be fixed using only the same columns as the piece. Therefore we want to use as many horizontal pieces as possible. Furthermore, if it is possible to cover an empty cell of a given hole with a horizontal piece, then it is also the case for every empty cell of the hole : simply slide the piece to the left or right. However, it can happen that a hole cannot be reached with a horizontal piece. Obviously, if the hole has a size one, we need to use a vertical piece. Figure 18 shows another example where the hole can only be filled with vertical pieces.

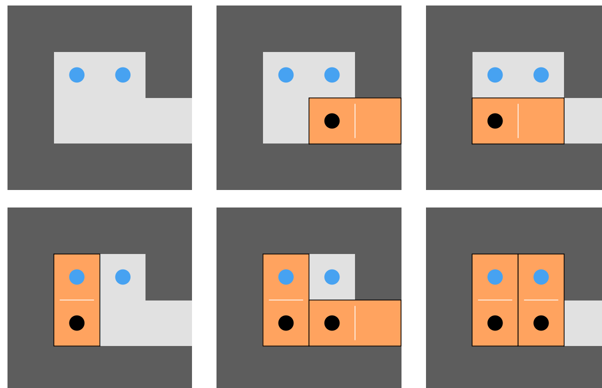


Figure 18: Example of a hole (blue marked cells) that cannot be reached with a horizontal pieces. The only possibility to complete the hole is by using vertical pieces that are arriving horizontally and are finally rotated upwards.

Possible shape of a hole. Up to a slight variation, the above example in Figure 18 is actually the only possible shape for a hole of size greater than 1 that cannot be reached with horizontal pieces. To prove this, let us look at all the configurations that can be used to complete two adjacent cells. Figure 19 shows those ten different possible configurations.

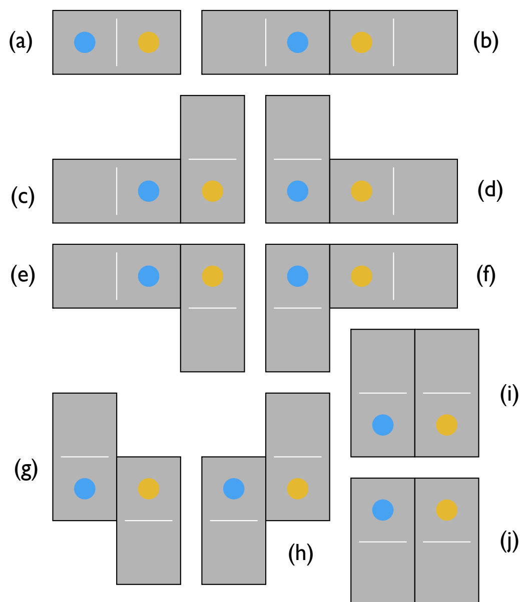


Figure 19: The ten different possibilities to cover two cells marked with a blue and a yellow dot.

Configurations (b), (c), (d), (e) and (f) use at least one horizontal piece. Thus by eventually sliding it to the left or right, both blue and yellow cells can be covered with a single horizontal piece as in configuration (a). It is also possible to reduce to configuration (a) from (g), (h), (i) or (j) if one of the piece's center is on the blue or yellow cell: simply rotate the piece.

As illustrated in Figure 20, in the configuration (i), even if both vertical pieces have their center above the marked cells, it is possible to reduce to configuration (a). In fact, if both vertical piece states are reachable, then the four cells are empty, leaving enough place to rotate a domino and let it drop to the marked cells.

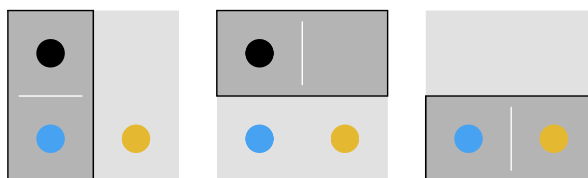


Figure 20: How to reduce configuration (i) to configuration (a) when both piece centers are not on the marked cells using only the four empty cells.

In the configurations (g) and (h), let's call the vertical piece that fills in the cell above the marked one an *up standing piece* and the one that fills in the cell below the marked one a *down standing piece*. Figure 21 shows that in this configurations, if the cell below the up standing piece is empty, then this latter one can drop by one level. Its center will consequently be on the marked cell. So the piece could rotate and cover both marked cells horizontally.

Hence the only configurations for which it is not possible to reduce to (a) – and so not reachable by horizontal pieces – are configurations (g), (h) and (j) with down standing pieces having their center on

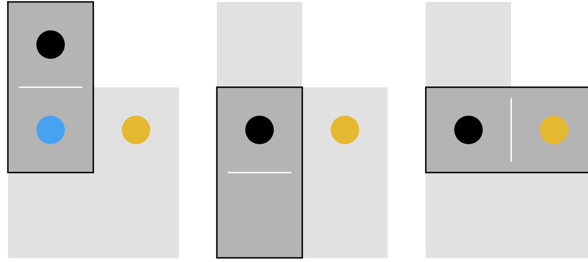


Figure 21: In the configuration (g), if the cell at the bottom left corner is empty, then it is possible to cover both blue and yellow cells with a single horizontal piece. Configuration (h) is the mirror of configuration (g).

the lower cell and up standing pieces having their center on the upper cell and the cell below them filled. In addition, in a hole that cannot be reached with horizontal pieces, there cannot be two adjacent up standing piece, or it will correspond to configuration (i). We observe furthermore that the only possibility for a piece to be down standing with its center on the lower cell is by coming from the lower row and ending with an upward rotation (as shown in Figure 18). Otherwise it would have gone through a state where its center is on the marked cell. Moreover, if we consider a hole where it is not possible to use a horizontal piece, there can only be at most one up standing piece. In fact if there were two, then the two filled cells below them would obstruct the access for down standing pieces between them (see Figure 22 for an illustration).

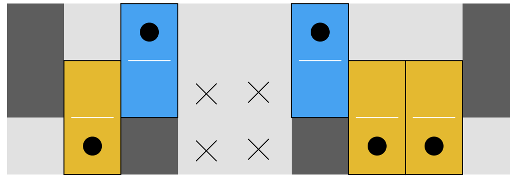


Figure 22: In a hole that can be reached only with vertical pieces, if there are two up standing pieces, then they have to be separated by at least one column (or this corresponds to configuration (i)). But if there are two upstanding pieces (blue) in a hole, then it is impossible to bring down standing pieces (yellow) in the cells between them (crossed cells). Indeed down standing pieces are coming from the lower row as illustrated in Figure 18.

To sum up, a hole can either be reached with horizontal pieces or it doesn't. In the second case, if it has a size one then the piece can come from the top or from the bottom. If it has a size 2 or more, then the only possibility to reach the empty cells is to use down standing pieces (as in 18) and at most an up standing piece. To determine if a hole can be reached with horizontal pieces or not, only verify if there is a reachable horizontal piece state that covers the first cell of the hole. This is done in constant time using the BFS table.

Let us now find an effective strategy to complete both kinds of holes.

Holes that can be reached with horizontal pieces. In a hole $(i, j, s)_H$ where horizontal pieces can be placed, if s is even, we can complete it using only horizontal pieces. But if s is odd, there should be an odd number of vertical pieces. We choose to use only one vertical piece. Using the same kind of argument as before with vertical holes, the vertical piece should stand in a column of same parity as j.

Since this vertical piece may need a support, we have to make sure that its support doesn't overlap with the support of its neighbor pieces.

This is by definition the case if it has a good support. Moreover, since it is the only vertical piece to fix in the hole, the neighbor pieces can only be horizontal. And horizontal pieces can always be fixed, regardless of the size of the vertical hole beneath them. So even if it has an acceptable support, it won't overlap with its neighbor's support. This applies both to up standing and down standing pieces in row i. Finally, the only case where it may not be possible to complete the hole in the **good** way, is if the only possibility to fix the vertical piece is by using an acceptable left support in column j or using an acceptable right support in columns $j + s - 1$.

Good hole. We characterize a hole $(i, j, s)_H$ that can be reached with horizontal pieces as **good** if either of the following stands:

1. s is even ;
2. there exists a $j' \in [\![j, j + s - 1]\!]$ of same parity as j such that either (i, j') or $(i - 1, j')$ has a good support ;
3. there exists a $j' \in [\![j + 1, j + s - 2]\!]$ of same parity as j such that either (i, j') or $(i - 1, j')$ has an acceptable right or left support ;
4. (i, j) or $(i - 1, j)$ has an acceptable *right* support ;
5. $(i, j + s - 1)$ or $(i - 1, j + s - 1)$ has an acceptable *left* support.

Bad hole. The hole is **bad** if it isn't good and either:

1. (i, j) or $(i - 1, j)$has an acceptable *left* support ;
2. $(i, j + s - 1)$ or $(i - 1, j + s - 1)$ has an acceptable *right* support.

Otherwise, the hole is impossible to complete, and so is the row. Finding the support for a vertical piece is done in $O(h)$ time. Thus, verifying if a hole is good or bad is done in $O(w \times h)$ time.

Holes that can only be reached with vertical pieces. In a hole that only accepts vertical pieces, we need as before to find a way to arrange the supports of each piece so that they don't overlap.

Share a support. An interesting case occurs when two adjacent vertical pieces in rows i and $i + 1$ and columns j and $j + 1$ both need a support (good or acceptable). In fact, if a horizontal piece could be fixed right below them into the cells $(i + 2, j')$, $(i + 2, j' + 1)$, this piece would serve as a common support and allow the player to fix both vertical pieces. This is actually always possible as illustrated in Figure 23. Since the two vertical piece states are reachable, the four cells (i, j), $(i, j + 1)$, $(i + 1, j)$ and $(i + 1, j + 1)$ are initially empty. We can therefore bring a vertical piece to the cells (i, j), $(i + 1, j)$ and rotate it to a horizontal position. Depending if the piece's center is in row i or $i + 1$, the rotated piece is in row i or $i + 1$. In both cases it can be dropped to the row $i + 2$. Thus the horizontal piece state is reachable. And as we already know, it is also always fixable. So it is always possible to fix two adjacent vertical pieces, as long as both gridsquares below them are empty.

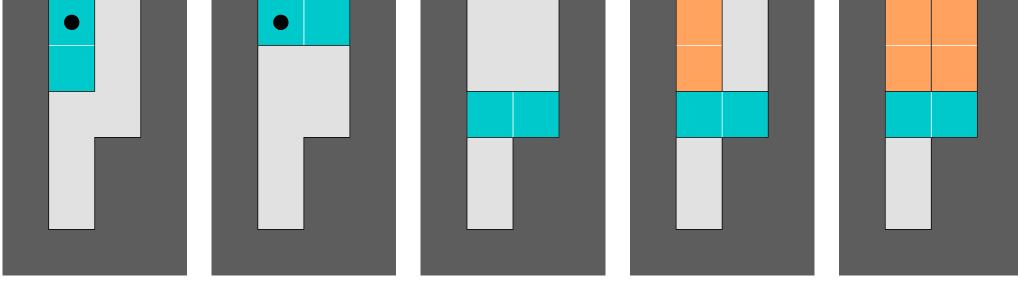


Figure 23: Shared support (blue) for a couple of vertical pieces (orange).

Grouping in subholes. Let's group together the columns for which a support has to be build to fix a vertical piece. We define a *subhole* as the maximum sequence of such columns and we note $(i, j, s)_{SH}$. Recall that since the only eventual up standing piece doesn't need a support, every piece to be fixed in a subhole is down standing. If the subhole is even-sized, we can group the columns two-by-two so that each couple of columns share a support (as illustrated in Figure 23). In this case, there is no overlapping. But if the subhole is odd-sized, there will necessarily be a single column j' left that cannot share its support. j' has the same parity as the first column of the subhole. There is trivially no overlapping if its support is a good one. However it may happen if it is an acceptable support.

So let's consider the cell (i, j') with an acceptable support that cannot share it. If the acceptable support is left, we call *neighbor* the vertical piece that has to be fixed in cell $(i, j' - 1)$. If the support is acceptable right, the *neighbor* is the vertical piece to be fixed in $(i, j' + 1)$ If the neighbor doesn't need a support, then its fixing is trivially unaffected. If the neighbor needs a support then it was paired with another vertical piece to share it (as the piece in (i, j') is the only one to not be paired). Since the common support consists of an horizontal piece, the overstepping cell of (i, j')'s support can be anywhere below the common support.

Hence the only case where it is not possible to fix the piece in (i, j') is when the unique possibility to place the horizontal piece of its acceptable support is in the cell right below the neighbor, so in row $i + 2$. The horizontal piece would in this case overlap with the common support.

Good subhole. A hole that can only be reached with vertical pieces is a **good hole** if every of its subhole are **good subholes**. As before, we need to pay attention to acceptable supports on edges of the hole or subhole. A subhole $(i, j, s)_{SH}$ is **good** if either of the following stands:

1. s is even ;
2. there exists a $j' \in [\![j, j + s - 1]\!]$ of same parity as j such that (i, j') has a good support ;
3. there exists a $j' \in [\![j + 1, j + s - 2]\!]$ of same parity as j such that (i, j') has an acceptable (right or left) support that oversteps in a row strictly below $i + 2$;
4. (i, j) has an acceptable *right* support that oversteps in a row strictly below $i + 2$;
5. $(i, j + s - 1)$ has an acceptable *left* support that oversteps in a row strictly below $i + 2$.

Bad subhole. The hole is **bad** if at least one of its subhole is **bad**. A **bad subhole** $(i, j, s)_{SH}$ is not a good one and either of the following stands:

1. (i, j) has an acceptable *left* support ;
2. $(i, j + s - 1)$ has an acceptable *right* support.

Otherwise, the subhole is impossible to complete and so is the row. Similarly to holes that accepts horizontal pieces, verifying if a hole unreachable by horizontal pieces is good or bad is done in time $O(w \times h)$.

3.1.4 Clear a row

In the following we will consider holes and subholes. To simplify the notation, for now on we will use the term *hole* also for *subhole*.

The final step of the algorithm is to verify that all the holes of row i can be concurrently completed. An overlapping can actually only happen between two bad holes separated with a single column. Indeed, good holes by definition don't overstep into columns of other holes. This is also the case for bad holes since they only use the directly adjacent column. Thus the overlapping can only happen if the left hole uses an acceptable right support and the right hole an acceptable left support, with both supports overstepping into the same column. The next paragraph describes a strategy to deal with bad holes in order to avoid such overlapping.

Strategy. If every bad hole uses an acceptable left support, there would be no right support to overlap with. Therefore the following strategy gives priority to left supports and ensures that right supports are used only if it is not possible to do otherwise to complete row i. Such a strategy correctly concludes if it is possible or not to concurrently complete the bad holes of a row.

Consider the bad holes $(i, j, s)_H$ from left to right (e.g. in increasing order of j). Do Step 1 then 2. If both Step 1 and 2 fail, then it is not possible to avoid an overlapping and the row cannot be completed.

Step 1. If (i, j) has an acceptable left support, try to use it. This means that if the column $j - 1$ is not marked, use any possible acceptable left support. If it is marked, verify for all potentially marked cells memorized in Step 2 that there is a possibility to use another cell for (i, j)'s support. This step takes $O(h^2)$ time complexity.

Step 2. If it is not possible to use an acceptable left support for the cell (i, j) (because there is none or because Step 1 failed) and if $(i, j + s - 1)$ has an acceptable right support, memorize all cells in column $j + s$ that can alternatively be filled with the overstepping cell. Mark the column $j + s$. This step takes $O(h)$ time.

Order in which to place pieces. At this point, we found a way to arrange pieces that will complete row i without overlapping each other. But is it possible to fix pieces so that the other pieces we need to fix remain reachable? Here comes the BFS table useful. It contains the length of the shortest path from the origin. Therefore if we place further pieces first, pieces with a shorter path will remain reachable. Additionally it seems obvious that the support should be build before fixing the pieces that will actually clear row i. The first support to be placed should be acceptable ones because they overstep into other columns. Once this is done, common support, good support, and support for horizontal pieces can be build, before finally placing the pieces that will clear the row.

Total complexity of the algorithm. To sum up, the algorithm has to run a Breadth First Search Algorithm on the graph of possible piece states. Notice that the graph is encoded implicitly. Hence the BFS algorithm takes $O(w \times h)$ time. For each row, it takes $O(w)$ time to decide if it is reachable, constant time to group empty cells in holes and $O(w \times h)$ time to decide if the holes are good or bad. The final strategy takes another $O(w \times h^2)$. Hence the complexity of the algorithm is $O(w \times h^3)$ as the strategy applies on every (reachable) line. This proves that determine whether it is possible to empty the first row is in P.

3.2 Clearing with rotation

The previous algorithm to clear a row can also be used to clear the entire board. The problem at hand is the following.

> Domino Clearing with rotation
> **Input:** A *2-tris* gameboard of size $(h \times w)$ partially filled with blocks and m *2-tris* pieces.
> **Question:** Is it possible to play in such a way that the gameboard is left empty in the end ? Rotations are allowed.

My supervisor Michail Lampis came up with the following algorithm. Consider the first two rows to be empty, so that the special case raised at the beginning of 3.1 doesn't occur. For this, the strategy of 3.1 will probably be needed. Then use the following strategy to clear the topmost non-empty row.

Clear row $i+1$. Suppose the first i rows to be empty. While there is a $j \in [\![1, w]\!]$ such that the cells $(i+1, j)$ and $(i+2, j)$ are empty, rotate a domino vertically and let it drop into column j. Since $i \geqslant 2$, it is always possible to rotate the piece and to move it to the column j. This leaves all rows $i' \leqslant i$ empty. By doing this as long as possible, either a row $i' > i$ completes and clears, or every empty cell of row $i+1$ has a filled cell below it (or we would have continue to place a vertical piece in this column). In the first case, we achieved to clear row $i+1$ and reached a state where the first $i+1$ rows of the gameboard are empty.

In the second case, try to clear row $i+2$. We observe that every empty cell of row $i+2$ has a filled cell above it. In order to reach those empty cells, we want to empty the cells above. To do this, we continue to place vertical dominoes in every empty cell of row $i+1$. This will temporarily fill some gridsquares in row i, but once all dominoes are placed, row $i+1$ clears and all filled cells in row i drop by one level, leaving once again the first i rows empty. This way, the state of every cell of row $i+1$ is changed. Indeed, every empty cell of row $i+1$ becomes filled by the upper gridsquare of the vertical domino. On the other hand, we didn't placed any piece into columns j where $(i+1, j)$ was filled. So this cell becomes empty as the row clears and as no filled gridsquare above drops. Hence for every empty cell $(i+2, j)$ where the above cell $(i+1, j)$ was filled, this later one becomes empty. This allows us to repeatedly place vertical pieces in those columns without filling any cell of row i. All cells $(i+2, j)$ will become filled (either by the upper or lower gridsquare of a domino) and row $i+2$ will eventually clear, leaving row $i+1$ empty.

Hence in both cases, we managed to reach a game state where the first $i+1$ rows are empty, starting from a board where the first i rows were empty. By repeating this operation, the entire board will eventually be cleared. Once the board is completely empty, the pieces can be easily and neatly stacked into the one or two bottom rows that successively clear.

This algorithm proves that it is possible to clear the board with at most $O(w \times h)$ pieces (use $O(w \times h)$ pieces for the first step and $O(w \times h)$ pieces for the second step). And for each piece, the decision is constant so this algorithm has time complexity $O(w \times h)$. However this doesn't prove that Domino Clearing with rotation is in P. To do this, we would need to determine for a given m if it is possible to clear the board. This problem seems harder than to determine the minimum number of pieces required to clear the board. This last problem seems a little difficult to solve – at least at my level.

3.3 Surviving without rotation

In [4, Section 4.1], Demaine et al. proved that the Domino Clearing without rotation problem is NP-complete by doing a reduction from 3-Partition. The reduction is similar to the one made for Tetris Clearing in 2.1. We can extend this proof in order to show that Domino Surviving without rotation is also NP-complete.

Proof. Call R the sequence of pieces used in the reduction. The idea is to append a sequence of pieces to R that can be placed without getting stuck if and only if the entire board was first emptied with R. Consider an empty board of size $(h \times w)$, with odd w. The maximum number of pieces that can be placed in an empty row is $\lfloor w/2 \rfloor$, yet the row won't clear as there will always be an empty cell since w is odd. So if we add $h \times \lfloor w/2 \rfloor$ horizontal dominoes to R and if the board width w is odd, the only way of surviving is to first empty the board with the pieces in R, i.e. solving the 3-PARTITION problem.

In the original reduction made by Demaine et al., the board has width $5n$ and height $3n + 2B/n$. If $5n$ is already odd, we simply need to append the $h \times \lfloor w/2 \rfloor$ to the piece sequence R. However if it is even, we need to slightly modify the 3-PARTITION instance to have an odd width board. To the sequence $\{x_1, x_2, \ldots, x_{3n}\}$ add $a = 1$, $b = 1$ and $c = B/n - 2$. As $c = B/n - 2$ is really big compared to B/n, it necessarily has to be packed together with a and b. Once this first partition is done, consider the remaining $\{x_1, x_2, \ldots, x_{3n}\}$ that all verify the property to be between $B/4n$ and $B/2n$. So this changes nothing to the problem. However if the old instance had $3n$ integers with $5n$ even (that is n is even), the new instance has $3n + 3 = 3n'$ integers with $n' = n + 1$. So n' is odd and so is $5n'$. Hence the board width of the reduction is odd and we boil down to the previous case. □

Conclusion

This report completes the work on the hardness of *Tetris*. It proposes another way to do the reduction from 3-PARTITION and approximate an optimal solution for the without inner hole case. It also strengthens the intuition that DOMINO SURVIVING WITH ROTATION is in P through a polynomial algorithm deciding whether a row can be completed. We further propose a polynomial algorithm that tries to solve the DOMINO CLEARING WITH ROTATION problem, even if it is only an approximation of the optimal algorithm. This consolidates the idea that this problem also is in P. Conversely, we proved that the DOMINO SURVIVING *without* ROTATION is NP-complete.

Further work should be done to complete the proof that DOMINO SURVIVING WITH ROTATION is in P, especially regarding the special case where the second row is not completable even if the first one is empty. The case of DOMINO CLEARING WITH ROTATION also needs some further reflexion in order to find an optimal algorithm.

As part of a research internship, I am particularly pleased to have been able to manipulate mathematical tools of theoretical computer science. This is a field of study that I discovered during the internship through the reading of many proofs, especially NP-completeness reductions, and that I find very interesting. It was also an opportunity to learn more about algorithmic techniques, such as dynamic programming, which will be very useful since I'm going to pursue a master degree in computer science.

This work was also an insight into the dynamics of a research work. I worked about problems for a long time, thinking one day I had find a lead and then get back the next day because it was incorrect. I spent a considerable amount of time proving that DOMINO SURVIVING WITHOUT ROTATION is in P before trying to prove that it was NP-complete (which is quite simple by the way). I think that the format of this internship was very productive. We met with my supervisor Michail Lampis on a regular basis, about every two weeks. This gave me enough time to have some result to discuss without being counterproductive.

Bibliography

[1] R. Breukelaar, H. J. Hoogeboom, and W. A. Kosters, “Tetris is hard, made easy,” *Leiden Institute of Advanced Computer Science, Universiteit Leiden*, 2003.

[2] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani, *Algorithms.* McGraw-Hill Higher Education New York, 2008.

[3] E. D. Demaine, S. Hohenberger, and D. Liben-Nowell, “Tetris is hard, even to approximate,” in *International Computing and Combinatorics Conference.* Springer, 2003, pp. 351–363.

[4] E. D. Demaine, M. L. Demaine, S. Eisenstat, A. Hesterberg, A. Lincoln, J. Lynch, and Y. W. Yu, “Total tetris: Tetris with monominoes, dominoes, trominoes, pentominoes,...” *Journal of Information Processing*, vol. 25, pp. 515–527, 2017.

[5] M. R. Garey and D. S. Johnson, *Computers and intractability.* freeman San Francisco, 1979, vol. 174.

[6] H. J. Hoogeboom and W. A. Kosters, “How to construct tetris configurations.” *Int. J. Intell. Games & Simulation*, vol. 3, no. 2, pp. 97–105, 2004.

[7] ——, “Tetris and decidability,” *Information Processing Letters*, vol. 89, no. 6, pp. 267–272, 2004.

[8] ——, “The theory of tetris,” *Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica*, vol. 9, pp. 14–21, 2005.

[9] K. Jansen, S. Kratsch, D. Marx, and I. Schlotter, “Bin packing with fixed number of bins revisited,” *Journal of Computer and System Sciences*, vol. 79, no. 1, pp. 39–49, 2013.