Artificial Intelligence (H) : COMPSCI 4004

Coursework - Mondrian Tiling Problem

5th December 2022

GRANIÉ Sébastien

This document is a report about the Mondrian Tiling Problem, part of the Artificial Intelligence (H) course. In this document, you will find the answers to the coursework questions and general explanations of the method used to complete the coursework.

University of Glasgow - School year 2022/2023

Table of contents

1 Introduction

First of all, this report states as written explanation about the thinking and research phases done for this project related to the Mondrian Tiling Problem. This document contains answers to the coursework's questions and further detailed explanations.

Let us remind what the Mondrian Tiling Problem consists in. It deals with a squared puzzle that seeks to be filled by a set of integer-sided rectangles so that the whole area is fully covered in the end. However, some rules must be respected to fulfill this challenging task :

- All used geometrical form must be integer-sided rectangles ;
- None of these rectangles must have heights and widths that match with any other tile used ;
- The area must be fully recovered without any overlap of any rectangles.

It is important to recall that the Mondrian Tiling Problem is considered as an Optimisation Problem and more precisely as NP-complete because there is no well-known algorithm solving the problem and any answer build with a layout of rectangles can be easily verified to know whether it is a correct answer. In this report, we are going to detail advanced reasoning methods to avoid solving this problem using the Brute Force method.

Since it is an optimisation problem, the quantity we would like to compute and minimize as much as possible is the Mondrian score. We build it by subtracting the area of the smallest rectangle to the largest rectangle's area.

Here is an example :

Taking this 5 x 5 grid puzzle as example, the Mondrian score is computed by the difference between its largest and smallest area, which result in :

Mondrian score = 10 - 6 = 4

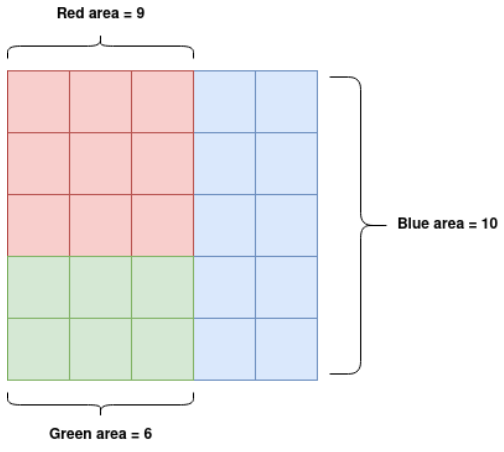

FIGURE 1 – Example of a tiled puzzle

To compile and execute the code, a ReadMe.md file is available in the given archive and explains how to process

2 Your tasks

After defining the context in the introduction, we will now begin to analyze the puzzle and understand how it works in practice.

2.1 Defining the states and actions

2.1.1 Definition of a state

Building a state space to solve Artificial Intelligence oriented problems is a common best practice and it helps to better understand the context, the challenges and the actions the algorithm can and need to perform to find an optimised solution. A state space is composed of an initial state, intermediates states and a goal state. Actions need to be performed to move from one state to another. We can perceive a state space as a graph, with nodes which are states in our case, and edges which are actions we will be explaining in the section 2.1.3.

In this context, a state would be a tiled square and the actions are the steps thats help us to move from one state to another, which main goal is to find a state closer to the desired solution. Recall, the desired solution is obtained by minimizing as much as possible the Mondrian score.

Since a state is defined by a tiled square and we have been assuming in the introduction that all tiled square must respect several conditions such as :

- All used geometrical form must be integer-sided rectangles ;
- None of these rectangles must have heights and widths that match with any other tile placed ;
- The area must be fully recovered without any overlap of any rectangles.

It means that our initial state must be a tiled square, to which we will be applying actions to optimise its Mondrian score, i.e. minimize as much as possible its Mondrian score.

2.1.2 Building our initial state

Before going into the details of the possible actions, we need to build our tiled square/puzzle to start the process at our initial state of the state space.

The main question now is : how do we create a tiled square ? Which method or process should we apply to an empty square to make it become an initial state ?

Let us start from an empty tiled puzzle of size 9 x 9 as drawn below.

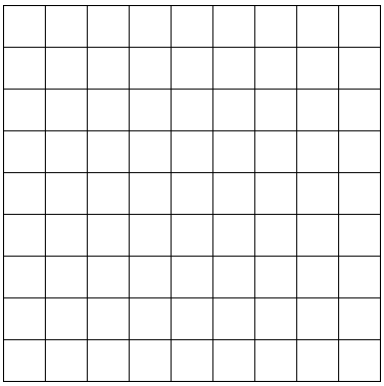

FIGURE 2 – An empty 9x9 grid

We will fill it with rectangles as the process progresses. We can decide to start from any side of the square. Let us start from the top left-hand corner and progressively add new rectangles to finally reach the bottom right-hand corner in the end.

The first rectangle shape will be defined randomly according to the 9x9 size of the choosen puzzle. The lower bound being 1 and the upper bound being 9. Let us consider, the randomness generated a 3x4 rectangle. We place it as shown below in the top left-hand corner.

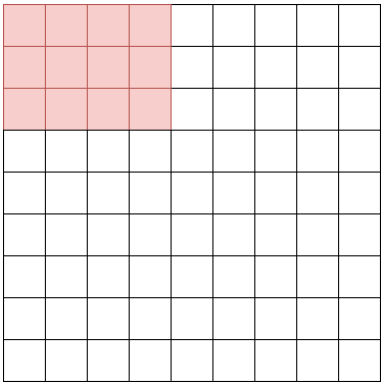

FIGURE 3 – First line fillment : step 1

Besides, we define a threshold of fillment that, once reached, we will fix the width of the generated rectangle's shape and only the height will be randomly generated. Let us say the threshold's value has been assigned to be equal to 2 for this grid's size. As long as this threshold has not been reached, we carry on generating rectangles to fill in the first line. The second rectangle shapes are 3x5. Here is below the new puzzle's representation :

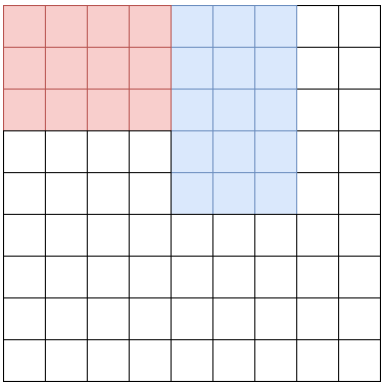

FIGURE 4 – First line fillment : step 2

We have now reached the threshold. The width of the next rectangle is set to 2 and we generate randomly its length. The new rectangle's shape equals 2x5.

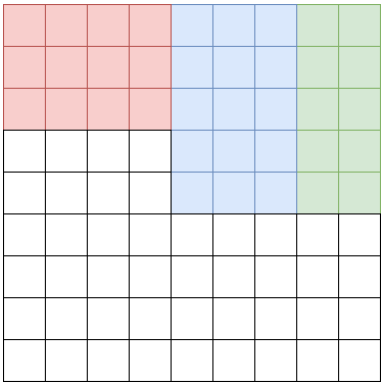

FIGURE 5 – First line fillment : step 3

We have filled in the first row completely. We must now carry on until the whole tiled puzzle is fully covered. To complete this task, we identify the location that is the furthest to the bottom line and it corresponds to the location we will fill next. In that case could start by adding a rectangle under the red one to equalize the depth between all rectangles, which will result in adding a $2 \times 4 = 8$ area rectangle. However, for the moment, we do not try to optimize anything, we are only trying to find an easy method to generate a valid initial state for our state space. Let us fill the remaining gap in the following way shown below by adding two rectangles of shape 4x6 and 5x4.

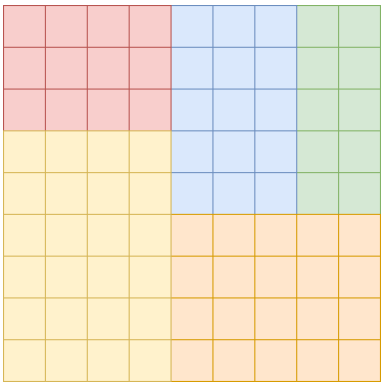

FIGURE 6 – Fully covered puzzle : initial state

Finally, our 9x9 tiled puzzle if entirely covered involving tiles of size : 3x4, 5x3, 5x2, 6x4, 4x5. We see that none of these combinations are pairwise congruent to another and thanks to the puzzle representation, we notice that the layout of all the rectangles fits well in the grid. Before computing the Mondrian score, I prefer to say that this method was my first idea of how to code the initial state. However, while coding I figured out that it was too much sophisticated only for an initial state that should in fact be quite easy to generate. Once we have our simple initial state, we can begin applying some actions to find more complex configurations and fitting of rectangles. I will be explaining the method I have chosen later on in the report in the section 2.4. which is quite similar anyway. We now have a valid initial state with a Mondrian score equals to :

Mondrian score = 24 - 10 = 14

It could have been worse for a first try. However, we have techniques to reduce even more the Mondrian score thanks to several actions.

2.1.3 Possible actions

Here are listed and explained in detail the possible actions we can apply to any states of the state space :

- Split action ;
- Merge action.

The **Spliting** action consists of dividing the largest rectangle into two smaller area. We need to be careful while processing every action as dividing or merging can result in considering a rectangle, for which its dimensions are already used in the current tiled puzzle. Using a rectangle size that is congruent with a rectangle already taken would result in an invalid state. As a consequence, we need to pay attention during the splitting action to choose new dimensions for both

of the two new created rectangles. This splitting action will definitely reduce the Mondrian score because the largest tile area will be reduced afterwards.

If we come back to our previous example, we have improvements to perform in order to reduce the Mondrian score. Let us perform a splitting action and see the effect on the Mondrian score.

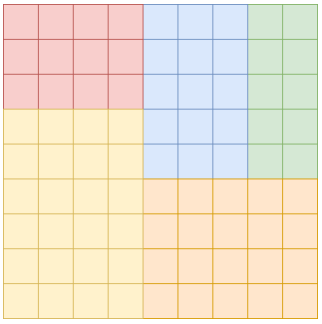

FIGURE 7 – Initial State

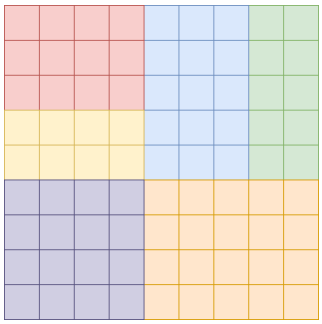

FIGURE 8 – State after performing Split Action

The transition splitted the yellow tile into a smaller yellow and a new purple one. We started from a Mondrian score of 14 to :

Mondrian score = 20 - 8 = 12 < 14

The **Merging** action consists of merging the smallest rectangle area with one of its neighbours, forming a new bigger rectangle area. Again, it is mandatory to pay attention to the Mondrian Tiling Problem rules and avoid congruent shape of rectangles within a puzzle. This merging action will definitely reduce the Mondrian score because the smallest tile area would increase afterwards.

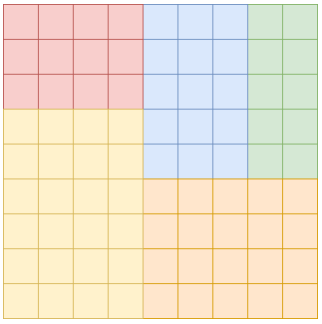

FIGURE 9 – Initial State

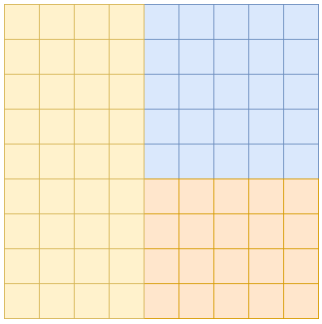

FIGURE 10 – State after performing a (double) Merge Action

The transition merged on one side the blue and green tiles into a bigger blue one. As it was not enough to reduce our example's Mondrian score, we decided to merge the yellow and red tiles into a bigger yellow one. We started from a Mondrian score of 14 to :

Mondrian score = 36 - 25 = 11 < 14

Once we understand how these two simple actions work, we can think of combining them into more complex ones :

— Split-Merge action ;
— Merge-Split action.

The **Split-Merge** action result in doing a splitting action followed by a merging action. This complex action involve more than an initial tile because once you divide the largest area into two parts, you look for the neighbours of the new created ones are try to merge with one of its neighbours.

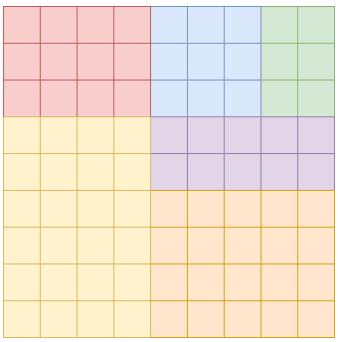

FIGURE 11 – Initial

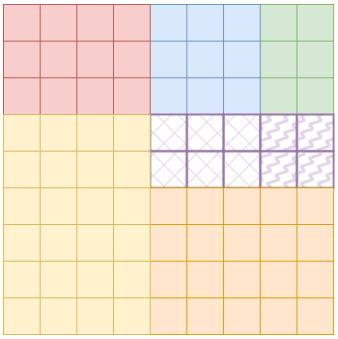

FIGURE 12 – intermediate

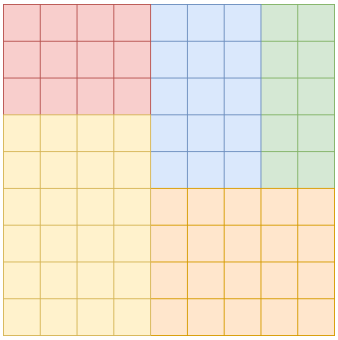

FIGURE 13 – final

We start by splitting the purple rectangle into to seperate parts : one 2x3 shape on the left and the second part with a 2x2 shape. Afterwards, we perform the (double) merge respectively with the blue and green rectangles. We started from a Mondrian score of 18 to :

Mondrian score = 14 < 18

The **Merge-Split** action keeps the same structure as the previous one. However, we will firstly perform the merging action, followed by the splitting one.

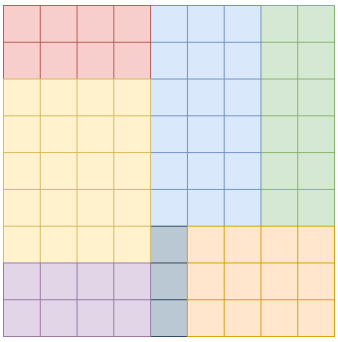

FIGURE 14 – Initial

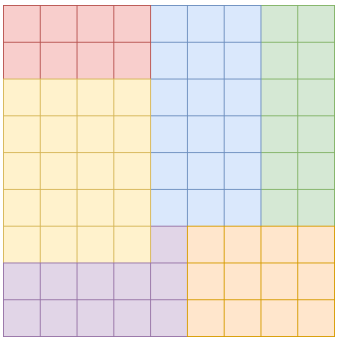

FIGURE 15 – intermediate

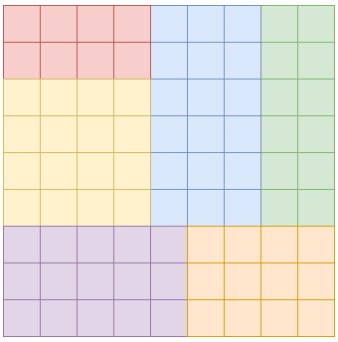

FIGURE 16 – final

This time, we start by merging the purple and grey rectangles. Afterwards, we perform the merge with the bottom part of the yellow rectangle. We started from a Mondrian score of 17 to :

Mondrian score = 10 < 17

It is mandatory that none of the new created rectangle's shapes are congruent to any of the shapes already existing in the grid. Otherwise, the performed action leads to an invalid state. In addition, these actions are not likely to produce it, but it is worth mentioning that no overlap is allowed and all definitive forms must be rectangles. If any of these conditions are not respected, the outcome of the action result in an invalid state.

2.1.4 Conclusion about states and actions configuration

Now that we have defined the states and actions of our desired state space. Let us recap the current state of our researches. We have an initial state that is a tiled puzzle filled with random rectangles and respecting the Mondrian Tiling Problem rules. In addition, we have defined as hinted in the coursework, four possible actions we can apply on every state on the state space. With these information, we can build a quaternary tree, starting from an initial generated grid and applying all four actions recursively to all the new generated states. The number of nodes (states) of such a tree is 4^n, with **n** being the number of times we apply the all four actions on all previous generated states, which represent a large number of possible states even after a few steps.

2.2 Actions leading to invalid states

First of all, let us define what an invalid state is. It is one state where either the tiles are overlapping (which means that it is not a valid tiling), or that at least a pair of rectangles is congruent. I take for granted that all shapes within a grid are integer-sided rectangles.

At the very beginning of the implementation, I tried to be able to generate too sophisticated initial states. After a while, I figured out generating more easy initial states is actually smarter because the algorithm afterwards plays the role of looking for a more optimal answer. What actually does the code by now is, to generate a more simple fitting of rectangles and then we are able to perform all the four actions and build our state space. Unfortunately, I did not have time to make the tree recursively generating states but the strategy I would have done is to accept invalid states and stop searching in a particular branch of the state space after generating two invalid states in a row. To my mind, we are unlikely to get a valid state after having two invalid states in a row. According to the simulations I may have done on my own, I aggreed that considering a few

invalid states can lead to a lower Mondrian score after a few actions performed. I do not reject the idea of accepting intermediate invalid states. However, It would have been interesting to evaluate the differences of performance between a model where we do not accept any invalid states on the one hand, and on the other hand, consider maximum two invalid states in a row before rejecting the results and compare both results. In all cases, we can agree that the more states you consider, the more costly the algorithm is and that is what we wish to avoid.

To support my argument, here is a step-by-step sequence of actions using intermediate invalid states and finding an optimal solution for a 5 x 5 square :

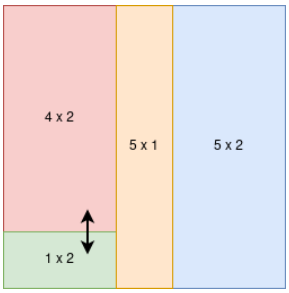

FIGURE 17 – Merge

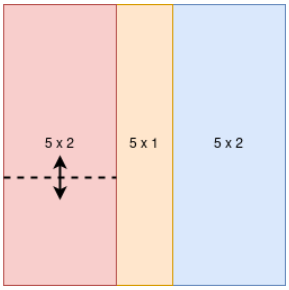

FIGURE 18 – Split

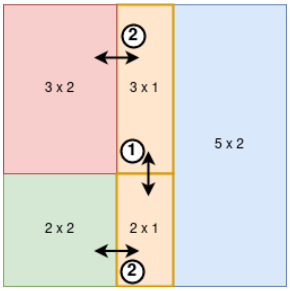

FIGURE 19 – Split-Merge

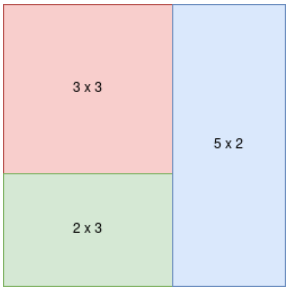

FIGURE 20 – final

Starting from an initial valid state, we perform a merge action, done on the smallest rectangle area. This merge action lead to an invalid state as there are two congruent shapes, the red and blue ones, of dimensions : 5 x 2. We decide to keep it and continue to perform actions. Then, we split one of the largest areas, which is either the blue or the red one. Finally, we perform a Split-Merge action that lead us to the minimal Mondrian score known for a 5 x 5 square, with a Mondrian score equals to 4. We can observe with that example that taking into account an invalid state in the state space can be an effective idea.

Algorithm 1: An algorithm that determines if the new state is a valid one or not given a state and an action.

```
Data: A state named s
Result: boolean
i ← 0;
globalArea ← 0;
for Rect ∈ rectList do
    i ← i + 1;
    /* Check for the integer-sided rectangles */
    if (!Rect.getLength().isInteger()) or (!Rect.getWidth().isInteger()) then
        return false;
    end
    for R ∈ rectList[i :] do
        /* Check for congruency */
        if condition congruency not verified then
            return false;
        end
    end
    /* Check for the coverage of the whole area */
    globalArea ← globalArea + Rectangle.area();
end
if globalArea ≠ globalArea ∗ globalArea then
    return false;
end
return true;
```

2.3 Compute Mondrian Score

Algorithm 2: An algorithm that computes the Mondrian score of a given state.

```
Data: A state named s
Result: integer
largest ← 0;
smallest ← s.getSize() × s.getSize();
for Rectangle ∈ rectList do
    /* Check for updating largest area */
    if Rectangle.getArea() > largest then
        largest = Rectangle.getArea();
    end
    /* Check for updating smallest area */
    if Rectangle.getArea() < smallest then
        smallest = Rectangle.getArea();
    end

end
return (largest - smallest);
```

2.4 Algorithm techniques

Once this tree is build over a few levels, we need to keep in mind that we are studying an optimisation problem. When it deals with optimisation and trees, there are many possible algorithms that are useable to reduce the computations and time complexity to reach our desired goal. We learned these algorithms in the lectures and in the Algorithm I (H) course such as : Breadth-First Search, Depth-First Search, Dijkstra algorithm, A* (A-Star) algorithm.

Although these algorithms have similarities in the solution they provide, they all have different purposes and each one is more efficient to perform specific tasks. In our case, we consider an initial state and a goal state within a quaternary tree. Starting from the initial state, we try to reach the goal state with the least possible computations and steps. We assume that the final state status is known and we know where we are heading to, at least in which direction. Direction in this context mean, that we have computations that can guide us to find a more optimal result, I am speaking here about the Mondrian score. In graph theory and before applying any of these algorithms, we need to know whether our graph is either directed or weighted or both. This features will have an impact on our decision.

2.4.1 Which Algorithm to apply ?

In our case, we study an undirected graph, as it is a quaternary tree and we consider the weights on the edges to be equal to one. We assume the cost of all previous actions (Split, Merge, SplitMerge, MergeSplit) explained above to be equal to one. As a consequence, we understand that the number of actions performed to reach a state is equal to its depth in the graph, starting from the initial state.

Since we know this fundamental information, we can decide which algorithm to apply to our graph. Breadth-First Search could have been a good idea as it ensures to find the optimal solution at the end of its computations. However, as mentioned earlier, the computations may be too complex to consider all possible states and we might need a more optimised algorithm to reach the goal state. As far as Dijkstra algorithm is concerned, we study a situation where all the weights are one, i.e. equivalent to considering an unweighted graph, in that case, dikstra algorithm is not useful for us. Let us now consider the A-star algorithm. It is known as the searching algorithm to find the shortest path between an initial state and a goal state. It works involving two information : the cost of each edges of the graph and an heuristic function.

2.4.2 A-Star algorithm

As explained before, the cost over every edges can be considered equal to one and we keep it so. An heuristic function is based on overall information about a situation and the main purpose is to guide an algorithm to reach its goal state faster than using any random heuristic function. It can be seen as additional information that helps and guides the algorithm to the goal state. In our case studying the Mondrian Tiling Problem, our heuristic function can be defined as the Mondrian score. As a consequence, the heuristic function applied to a specific state return the Mondrian score of this specific state. Knowing we try to minimize the Mondrian score, the A-Star algorithm will be guided by the sum of the depth of the graph, which equals to the number of actions performed to reach a state, and the heuristic score of a state, which equals to its Mondrian score.

2.4.3 Detailed explanation of the A-Star algorithm

As asked in the coursework, I will not write the A-star algorithm's pseudo-code in the report, however I will explain it in details so that my solution is easily understandeable.

The **inputs** are : firstly the dimension of the square : ***n***, which comes at the very beginning of the program while calling the main function, and secondly the limiting conditions, which are the maximum depth ***maxDepth*** and the maximum size of the priority queue ***maxSize*** in order to avoid too complex computations and avoid errors and exceptions.

The **output** is definitely the state with the lowest Mondrian score, considered as the most optimal found solution. We will be calling it the best state during the algorithm explained below.

To start with, we build our initial tiled puzzle of size n x n as explained in part 2.1.2. We are preparing ourselves to iterate through the graph (quaternary tree), starting from this initial non-optimised tiled puzzle. The depth of the initial state is of course zero and its Mondrian score can be easily computed. We recall that the Mondrian score is the difference between the largest and smallest rectangle area in a tiled grid.

The A-Star algorithm is based on a **Priority Queue**. In our case the feature that will give a higher priority to one state instead of another is its global score, also defined as the sum of its depth and its Mondrian score. The priority has a maximum size to avoid useless and costly computations. We would like to consider only the most useful states and they will guide us more efficiently and faster to our goal state.

To begin with, we start by adding the initial state in the priority queue, by default we understand that it is also assigned to be considered as the best state at this stage of the research because

no other states have been discovered. Afterwards, we start the main loop of the algorithm that will stop only when the priority queue will be empty. We begin the search by dequeueing the state on-top of the queue, which will always be the case as we need to consider the most relevant state to carry on the algorithm after each computation. We generate four possible new valid layout grids by applying the four possible actions : Split, Merge, Split-Merge, Merge-Split. Let us call them S_1, S_2, S_3, S_4. These four states possess two major variables : their depth in the algorithm, which is equal to one here as it is the first iteration, and their respective Mondrian score, which might be closer to the goal state's Mondrian score.

One major requirements is to not consider previous discovered states, otherwise, the algorithm might possibly loops endlessly. Once a state has been taken into account by the algorithm, we need to set its visitedness to *true*. In that way, the algorithm will not consider it twice and will carry on looking for the next new generated grids neighbours.

Between each generation of a new state coming from one of the possible actions, each time the global score of the current studied state is less than the current best state's score, we update by assigning the current state to be the new best state.

Every time we generate a new state, while the priority queue is not full, we add it into it, otherwise we check whether the global score's current state is upper or lower the lowest global score of the priority queue and we proceed to the insertion of the current state if it is above and as a consequence, we remove the state with the lowest global score.

The algorithm can carry on by generating all the sixteen new states because $4^2 = 16$ by updating the priority queue and the best state at each iteration.

In the end, the algorithm returns the best state, which is the most optimal state found by the algorithm, defined by the lowest Mondrian score found.

2.5 Status of the code

As shortly explained before, the program takes the size of a grid as parameter and then follows the steps described in the previous sections in the report. Starting from the construction of an initial state, then applying the different actions to this initial state in four different forms respectively corresponding to the **Split()**, **Merge()**, **SplitMerge()** and **MergeSplit()** actions. In this way, we build recursively a quaternary tree by applying to each new generated state, the four actions every time.

Let us explain the structure of the **.java** files and how they are related all together. We have mainly three classes : **Graph.java**, **State.java** and **Rectangle.java**, apart from the main function.

The **Rectangle** class is mainly used to define the structure of each rectangle shapes we add to grids to build a proper and valid puzzle. A Rectangle is created with a **length**, a **width** and an **index**. I assume that the concept of length and width are quite straight forward to understand. As far as the index is concerned, it corresponds to the integer value assigned to a specific rectangle within the grid. Each rectangle covers an area which is equal to the product between length and width. Finally, each rectangle possess as well its coordinates, which is an integer array of size four. Here is an illustrated example :

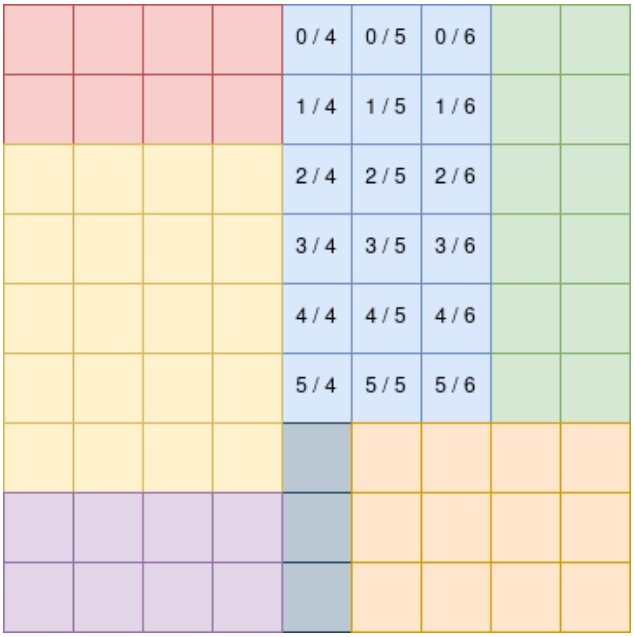

FIGURE 21 – Example to illustrate the notion of coordinates

In this grid the blue rectangle's coordinate array will be : **[0, 5, 4, 6]**. The first integer **0** refers to the first index (row) in the vertical axis where a blue cell is drawn. The second integer **5** is the last index (row) in which a blue cell is drawn at the bottom. We apply the same method for the horizontal axis. The integer **4** is the first index (column) and **6** is the last index (column).

The **Rectangle** class comes with getters and setters in order to keep a proper structure within the code.

The **State** class is most probably the main class as it has lots of attributes and methods. A state is build only using an integer which corresponds to its grid's size. Each state has an arraylist of Rectangles fitting all together in the grid. This grid is a two-dimentional integer table. Each state has a mondrian attribute that refers to its Mondrian score, a **depth** attribute referring to the state's depth in the quaternary tree. It corresponds to the number of actions that have been needed to reach this state. The **globalScore** attribute is equal to the sum of these two previous attributes. This notion is related to the implementation of the A-Star algorithm. Finally, the last attribute is a boolean variable asserting whether a state has already been reached in the tree. This class provides its usual getters and setters, as well as useful methods such as **getMondrian()**, **isValid()** and the actions methods allowing the quaternary tree, also known as state space, to be built. The getMondrian() method compute the Mondrian score of a state. The isValid() method is used to check whether a state with its list of rectangles respect the Mondrian Tiling Problem conditions.

The **Graph** class enables the construction of the state space. It is build using an initial state that corresponds to the root. A Graph is composed of **states**. That is the reason why one of the main attributes is a list of states. It mainly has getters, setters and particularly a display method which allows to display the state space in the shell.

As far as the **main** is concerned, unfortunately all the desired features and methods have not yet been implemented due mainly to a lack of time. However, the code is able to build an initial state and was about to perform the Split() action and then building the state space using a priority queue. The main ideas can be retrieved looking and understanding the written code. For your information, the code is more likely to produce a result for a square size greater than or equal to 5.

2.5.1 Next step

While building the state space, the aim is to apply the A-Star algorithm where you can find the code in the **Main.java** file. You can find the explanations in the section 2.3.4. To sum up the ideas of applying the A-Star algorithm is to use a Priority Queue while generating new states by applying the four actions. At each step, we rank all the new states into the Priority Queue with the ranking bases of their Global score (Mondrian score + Depth). The A-Star algorithm allows us to avoid useless computations and reach the more optimal state much faster. Unfortunately, any simulations could have been performed, however, based on the difference of complexity involved in a Brute-Force algorithm on one hand and the one using A-Star algorithm must be quite clear. The second option is the best as it guides the algorithm in the right way from the beginning to the

end. It is important to acknowledge that my algorithm, even once completed, does not provide the most optimal solution because the **maxDepth** and **maxSize** variables restrict this search. The main quality of this algorithm is to provide a solution very close to the optimal one in a very reasonable execution time.

2.6 Adapt the program to solve a x b grid

Currently, I am calling the **Main** function using only one parameter corresponding to the **size** of the grid. This integer dimension fits for all four edges of the grid because it is a square. We now want to change the dimensions of the grid and we would like it to shape a rectangle with different sizes for its length and its width. The major change would be to add another argument while calling the main function. The first argument would be the length of the grid and the second would be its width. Once these two arguments have been sucessfully specified, as we used to retrieve the size of the grid previously, we now declare two different variables corresponding to the length and width of the grid. Apart from renaming a few variables, the structure of the code will remain the same. In the **for loops**, we will only have by now to distinguish between the horizontal and vertical axis and take care of the maximum bounds we are allowed to reach, in order to avoid any ***"out of bounds errors"*** while accessing to specific coordinates in the two-dimentional grid table. As far as the **State** class is concerned, we will add a new attribute so that we can access to its grid's length and grid's width. One example of implementaiton change is while creating a **state object**, we will take care to specify the new dimensions **a** and **b** to the grid's size.

2.7 Complexity - Data Structures choices

In this section, I will explain my major choices of data structures that helped me to optimize the efficiency of my programs. First of all, I chose **Java** to code this Mondrian Tiling Problem because I have found it a good idea since we were required to handle objects such as rectangles, states and finally a graph, gathering all these objects. Java language is well built to fit oriented-objects programmation.

- A first choice is the use of a **LinkedList** called **rectList** storing the different shapes of rectangles fitting in a grid. We prefer using a LinkedList rather than an ArrayList because LinkedList are better for manipulating data and during the execution we need to update quite a lot.
- I did the same choice for the adjacency list of neighbours in the state space.
- Last interesting data structure used in the project is the **Priority Queue** in the main function to perform the A-Star algorithm. It helps us to keep a good structure in the code and it

assures us to always consider the most relevant state and look for the most optimal state at each loop. The priority is based on the global score of a state.

— More generally in the code, I paid attention to optimize the actions within each loops to avoid running them twice unnecessarily.

2.8 Conclusion - Personal retrospective

As far as the topic of the project is concerned, I really enjoyed solving the Mondrian Tiling Problem. It helped me better understand the concept and also to solve a practical exercise related to Artificial Intelligence and the use of state space method. Besides trying to solve the coursework's question, I still took care about the Java best practice to use, such as adding getters and setters to classes to access private attributes, using enhanced for loops, proper commenting to improve your understaning of my work.

To conclude on the technical aspect of the projet, the goal is to continue developing the four different actions and being able to build the state space respecting size conditions. Then, I would have to find the perfect trade-off between complexity (time and memory storage), efficiency and effectiveness. The time complexity is the amount of time it takes to output the result. The memory storage is how many bytes does our algorithm need in memory to be executed. Efficiency refers to the ratio of useful work that have been done to get the outputed result. Finally, the effectiveness refers to how much successful the output is from the known most optimal result you can find for a specific size of grid solving a Mondrian Tiling Problem.