Mondrian Tiling

1. Introduction and Contents

The code can be ran by simply pressing the run button. The parameters of the algorithm can be changed on line 368.

2. Defining the States and Actions

2.1 States

The Mondrian Tiling problem has four distinct types of states:

1. Starting state: this is the starting state of the square before any actions have taken place (being a singular square). In this state, the Mondrian score is not calculable as at least two rectangles are needed. Seen in Figure 1.
2. Intermediate state: this is a state which is in the set of all possible valid configurations that the problem can be in, between the starting state, and the

best solution state. It can also be considered to be a “partially-tiled” state. However, it is important to note, with the definitions of possible actions below, the square is always technically “fully-tilled”, as we consider there to be no blank spaces. Seen in Figure 2.

3. Best solution state: this is when no further optimisations to the problem can be found and the square is considered to give the lowest possible score. Seen in Figure 3.
4. Invalid state: this is a state where the configuration is invalid. There are multiple different ways variations of an invalid state:
    a. Solution contains overlapping tiles
    b. Solution contains a rectangle with a non-integer side
    c. Solution contains a congruential rectangle pair
    d. Solution is not completely covered
    e. Solution contains no tiles/a singular tile (the starting state)
    f. A tile is non-rectangular

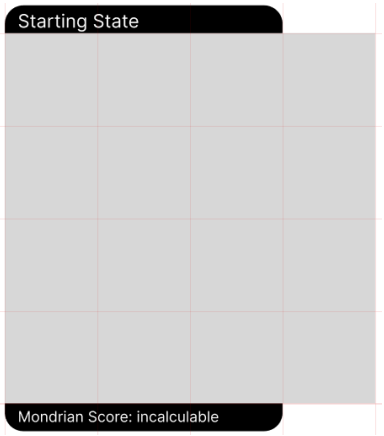


Figure 1: Starting state for a 4×4 Mondrian tile. Mondrian score is incalculable

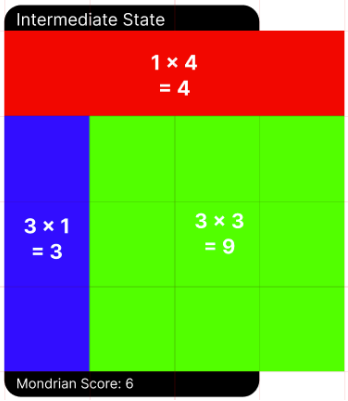


Figure 2: Intermediate state for a 4×4 Mondrian Tile. Mondrian score is 6

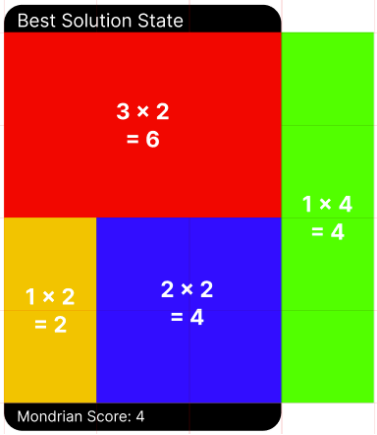


Figure 3: Best Solution state for a 4×4 Mondrian Tile. Mondrian score is 4.

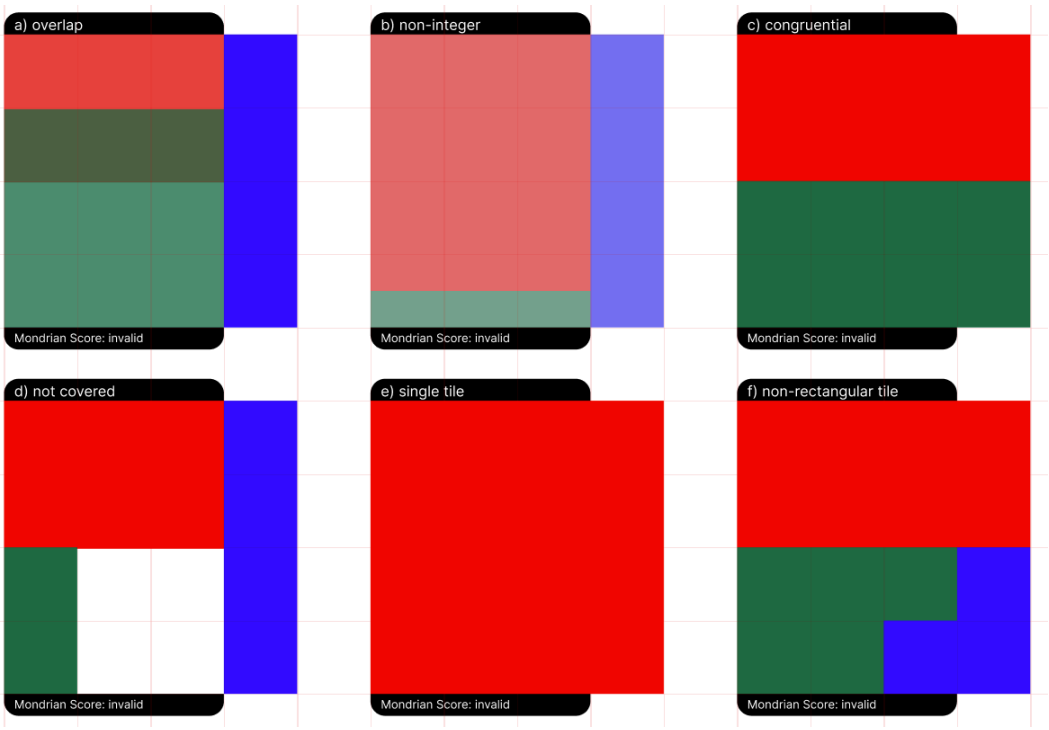


Figure 4: Invalid states for a 4×4 Mondrian Tile

2.2 Actions

The problem can be thought to contain three distinct actions:

- Split: splits a tile into two tiles
- Merge: merges two adjacent tiles within a boundary into one tile
- L-merge: merge a tile into another adjacent tile in the shape of an L - the number of tiles stay the same.

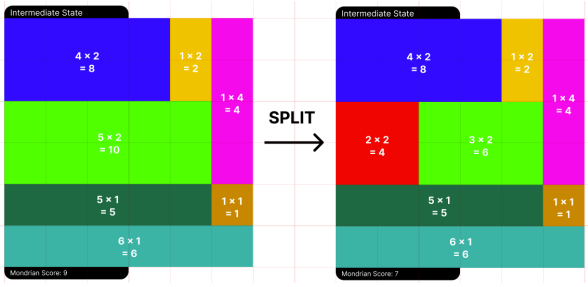


Figure 5: Example of a Split on a 6×6 Mondrian Tile.

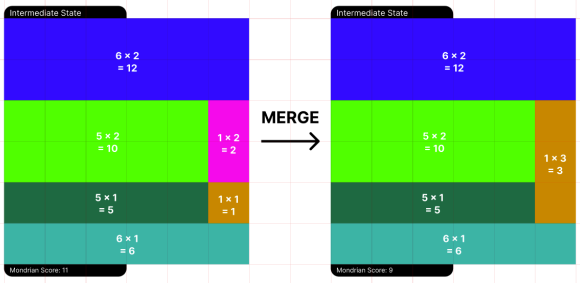


Figure 6: Example of a Merge on a 6×6 Mondrian Tile

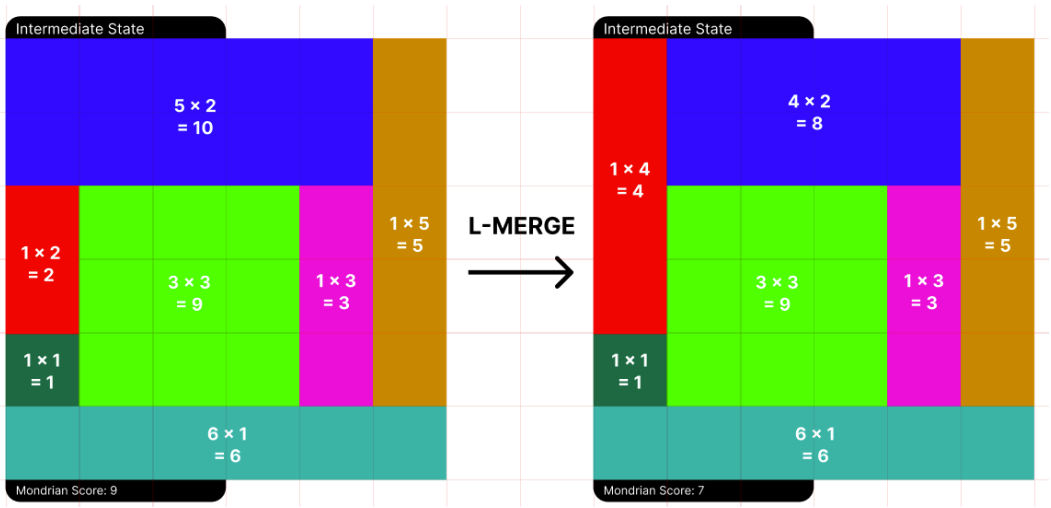


Figure 7: Example of an L-Merge on a 6×6 Mondrian Tile

3. Actions leading to invalid states

3.1 Can my Definitions of States and Actions lead to Invalid States?

With these defined states and actions, there is not the notion of "*adding*" or "*moving*" a tile to the solution. Instead, tiles are always either ***split*** or ***merged*** in some way. This method means some invalid states can never occur: tiles can never overlap each-other [4a], the solution is always considered to be covered [4d], and tiles can only ever be rectangular [4f].

[4e] is avoided by ensuring the solution never transitions into the starting state (which is invalid). In this state, the Mondrian score is incalculable as there is only one rectangle. Therefore, the first action must always be to split the square. Conversely, a merge cannot occur when there are only two rectangles, as this would lead back to the initial state.

3.2 Validate State Pseudo-code

```
INPUTS:
- state: state to be checked
    - state.tiles contains a list of every Tile object within the
    current state.
```

```
  - new_tile is a new tile which has just been added to the state
  - Each Tile object has a width and height attribute
- action: action to occur / just occurred

OUTPUTS:
- boolean variable stating whether state is valid or not


PROCESS:
<BEFORE ACTION HAS TAKEN PLACE>
1. Check merge cannot occur if LENGTH OF state.tiles <= 2

<AFTER ACTION HAS TAKEN PLACE>
2. Check new_tile has not led to congruent tiles in state.tiles
3. Check no tile has a non-integer side

<ADDITIONAL CHECKS - NOT NECESSARY DUE TO MY DEFINITION OF STAT
ES AND ACTIONS>
4. Check tiles in state.tiles do not overlap
5. Check state is fully covered
6. Check tiles in state.tiles are all rectangular

<EXPANDED PSEUDO-CODE FOR SECTIONS 1, 2, AND 3>
1.1 IF action = merge THEN
  1.1.1 IF LENGTH OF state.tiles <= 2 THEN
    1.1.1.1 RETURN FALSE
    1.1.2 ELSE RETURN TRUE

2.1 FOR tile in state.tiles
  2.1.1 IF new_tile EQUALS tile THEN
     2.1.1.1 RETURN FALSE
2.2 RETURN TRUE

3.1 FOR tile in state.tiles
  3.1.1 IF tile.width < 1 or tile.height < 1 THEN
```

```
        3.1.1.1 RETURN FALSE
3.2 RETURN TRUE
```

3.3 Should Transitions to Invalid States be Allowed?

Some invalid states should never be transitioned into and should be avoided (as mentioned in [3.1]). Entering these invalid states provide no benefits to finding an optimal solution. Instead, they only increase the state-space to be searched.

Regarding invalid state [4b], there is no benefit to entering a state containing non-integer sides rectangles, so this should always be avoided also.

However, an ideal solution would allow transitions into invalid states with congruent rectangles [4c]. This is because with the actions defined, some states are only accessible if congruent tiles are temporarily allowed. This can be seen in Figure 8, which shows some potential paths to an optimal state for a 4×4 Mondrian Tile. This best solution state can only be reached if congruent tiles are temporarily allowed. This was not included in my solution as it increases the state-space to be searched by a considerable amount, decreasing the efficiency of the solution.

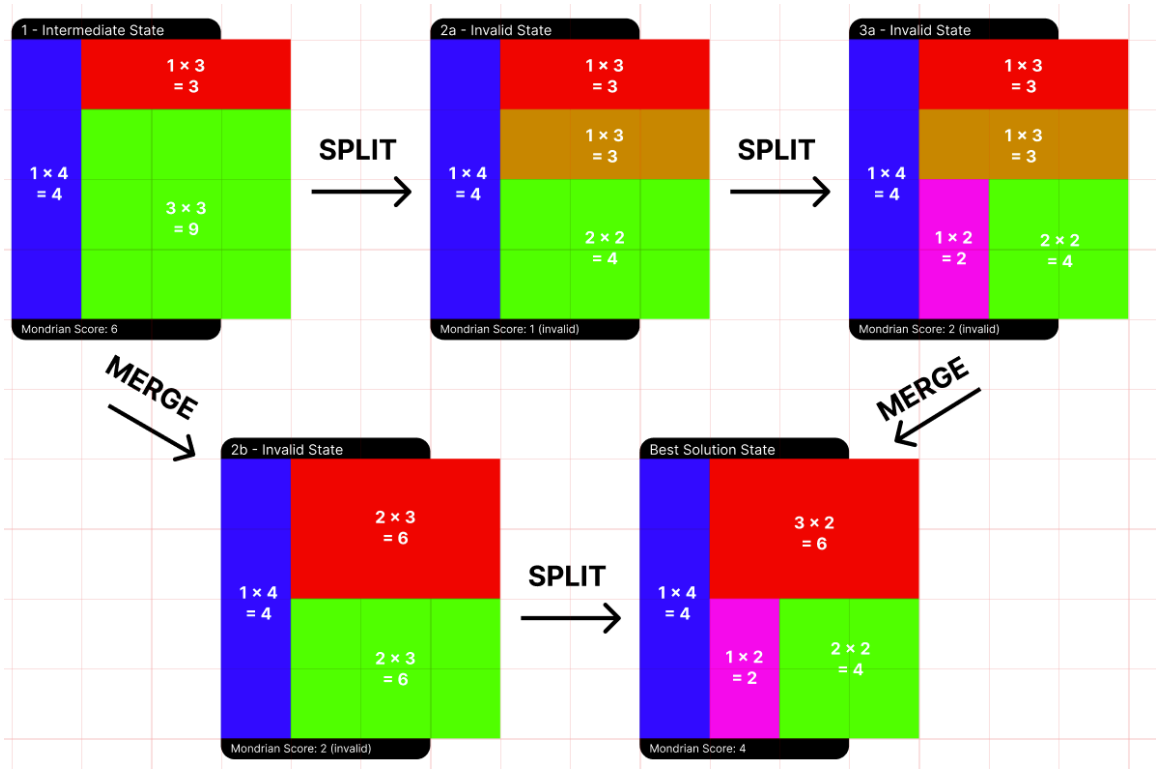


Figure 8: Potential paths from an intermediate state to a best solution state for a 4×4 Mondrian Tile. These paths include invalid states

4. Compute Mondrian Score of a State

INPUTS:
- state.tiles contains a list of every Tile object within the current state.
- Each Tile object has an area attribute, calculated via Tile.width * Tile.height

OUTPUTS:
- Mondrian score of given state. If a state is invalid, the output will be infinity.

PROCESS:
1. SORT state.tiles BY Tile.area WITH LARGEST FIRST
2. IF LEN(state.tiles) < 2 THEN
  2.1 RETURN float("inf")
3. SET largest TO state.tiles[0]
4. SET smallest TO state.tiles[-1]

5. SET score TO largest.area - smallest.area
6. RETURN score

5. A Discrete State-space Search Method

5.1 Solution Description

An A* search algorithm was chosen as the solution. I believe this to be the optimal algorithm for the solution as it encourages exploration of different solutions whilst also favouring the path of the most optimal solution.

The solution creates a tree, where each node is a new state made via one of the three actions, meaning it has a branching factor of three. The root of the tree is an initial state, which has been randomly generated by making random splits from a singular tile. A diagram of the tree can be seen in Figure 9. This tree expands until the limiting factor is reached (mentioned in section 5.2). At this point, the state with the lowest Mondrian score is returned.

The algorithm decides which state to operate on by keeping a Priority queue of states to operate on. This queue is sorted via an evaluation function $f(s)$, for state s. This is:

$$f(s) = h(s) + g(s)$$

where $h(s)$ is the Mondrian score of state s and $g(s)$ is the depth of s in the tree.

Using this evaluation function means the solution will encourage exploring the states which are most likely to improve - improving the efficiency of the solution.

Finally, this entire algorithm is repeated multiple times with different randomly generated initial states. The state with the best score out of every iteration is outputted.

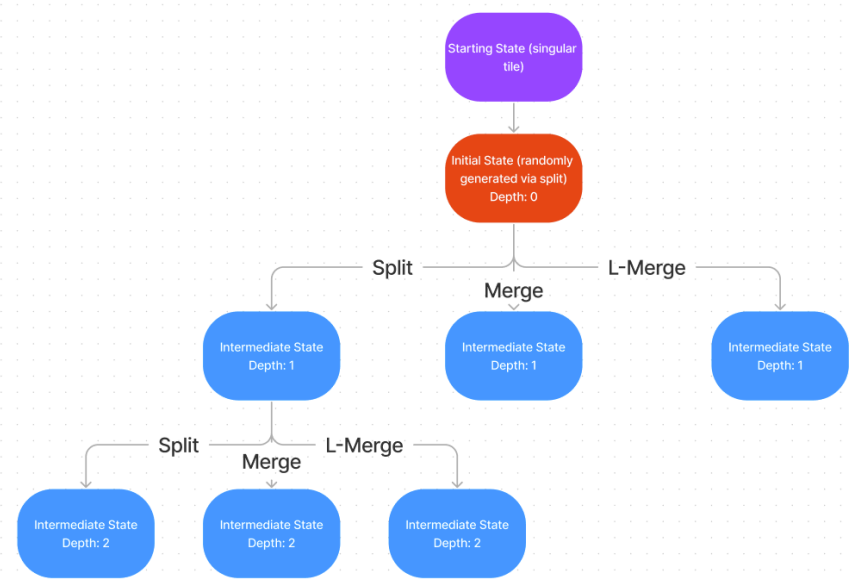


Figure 9: Example of start of tree generated via A* algorithm.

5.2 The Limiting Condition, M

My solution uses different metrics to limit the algorithm, these are:

- maximum depth of tree: this metric is what stops the A* algorithm from exploring any deeper in the tree and directly relates to how many operations are allowed on any given initial state (the depth = how many operations one state has undergone). Examples of different depths can be seen in Figure 9.
- maximum queue size: this is the maximum size of the priority queue which holds every state to be operated on.
- number of iterations: this is how many times the overall algorithm repeats. Each iteration a new tree as seen in Figure 9 is shown, with a different random initial state. The best state out of all these iterations is taken.

5.3 Performance of M

My most important metric is the number of iterations. This is because the algorithm is largely dependent on the random initial state, which can drastically vary what the resulting best Mondrian Score is.

For this experiment, the other metrics have been set to the following:

- maximum depth= 10

- max queue size = 20

5.3.1 Results

Number of Iterations	a=8		a=12		a=16		a=20	
	Score	Run-time (s)	Score	Run-time (s)	Score	Run-time (s)	Score	Run-time (s)
1	6	0.204	11	0.225	16	0.521	25	0.322
10	6	1.236	8	3.127	15	1.627	20	4.693
25	6	4.237	8	4.780	12	7.404	14	9.017
50	6	7.071	8	11.727	12	13.551	14	16.596
100	6	14.082	8	28.466	12	29.982	13	46.112
200	6	28.084	8	51.589	12	75.479	13	61.451

Table 1: Results showing Mondrian score and run-time for different number of iterations.

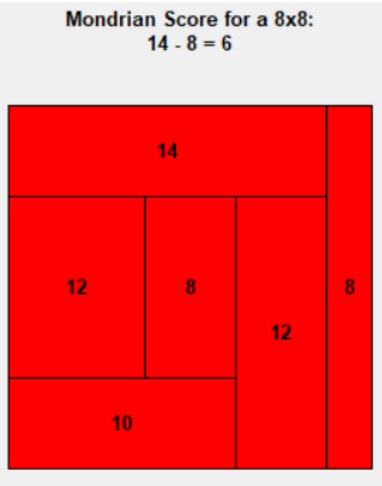
a=20, iterations=10	
Score	**Run-time (s)**
14	4.215
19	2.830
18	3.053
14	1.644
21	1.769
18	2.842
19	2.588
21	2.285
14	2.573
20	4.255
AVG: 17.8	**AVG: 2.805**

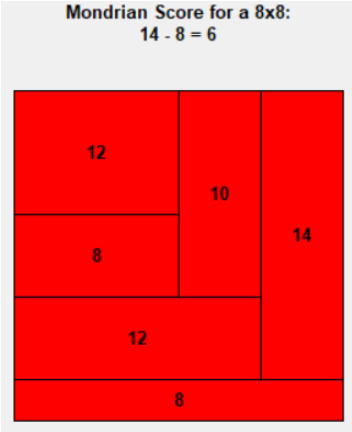
Table 2: shows average time score and time for a=20, with iterations=10

Figure 10: Graph of number of iterations vs run-time for different values of a.

5.3.2 Visualisations

For each value of $a = 8, 12, 16, 20$, visual representations were generated as shown in the Figures below. For each a, 3 different visualisations have been shown, to show the variability in possible results.





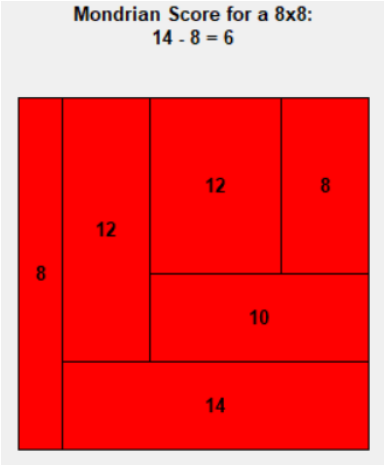
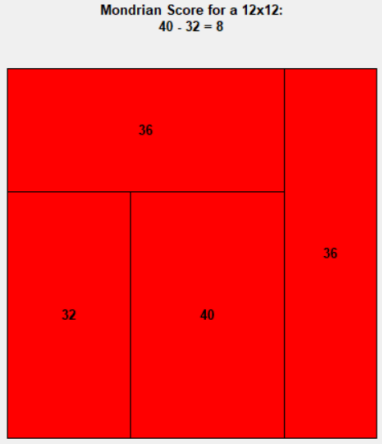
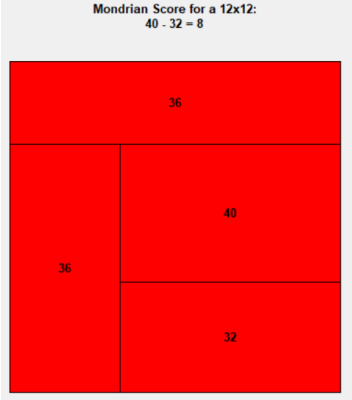


Figure 11: Visualisations of 8×8 Mondrian





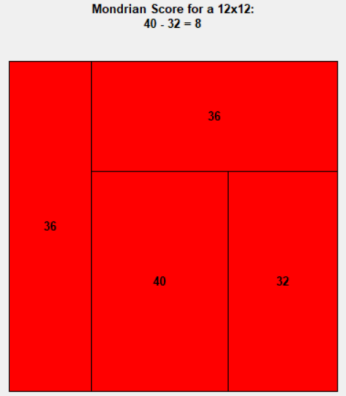
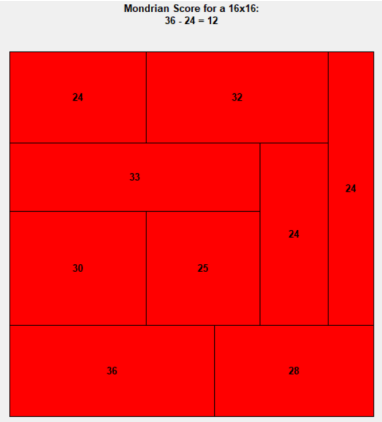
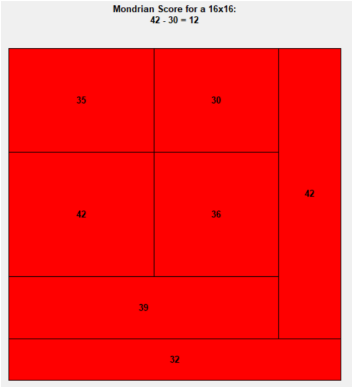


Figure 12: Visualisations of 12×12 Mondrian





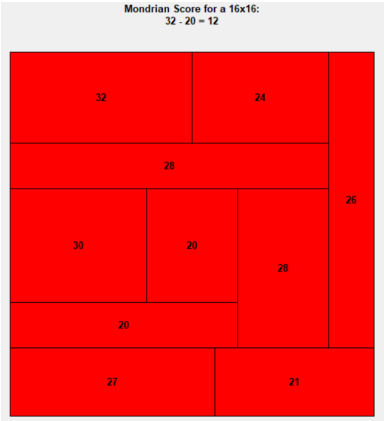
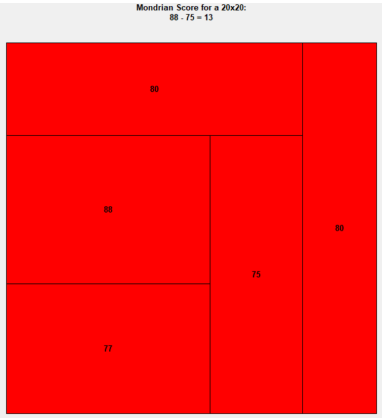
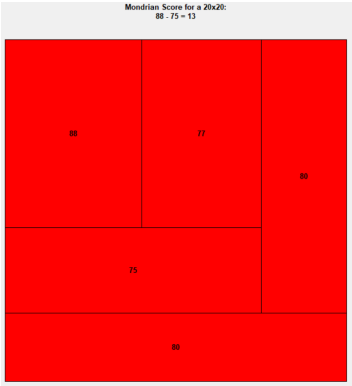


Figure 13: Visualisations of 16×16 Mondrian





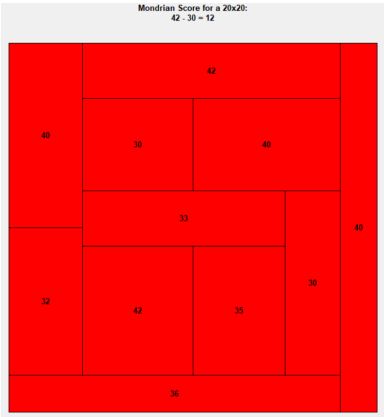


Figure 14: Visualisations of 20×20 Mondrian

5.3.3 Discussion

As seen from the Table 1, for $a = 8$ and $a = 12$ the number of iterations for the most part did not have an affect on the Mondrian score. Allowing an optimal Mondrian score to be found in a few seconds with < 10 iterations. Making it very efficient.

For larger sizes ($a = 16$ and $a = 20$), a larger number of iterations was required to get a good score.

For a 20×20 Mondrian tile, this usually required at least 50 iterations (taking around 15-20 seconds) to get a score of 14. To get a score of 13, it usually requires at least 100 iterations. A score of 12 was recorded once (seen in Figure 14).

When $a = 20$, the score could widely vary for lower iterations. Table 2 shows the variability of scores and run-times when the number of iterations is set to 10. It shows for larger a, a larger number of iterations is required to get consistent results.

Across all sizes of a, the number of iterations had a direct relation to the run-time. As the number of iterations increases, as does the run-time. This can be seen in Figure 10, which depicts a near linear relationship between the two variables. For $a = 8, 12, 16$, it can be seen that as a increases, the run-time also increases (for iterations > 10). However, once again, the variability of $a = 20$ can be seen as it's run-time can vary by large amounts.

Overall the the solution did well to get an optimal score for sizes $a = 8$ and $a = 12$ in a very quick time. For $a = 16$, the solution can reach a near-optimal score in a relatively quick time (less than 10 seconds) - making it efficient. For $a = 20$, the solution requires much longer to run to get a score of 13 - which is 4 off of the actual optimal score of 9 for a 20×20 Mondrian Tile. Therefore there is room for the algorithm to be made more efficient with scalability in mind.

5.4 Improvements

The algorithm would work just the same with a rectangle of $a \times b$ instead of $a \times a$. This is because each action is based off the coordinates of the rectangle undergoing the action - the actual dimension of the overall Mondrian Tile is never considered and is only used to place the initial tile. Therefore it would be an easy change to make.

A possible improvement to make would be to allow some invalid states - this is because there are some states which can’t be accessed without temporarily allowing a congruent tile (as discussed in section 3.3).