AI Coursework: Mondrian Tiling problem

2460800V

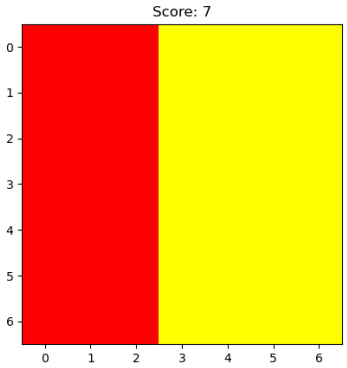
November 2022

1 States and actions

The states for this problem are all possible combinations of tilled $n \times n$ squares. For example, let s be a 7×7 square, tilled with two rectangles. The representations of s are shown. Figure 1a shows how the agent views s and Figure 1b shows how we can visualise s. Both of these and similar graphics used throughout the report were generated in `visualiseActions.py` which is explained further in Section 4.3.

1	1	1	2	2	2	2
1	1	1	2	2	2	2
1	1	1	2	2	2	2
1	1	1	2	2	2	2
1	1	1	2	2	2	2
1	1	1	2	2	2	2
1	1	1	2	2	2	2

(a) Agent view of s

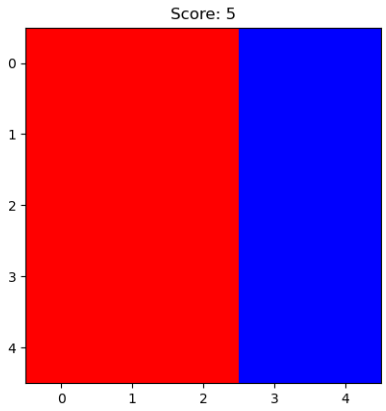


(b) Coloured view of s

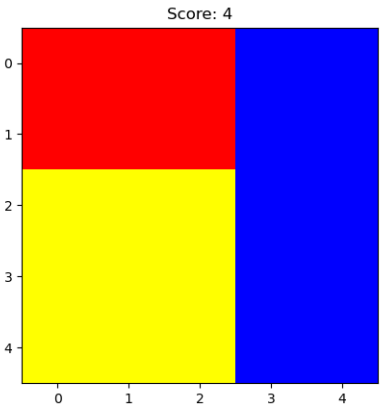
Figure 1: Representations of state s

There are several actions we want our agent to be able to perform to transition between states. Each of the actions should be taken with the goal of minimising the Mondrian score (explained in Section 3). The implementations of the actions are in the `actions.py` file. There are two possible actions;

1. Split - One way to improve the Mondrian score is to decrease the area of the largest rectangle in the square. We do this by splitting the largest rectangle in the square into two smaller rectangles. To visualise this action, let state s be a 5×5 square with two tilled rectangles as shown in 2a. We perform a split, resulting in the red rectangle being split in two. This transitions leaves us in a new state s' as shown in 2b. We can see the state s' has a lower Mondrian score than state s.



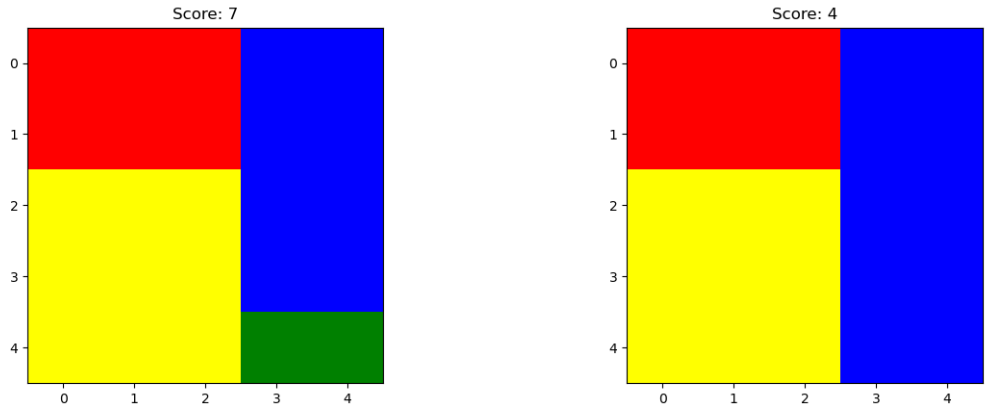
(a) State s



(b) State s'

Figure 2: States before and after split action

2. Merge - Another way to improve the Mondrian score is to increase the area of the smallest rectangle in the state. We do this by merging the smallest rectangle with one of its neighbours. To visualise this action, let state s be a 5×5 square with four tilled rectangles as shown in 3a. We perform a merge with the blue and green rectangles resulting in a new state s' as shown in 3b. Again, we see that state s' has a lower Mondrian score than state s. In my implementation, the *merge* function returns a list of candidate merge options so we can evaluate our options before performing the action. This is also to allow for simulated annealing (explained further in Section 4).



(a) State s (b) State s'

Figure 3: States before and after merge action

2 Invalid States

2.1 Check for state validity

Algorithm 1 shows how we check if performing a given action to a given state will result in a valid state. In my implementation, this is spread over several functions in the `validateActions.py` file.

Algorithm 1: Check if performing action a on state s results in a valid state

Data: State s, Action A
Result: l, state validity of s'

$s' \leftarrow A(s)$;
$L_u, dimensions \leftarrow [\,], [\,]$;
$unique \leftarrow$ unique numbers in s';
for *row in* s' **do**
  for u *in unique* **do**
    L_u.append($s[row[u]]$);
foreach L_u **do**
  if L_u *not in order* || *elements in* L_u *not the same* **then**
    $l \leftarrow false$;
    return l;
  $dimensions.append([len(L_u[0]), len(L_u)])$;
if *all element of dimensions are different* **then**
  $l \leftarrow true$;
else
  $l \leftarrow false$;
return l;

2.2 Allow invalid states

We want to allow the merge and split operations to go to invalid states. By doing this we allow new states to be reached much faster and can apply operations to any invalid states in the future to hopefully reach valid and lower scoring states. There are disadvantages to this decision, namely; we need to search the space for a lot longer as there are now vastly more states that we can traverse through. Additionally, there is no guarantee that performing another action on the invalid state will lead to a valid one. To reinforce this design choice, we give a concrete example. Let s be a tilled 5×5 grid (Figure 4). Say we perform a *merge* on s resulting in the new state s' (Figure 5). s' is clearly not a valid state however, if we split s' we land on a new, valid and lower scoring state s'' (Figure 6).

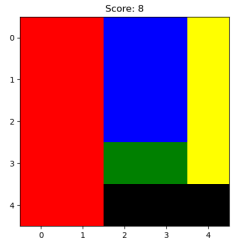


Figure 4: State s

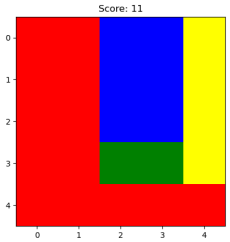


Figure 5: State s'

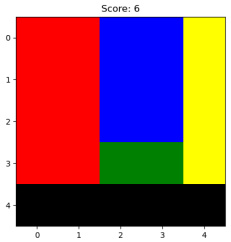


Figure 6: State s''

3 Mondrian Score

The Mondrian score of a given state is defined as the area of the largest tile minus the area of the smallest tile. Algorithm 2 shows how to compute the Mondrian score of a given state.

Algorithm 2: Find Mondrian Score

Data: State s
Result: The Mondrian score of s

```
counts ← {};
for row in s do
    for element in row do
        if element in counts.values() then
            counts[element] = counts[element] + 1;
        else
            counts[element] = 1;
sorted_counts = {sorted(counts.values())};
return sorted_counts.values[0] − counts.values[−1]
```

It is worth mentioning that in my implementation I use numpy's *unique* function to find and count all the unique values in one go. Moreover, I return a score of 100 if the given state s is tiled with just one rectangle to discourage this state.

4 State search

My implementation is a stochastic best-first search which uses two arrays to keep track of states to visit and states which we have already visited respectively. We use best-first search to avoid taking a bad path as this could be detrimental to our search. We initialise an $n \times n$ grid (using my *initialiseGrid* function) with either one tile (for even n) or two tiles (for odd n). At each iteration, we add one new state from the merge options and one new state from split. We use simulated annealing to occasionally (10% of the time) take a bad option in an effort not to get stuck in a local minima. The code for the search is in the `solve.py` file.

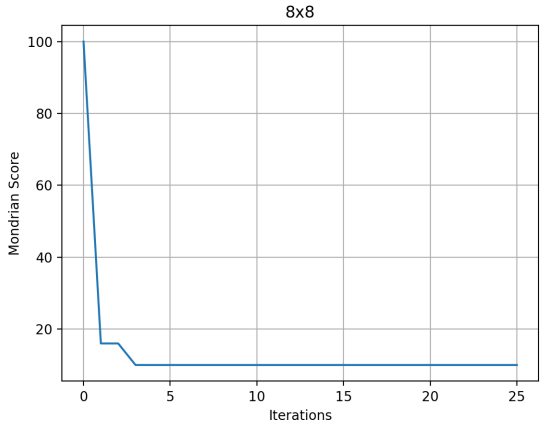
4.1 Parameters

The *SolveMondrian* function takes two parameters; a, the dimension of the grid, and M, the maximum depth of the tree. During the search, we keep track of where we are in the tree, specifically our depth. We stop the search either when we reach a depth $>= M$ or when the list of states to search is empty (ie we have reached a leaf node). M can be seen as the number of iterations we run our solve method for.

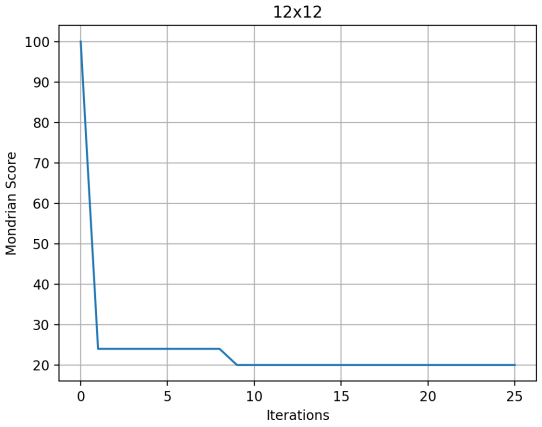
4.2 Search performance

We now solve the Mondrian tile problem for $a = \{8, 12, 16, 20\}$ with our implementation of *SolveMondrian*. We plot the values of M (iteration number) versus the Mondrian score to show how our best score varies as we search. We can also read how quickly our method converges. The best score found for each state with $M = 25$ is:

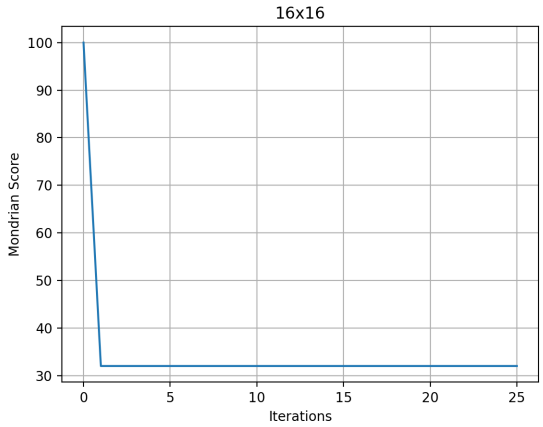
a	Mondrian score
8	10
12	20
16	32
20	40



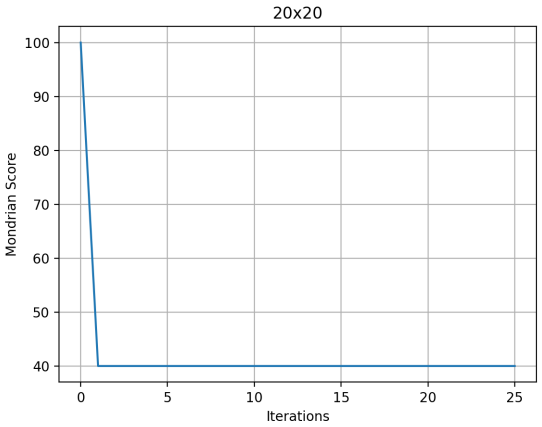
(a) 8×8 grid



(b) 12×12 grid



(c) 16×16 grid



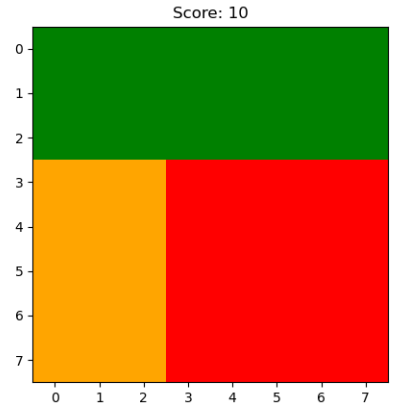
(d) 20×20 grid

Figure 7: Performance for varying a's

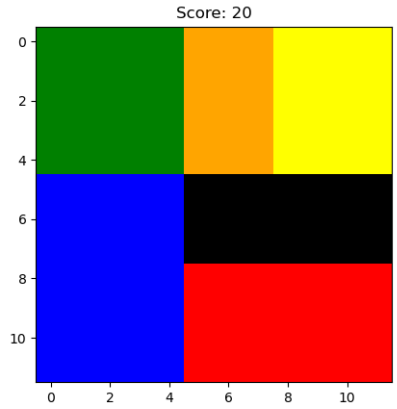
From the plots above, we can see that our implementation converges quickly for every value of a.

4.3 Representation

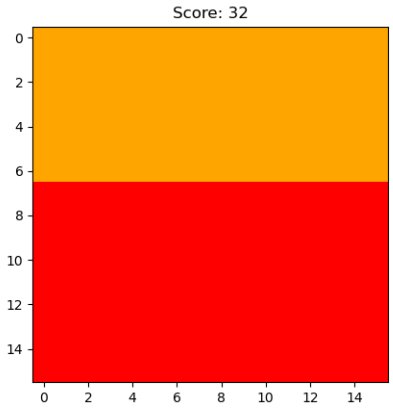
To visualise the states, we use *matplotlib* to plot and colour a given numpy array. This works by simply giving each unique value in the array a specific colour and displaying the colours as a plot. We show the visualisations of each optimal solution found for the varying values of a.



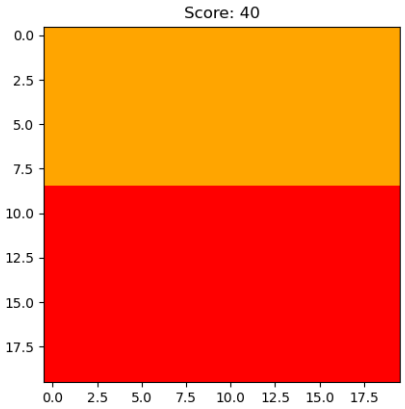
(a) 8×8 grid



(b) 12×12 grid



(c) 16×16 grid



(d) 20×20 grid

Figure 8: Visual representation of each optimal state

4.4 Discussion

In general, as the value of M increases the Mondrian score of the state decreases. This is expected as we only update the score when there is an improvement. As explained above (Section 3), our initial states are all scored 100 and from here we can begin our search. For all values of a, our solution converges quickly however, not to an optimal solution. Despite the stochasticity of the method we quickly get stuck in a local minima and we are unable to escape it. I tried increasing M to hopefully find a better solution however, this did not work. The solution is efficient as we explore two states with every iteration and converge quickly. We can also see from the visual representations that the optimal solutions for $a = 16$ and $a = 20$ are quite bad. Again, even with more iterations, the search does not find better solutions. For these two values of a, I experimented with the simulated annealing step but could not improve the best score. The decision to allow invalid states may be to blame as we never find better states even with more iterations. Overall, the state search did find a valid and low scoring solution for each value of a however, never the optimal solution.

4.5 $a \times b$ rectangles

To solve the Mondrian tile problem with arbitrary rectangles of dimensions $a \times b$, my implementation could be used with a few modifications. To show this, we go through all the parts of my implementation. The *initialiseGrid* function would need to be slightly modified to take two parameters a and b rather than just a, and return a $a \times b$ numpy array. The *merge* and *split* functions would largely work however, some changes would have to made to make them more robust. If either a or b were equal to one, my *getIndices* function, which finds the indices of the smallest and largest rectangles, would fail as we index the grid assuming it has two non-empty dimensions. No changes would have to be made to compute the Mondrian score of an $a \times b$ rectangle. Likewise, to check for state validity no changes to the current implementation would have to be made. The actual state search would not have to be changed at all and works just the same as before. Overall, allowing for $a \times b$ states could by incorporated very easily in my current implementation however, as with $a \times a$ states an optimal solution would probably not be found.

I experimented with some $a \times b$ states to study this. As expected, with $a \times b$ states $(a, b \neq 1)$ we find solutions after around 10 iterations. The visualisation of the states also works – to demonstrate this, Figure 9, shows the solution found for $a = 5$, $b = 8$.

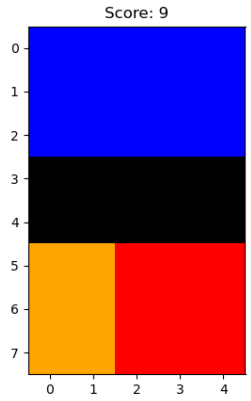


Figure 9: Solution found after 25 iterations with $a = 5$, $b = 8$.