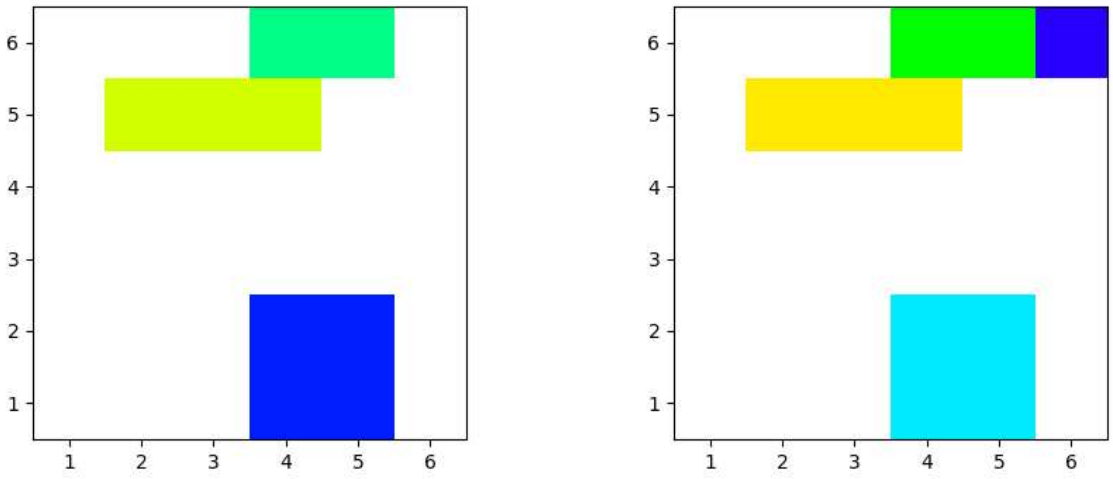
My initial ill-thought through idea to approach this problem produced very bad results, therefore I chose to start again and implement a different algorithm. However in order to avoid having to count all the time spent writing code for the first flawed “solution” as completely lost I’ll briefly describe the first effort.

Actions and states

I defined an action as placing a random rectangle on a grid. This took into account the presence of other - already placed - rectangles, so that invalid states were not reachable. The position and the length of the sides of each triangle were chosen randomly from a matrix of as-yet-not-used dimensions and positions.

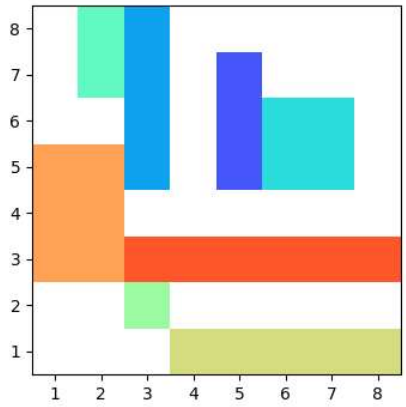
At each iteration the algorithm would create a random sample of rectangles that could be placed on the grid, calculate the Mondrian score associated with each of them and, following the addition of some random noise or “temperature” (according to a linear or an exponential temperature schedule), choose the best one.

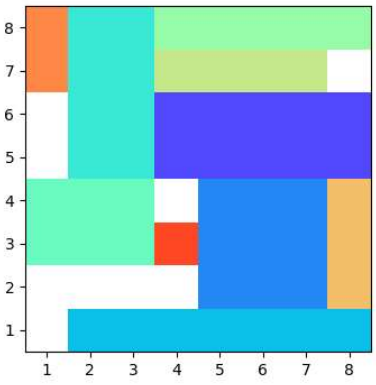


A transition between two states

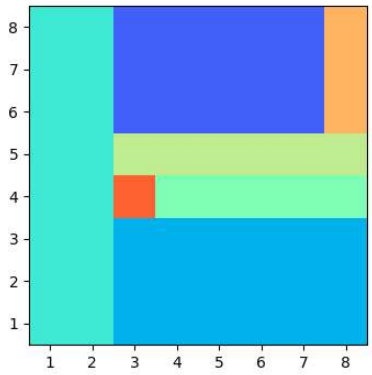
The results were terrible. The algorithm got stuck most of the time without filling the grid. For a grid of size 8 a typical result was 32 completed grids out of a 1000 runs of the grid filling algorithm, with the range of scores from 10 to 54, mean 24.2 and STD 11.7 .

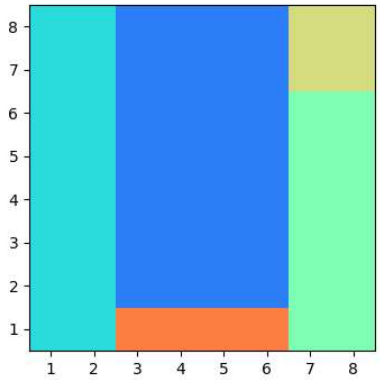
The only easily noticeable, apparently systematic, difference produced by varying the temperature parameters was that switching off the annealing altogether tended to result in around twice as many completed grids.

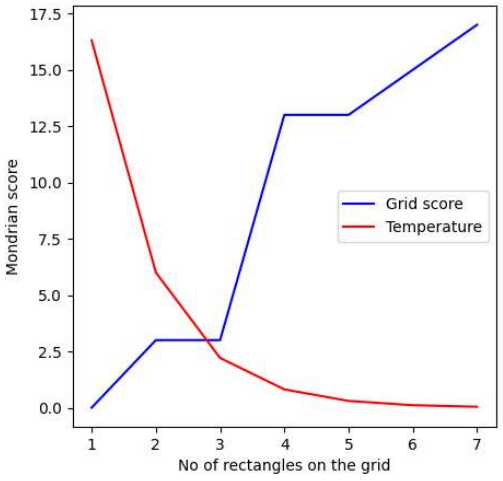


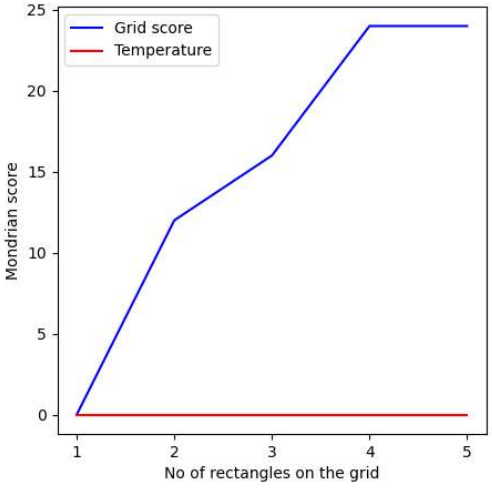


2 examples of “stuck” grids









Two successful (i.e. completely filling the grid) example runs of the grid-filling algorithm with different temperature settings and the produced solutions. “Temperature” denotes the variance of the noise (of mean 0) added to the potential Mondrian score associated with each rectangle considered for addition to the grid.

Second approach:

In my second approach I attempted to find the best solution using a very basic, limited form of q-learning combined with best first. This provided much better results than the first approach but still very far from ideal. In the end I should probably just have implemented a tree-search algorithm as suggested.

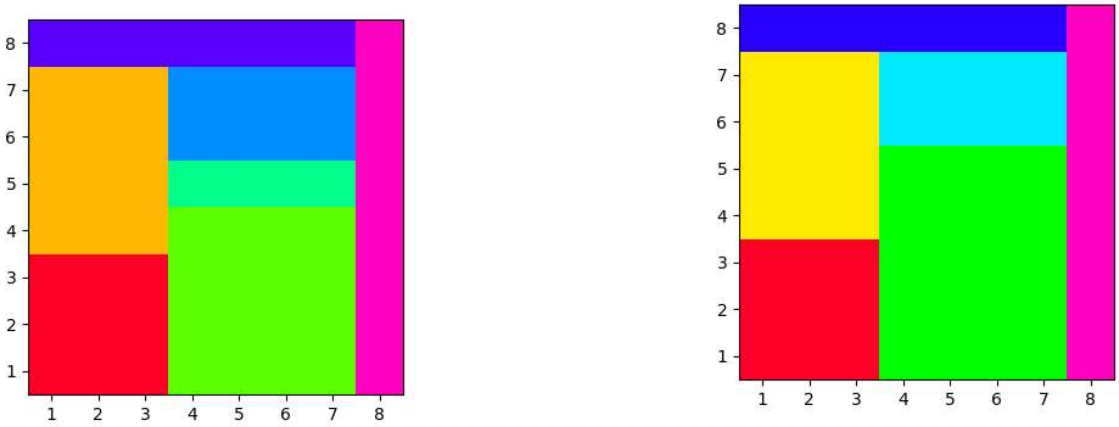
States: to make some q-learning feasible despite the enormous number of states (rectangle, positions on the grid, rectangle dimensions etc) a “state” is reduced to 3 variables
- no of rectangles on the grid
- Mondrian score
- binary label to indicate valid and invalid states

Actions:

- Do nothing: I included this action so see if the algorithm can learning anything and after a number of epochs starts settling on one value

- Split a rectangle with some randomness in the choice of split:
  I.e.: for the largest rectangle on the grid - collect a list of splits that result in a valid state and choose a random split- if no valid splits are available, check the next largest rectangle and so on
- Split a rectangle to max Mond score:
  as above but choose the split that maximises the Mondrian score of the resulting state

- Merge 2 random rectangles:
  Create a list of pairs rectangles that can be merged and merge a random pair
- Merge 2 rectangles to max Mond score:
  As above but choose a pair that after merging will result in the highest Mondrien score for the grid



Transition between two states: random merge

Actions leading to invalid states:

Input: a list of all rectangles on the grid, each entry: (height, width, area)
Output: boolean
Function: check is_valid

Sorted = sort each row of the input list reduced to the 1st two columns
Count = count the number of unique entries in Sorted

If the Count is equal to the length of the input list return True
If not return False

I planned to allow the actions to lead to states where there are congruent rectangles on the grid (but no overlapping) ones. However I thought that in order to do this I would probably need another action “Remove invalidity” where the algorithm attempts to merge or split some rectangles to get back to a valid state. My reasoning was that if I only applied a large penalty to being in invalid states the algorithm will simply learn to avoid those states and any benefits that allowing invalid states might provide - that is better exploration and not getting stuck in local optima - will vanish. On the other hand, if the penalty were small the search space would become many times larger. Therefore the plan was to have the algorithm sometimes get into an invalid state and get out of it after 1 or two steps. Such behaviour could probably also be achieved by careful adjusting of penalties, rewards and other parameters but I thought it would take longer to do. In any case, due to limited time I decided not to go with either option and simply not allowed invalid states at all.

Algorithm to compute the Mondrian score of a state

Input: a list of all rectangles on the grid, each entry: (height, width, area)
Output: integer
Function: get_mondrian score

Sorted = sort the input list on the last column
Return the difference between the largest and the smallest area in the list

A discrete state-space search method

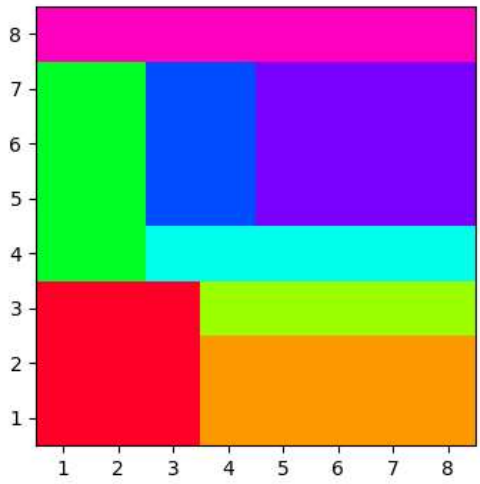
I chose q-learning in part as an exercise and in part to avoid the enormous search space that would have to be explored in a depth- or breadth- first search. I also couldn't think of a good heuristic function to use for an A* algorithm.

The idea was that with each epoch (where an epoch is defined as the run of the algorithm from the initial state of 1 rectangle on the grid until the max number of steps specified) the algorithm would improve its estimate of the best path to the optimal solution. Therefore the terminal state at the end of each epoch should have a progressively decreasing Mondrian score and in the meantime the exploration would serve as a (progressively improving - guided by the q-table) random search meaning that the algorithm could stumble on an even better solution than the one found at the end of any epoch.

My implementation has noise and noise_decay parameters to allow the algorithm to choose actions more randomly at the beginning to increase exploration, preventing it from only choosing the best action from the q-table when the table has not yet "settled".

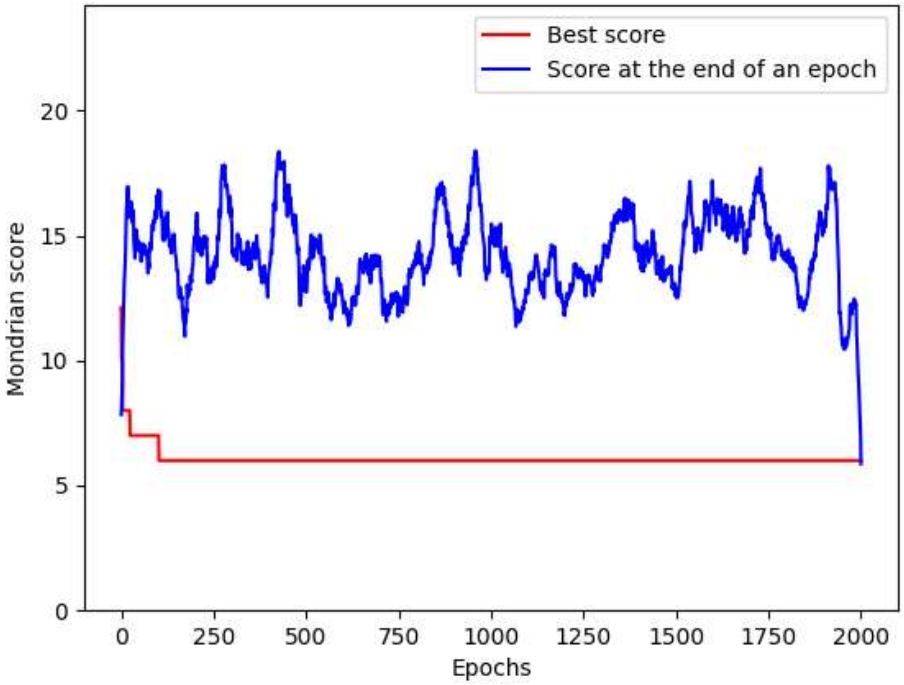
1. M can be the number of epochs defined as above.

2. and 3.



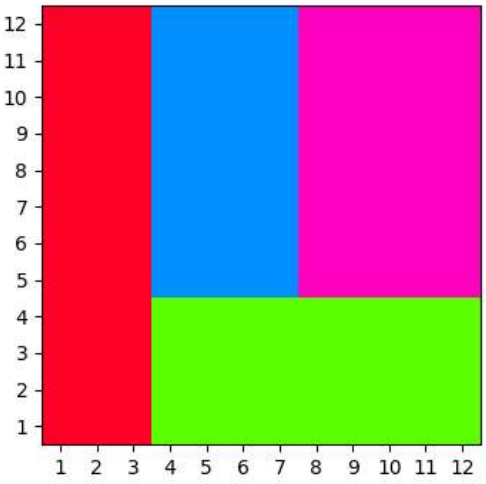
Grid size 8

Time taken: 25.444 sec
Best score found = 6

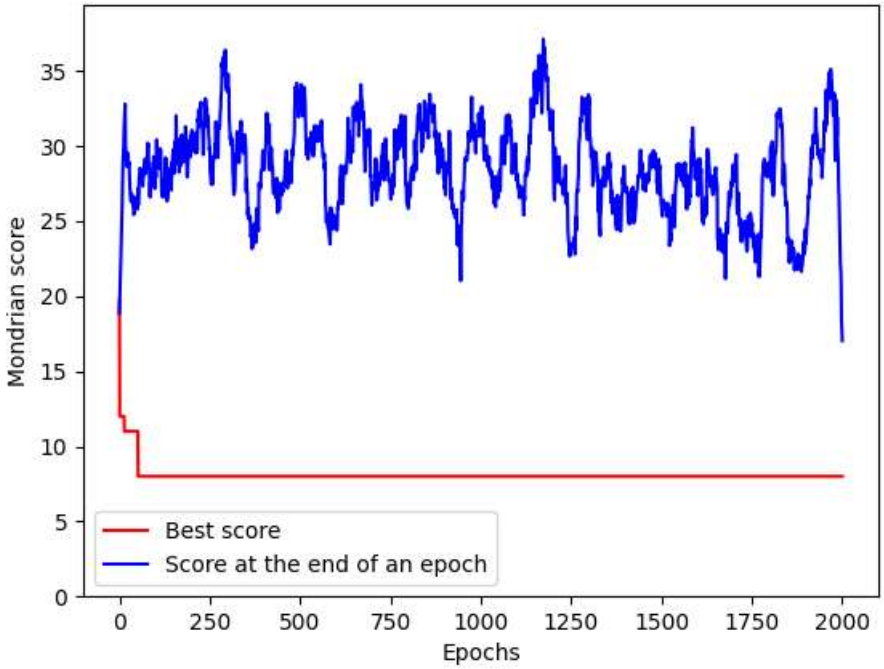


As can be seen from the above graph the best score is found relatively quickly through exploration but the q-learning algorithm (blue line - a 30-long running mean) seems to be learning nothing. The sharp drop-off at the end of the blue graph is a distortion caused by the convolution algorithm used to compute the running mean

Grid size 12



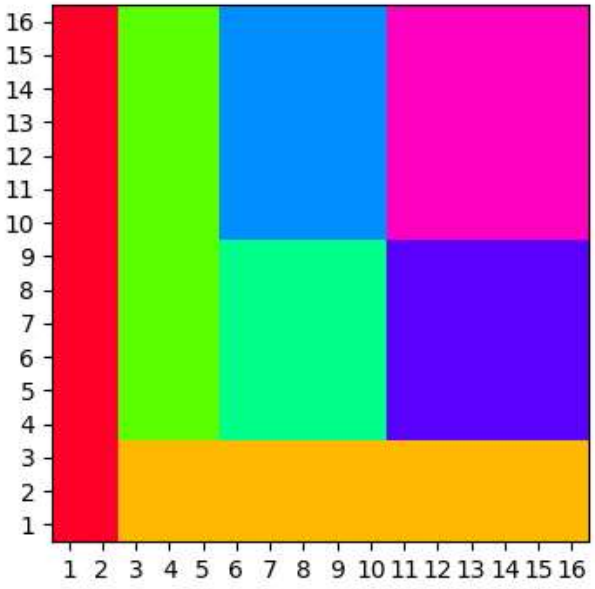
Time: 34.084 sec
Best score found: 8

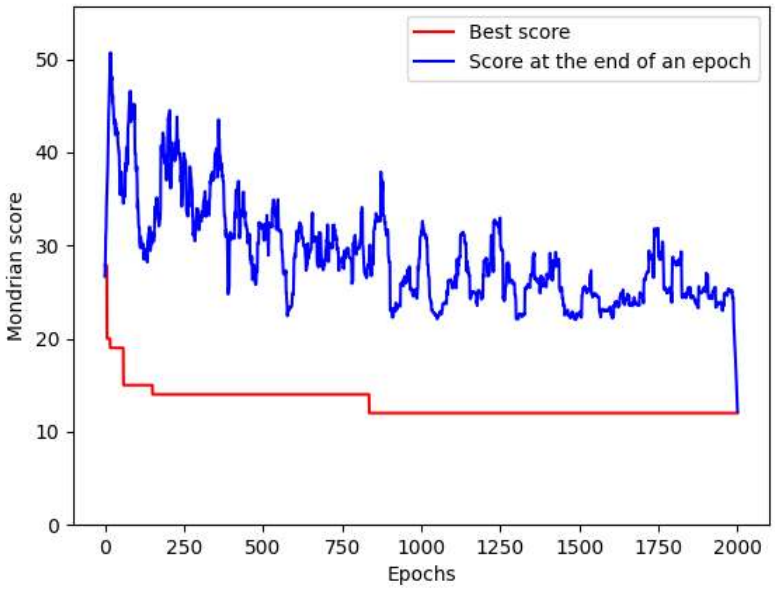


Grid size 16

Time : 35.435 sec

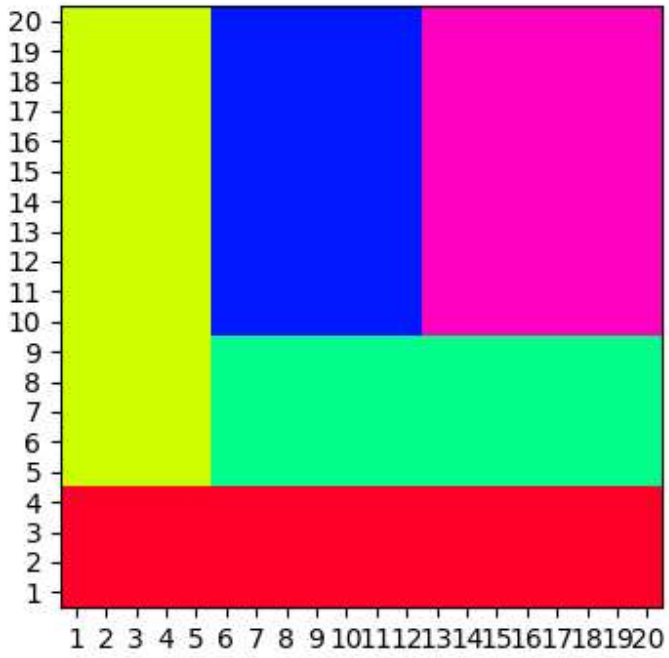
Best score found: 12



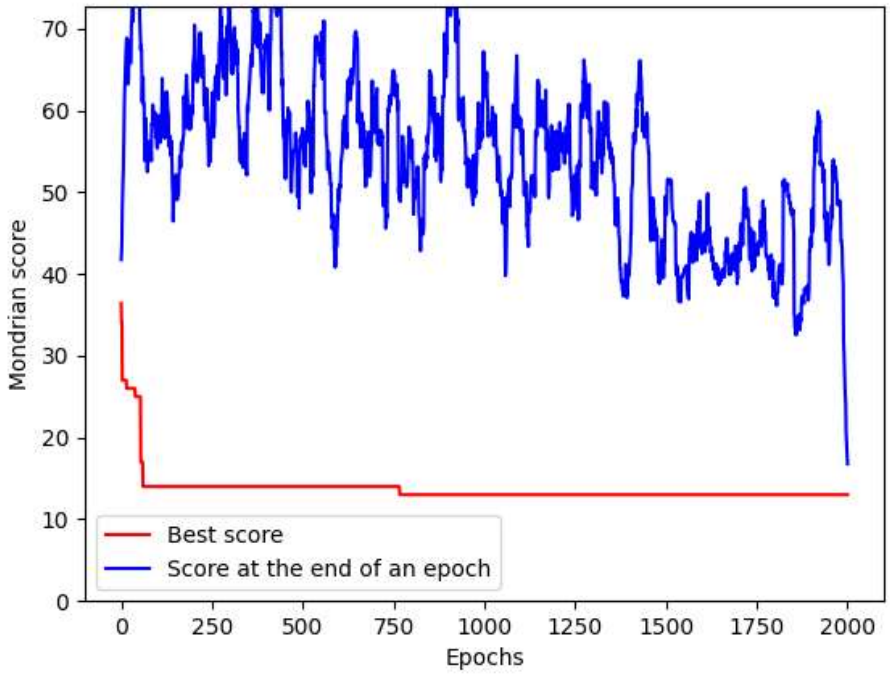


Here there is some trend visible in the blue line - a result of adjusting some parameters including the rewards.

Grid size 20



Time: 40.00 sec
Best score found: 13



4. Thinking about this now, rather than just trying to quickly write some code before the deadline, it seems embarrassingly obvious to me why this second approach was also always unlikely to work well.

First, the actions the q-learning algorithm uses often lead to highly unpredictable states (there is a *random* split and merge). For the algorithm to be able to learn more effectively a more detailed set of actions and states would have to be presented to it (split this rectangle in this ratio etc) or a better, much smarter way to reduce the search space and/or bundle some possible actions together would have to be found.

Second, even if the algorithm learned better, it would not necessarily find better solutions. To use an analogy of a landscape where small villages, representing solutions, are dotted around thick forests: it is not clear that cutting down trees to create a road (as the q-table is supposed to do here) to a village, after explorers stumble on it wandering aimlessly through the vegetation, will make it any easier for the explorers to find any other villages. This is especially true if the explorers cannot cross rivers or any other obstacles (invalid states are not allowed). In fact it is not impossible that some optimal solutions can never be reached by any sequence of merges and splits as they are defined here. However, at least when considering comparisons between some sets of parameter values the q-table didn't seem to be *completely* useless (Figure 3). I speculated that in so far as the q-table is improving the search at all, it is doing so not by guiding the search towards good solutions but by keeping it *away* from very bad states. To test this theory I observed the behaviour of the algorithm when the reward function was changed from an exponential, used in all experiments up until this point (Figure 1), that assigned a very high reward to good states to a log function which instead heavily penalised bad states (Figure 2). Unfortunately, this resulted in no improvement and the algorithm only became more prone to getting stuck on suboptimal states (Figure 4).

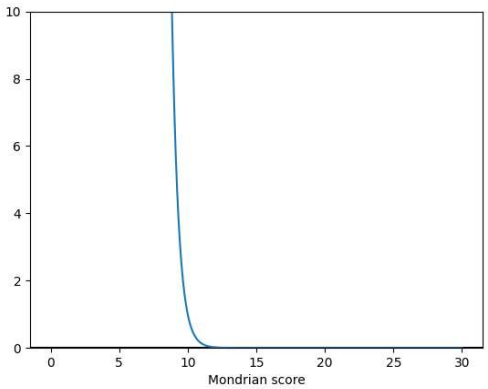


Figure 1

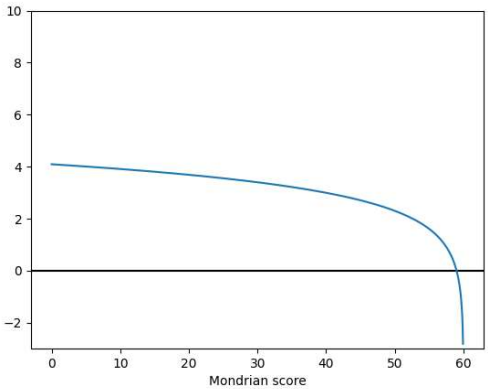
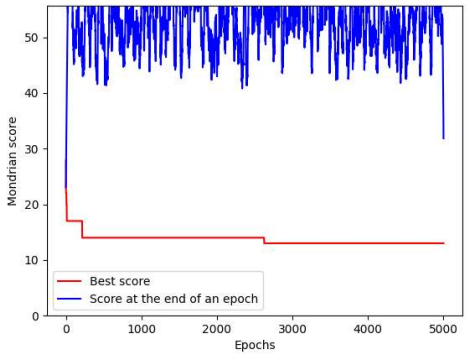


Figure 2

Two different reward functions used for a grid size 16

Grid size 16



Grid size 20

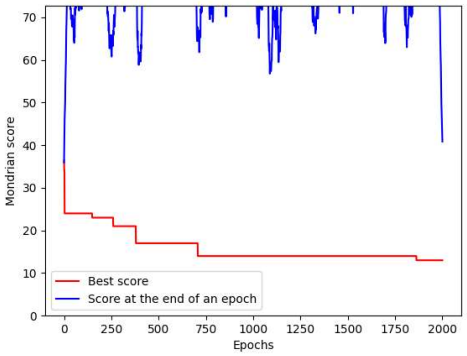


Figure 2

Two plots of search with a random action taken at every step and the q-table ignored. The blue line shows sustained high values with little trend - showing that the algorithm is of course not learning anything. More importantly however, the red line seems to take longer (5k epochs on the left) to converge (and converges to higher values) than on earlier plots. The results were not unequivocal but switching the q-table off did seem to generally result in higher Mondrian scores by 1-3 points for the same number of epochs and in a less steep convergence; at least for grids with size 16 upwards.

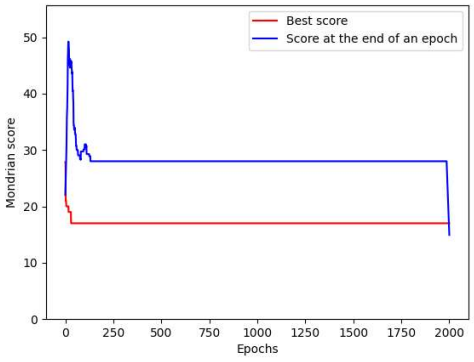


Figure 4

Algorithm stuck on a sub-optimal solution with a log reward scaling.

5. This would be trivial to implement. I would only need to change the dimensions of the grid as it is initialised (and propagate those changes to other classes e.g. the q-learning class itself which also uses the attribute “grid size” - now it would be a pair of ints not just one).

In short this is because my initial state is implemented as the Grid having only one rectangle (a square) filling its whole space. This initial, one rectangle is treated as any other rectangle (a square or not) on the grid and therefore it would not make any difference what its dimensions are.