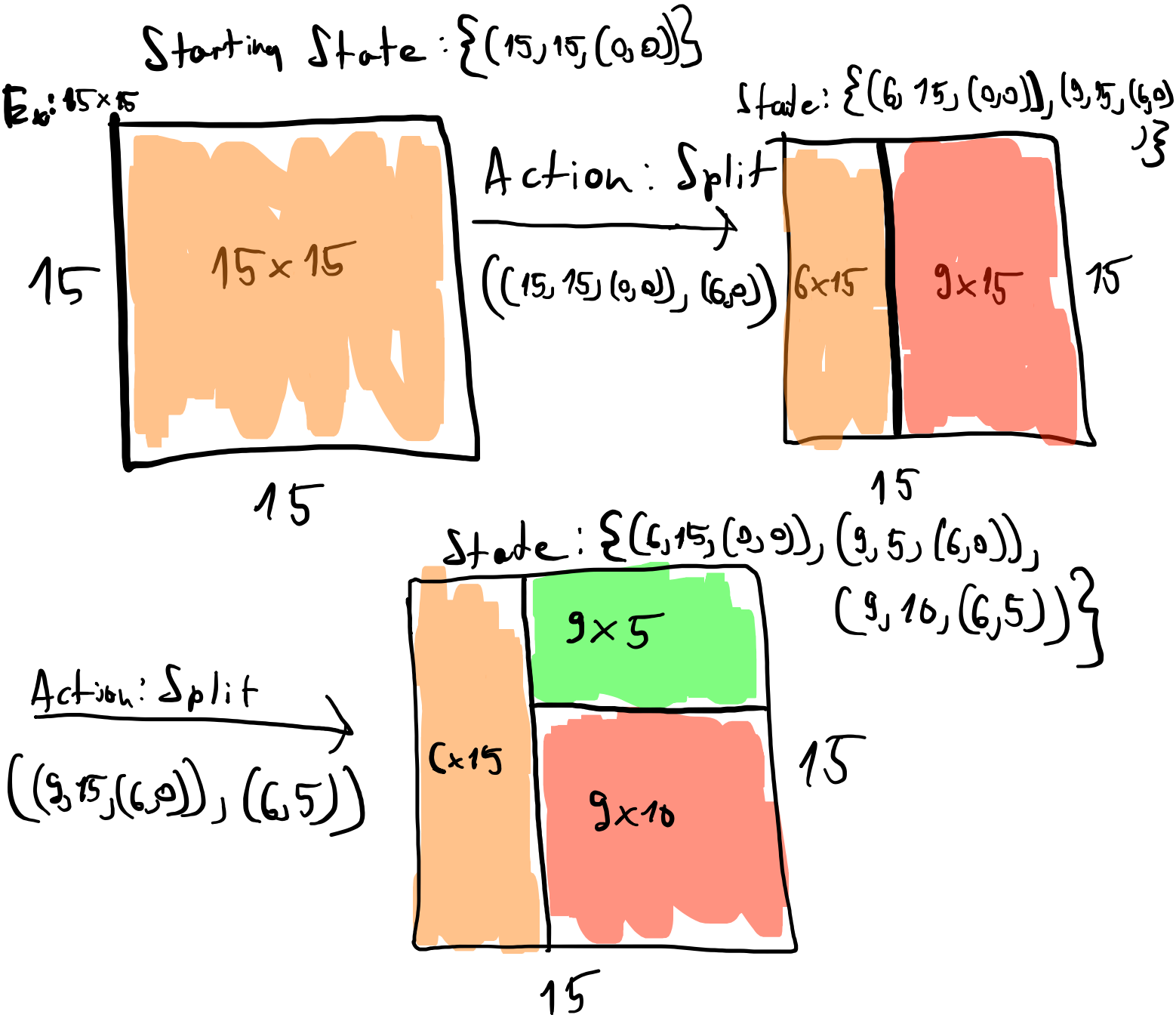
Report

1.2.1

The starting state involves having a single tile of equivalent dimensions to the given square. New states are created by splitting a tile into two new rectangular tiles (each tile can be split in many ways). States are represented by a list, where each value is a tile, which are defined by their dimensions and positions in the square. A tile is therefore described in the form: (*horizontal size, vertical size, location of top left 1x1 square*). The set of possible actions from a given state is all the ways in which all the tiles can be split. Therefore, a split can be executed given a tile in the current state and a position to split along (split(*tile, top left 1x1 square of new tile*).



1.2.2

Split can take illegal positions for the new tile if the given coordinate is not on one of the edges of the original tile, resulting in an illegal state, where a tile might extend outside the bounds of the square or overlap other tiles (split cannot be done). Even if the split position is on one of the edges of the original tile, the new tile may have a side with 0 length, which also should not be allowed. Additionally, actions can also create tiles with dimensions that already exist in the state.

1. Illegal state check pseudo code (assuming input state is legal):

Input:

Action (split (tile (horizontal size, vertical size, coord of top left 1x1 square), coord of top left 1x1 square of new tile))

current_state (list of tiles)

Output: Boolean (True if state is valid, False if not)

```
Def stateCheck (Action a, State s):
        //check split position
        If (a.tile.coord[0] == a.coord[0] ):
                //x coordinates match, so vertical split
                If(a.tile.coord[1]+a.tile.vertical_size < a.coord[1] OR a.coord[1] < a.tile.coord[1]):
                        //checking if vertical coordinate is within the tile
                        Return False
        Elif (a.tile.coord[1] == a.coord[1]):
                //y coordinates match, so horizontal split
                If(a.tile.coord[0]+a.tile.horizontal_size > a.coord[0] OR a.coord[0] < a.tile.coord[0]):
                        //checking if horizontal coordinate is within the tile
                        Return False
        Else:
                //Neither coordinates match, so illegal split
                Return False
        //generate new state and check for duplicates
        New_state = s.apply(a)
        For t1 in new_state:
                For t2 from (t1+1) in new_state:
                        If ((t1.horizontal, t1.vertical) == (t2.horizontal, t2.vertical) OR (t1.vertical,
t1.horizontal) == (t2.horizontal, t2.vertical)):
                                Return False

        Return True
```

2.

Splitting a tile with illegal given coordinates is not feasible. This is because it results in one of the new tiles being L-shaped as opposed to rectangular or overlapping tiles, both of which my state design is unable to describe, causing the algorithm to fall apart.

The only illegal states I would allow transitions to are ones where at least a pair of tiles are congruent, as these do not involve tiles with non-rectangular shapes, so they can be correctly described with the given state design. Allowing this is helpful because it may allow the search method to converge more quickly. Introducing these "transition" states give the algorithm more options, especially in the beginning, where two (relatively large) congruent tiles will be further split anyway, so there is no reason they should not be allowed. However, this results in having way too many states, making the program very slow, therefore only fully legal states are considered. This means that my approach cannot get to some optimal solutions, however it can get quite close more quickly.

1.2.3

(Assuming input state is legal)

Inputs: current_state (list of tiles), size of square (Int)

Output: Integer Mondrian score

```
Def Mondrian (state s, int a):
        Int min = a*a
        Int max = 0
        For tile in s:
                Int area = tile.horizontal * tile.vertical
                If (area < min):
                        Min = area
                If (area > max):
                        Max = area

//check only in place for starting states, which would equate to zero otherwise
If (max – min == 0)
                Return max – min
        Else
                Return a
```

1.2.4

The state-space search method I have decided to implement is recursive best-first search (RBFS). I chose an informed algorithm, as the Mondrian score of the current state can be used to estimate the distance from the best solution, so this will provide improved performance over uninformed options. RBFS is a good choice, since it is complete and optimal (given an appropriate heuristic function), returning the best reachable solution once the limiting condition *M* is reached (fits scenario as the specific goal state is unknown). It solves the main issue with normal best-first search, which is

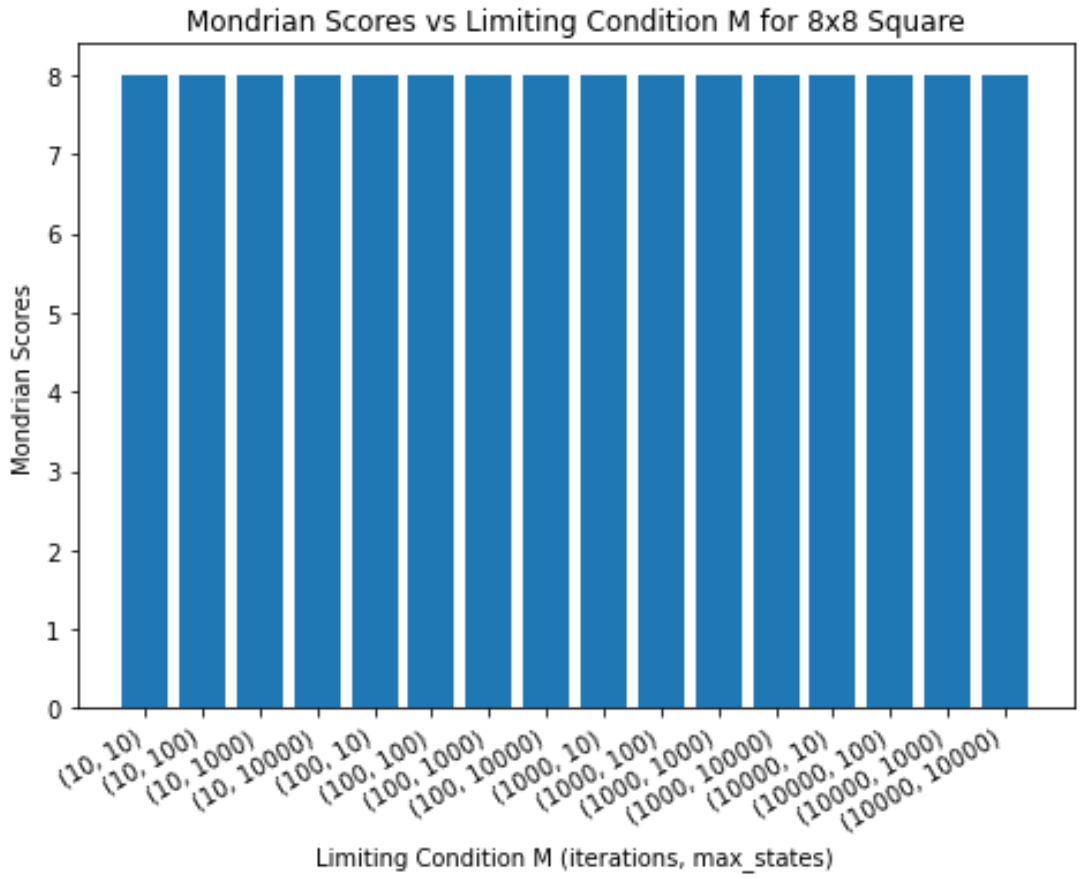
getting stuck in a suboptimal subtree, as it can unwind the tree. This means that the algorithm can move back to states with less tiles, essentially acting as an action joining states. Therefore, there is no need to have a dedicated join action, decreasing the number of actions which need to be considered each pass.

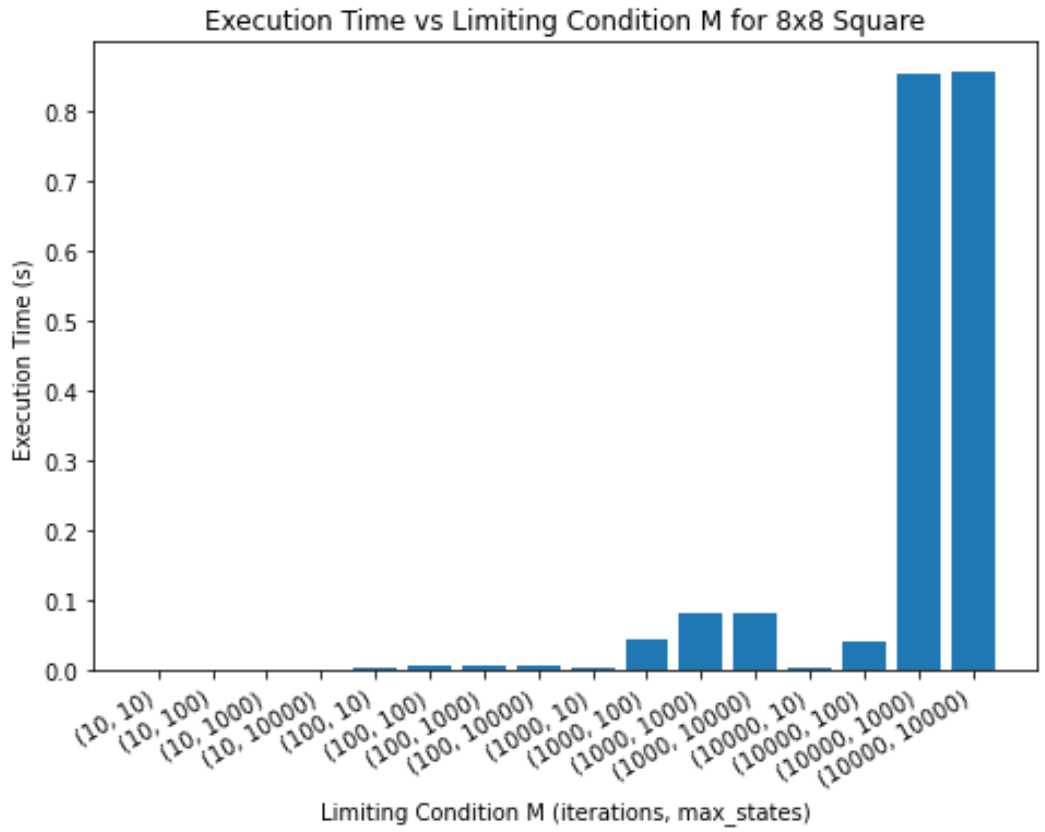
1.

The immediate drawback for RBFS is that it must keep track of all alternate states, which is a list that gets extremely large as the tree is traversed. This is addressed by the limiting condition *M*, which contains *iterations* (number of iterations of the algorithm / number of nodes visited) and max_states(maximum number of best alternate states tracked, finishing execution once no more states to explore). *max_states* makes it so that less options are considered each pass, speeding up execution.

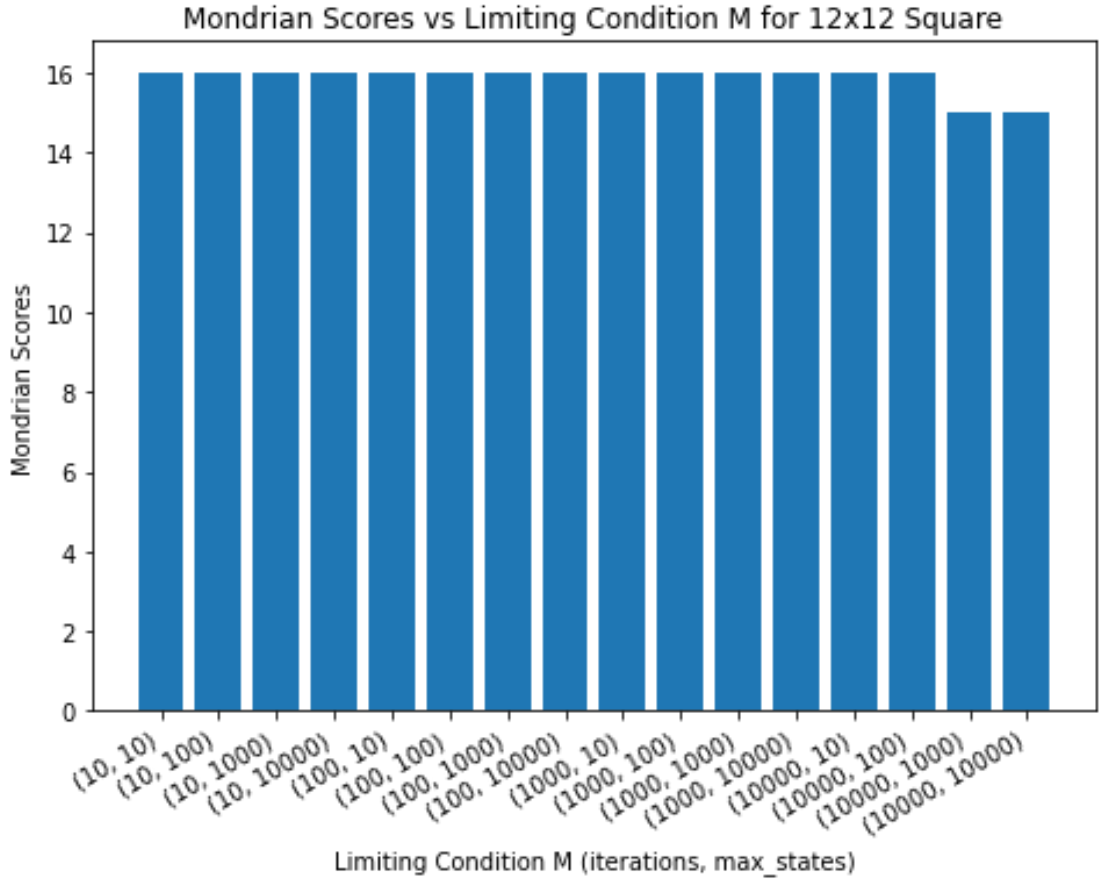
2.

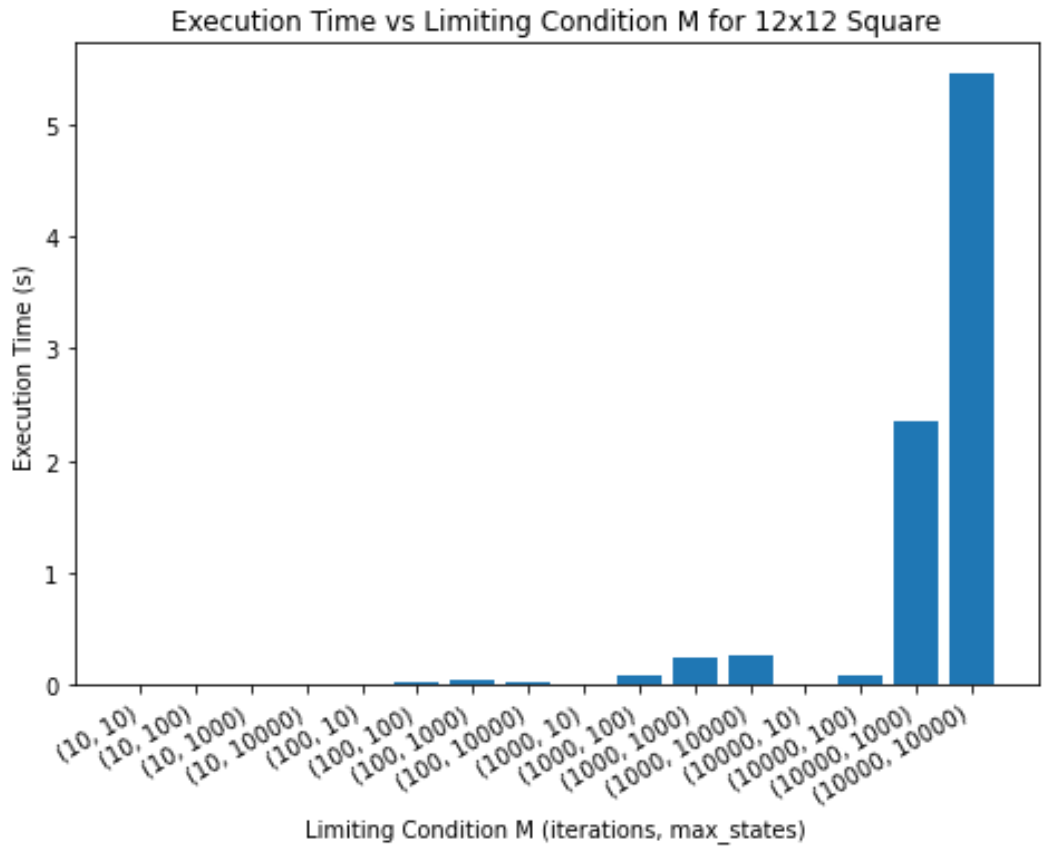
8: σ = 8; optimal= 6



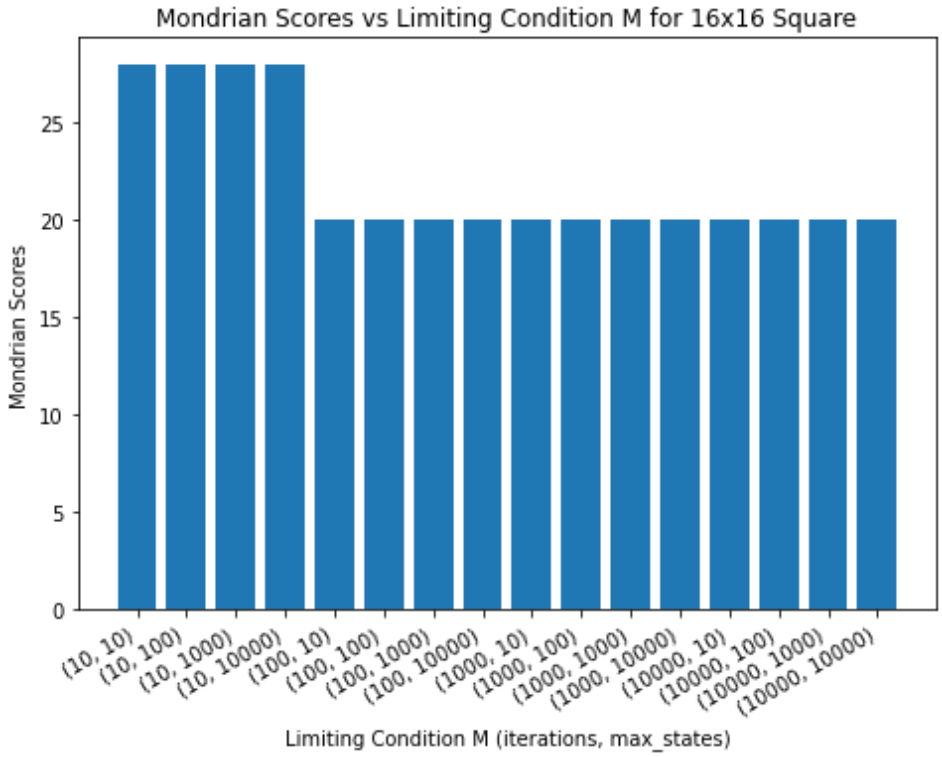


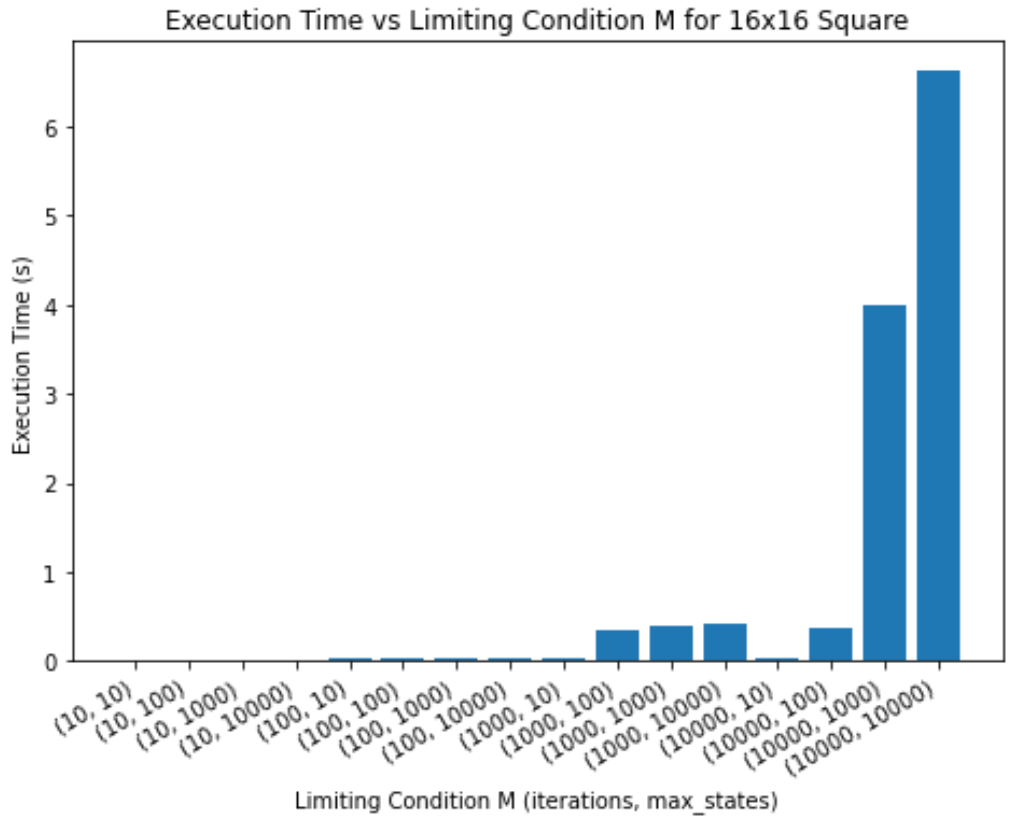
12: σ = 15 optimal = 7



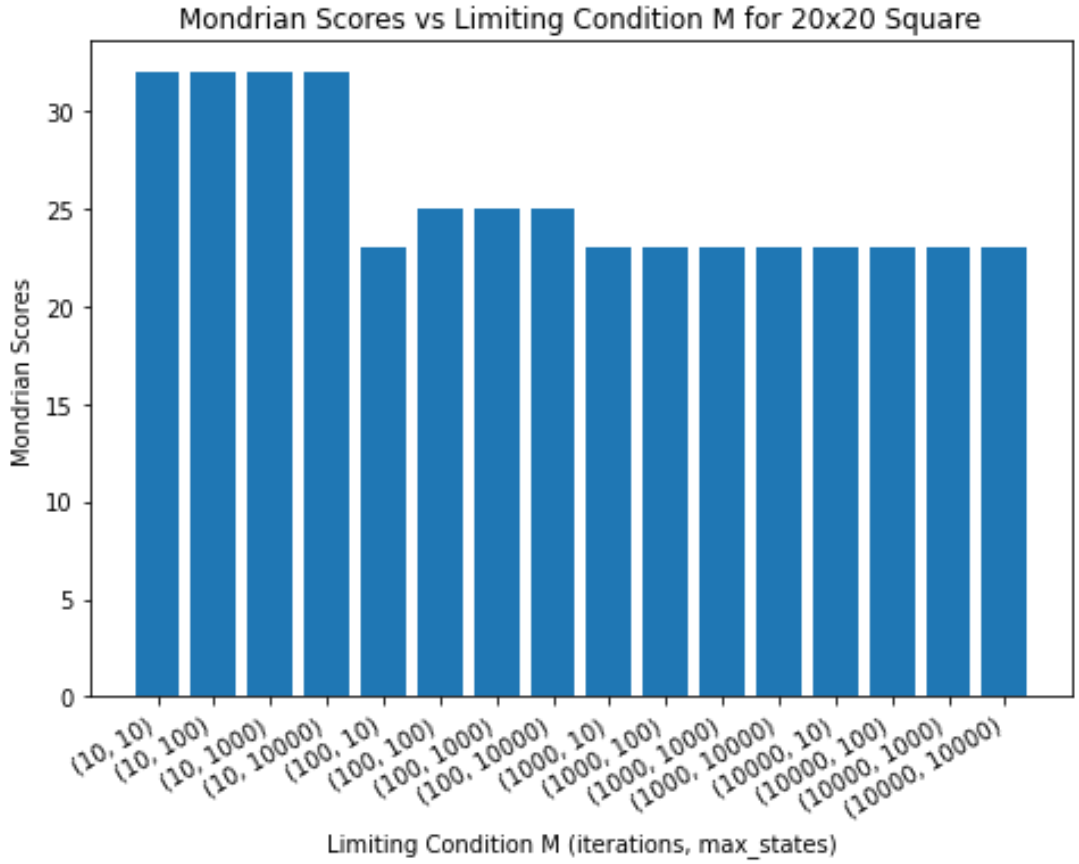


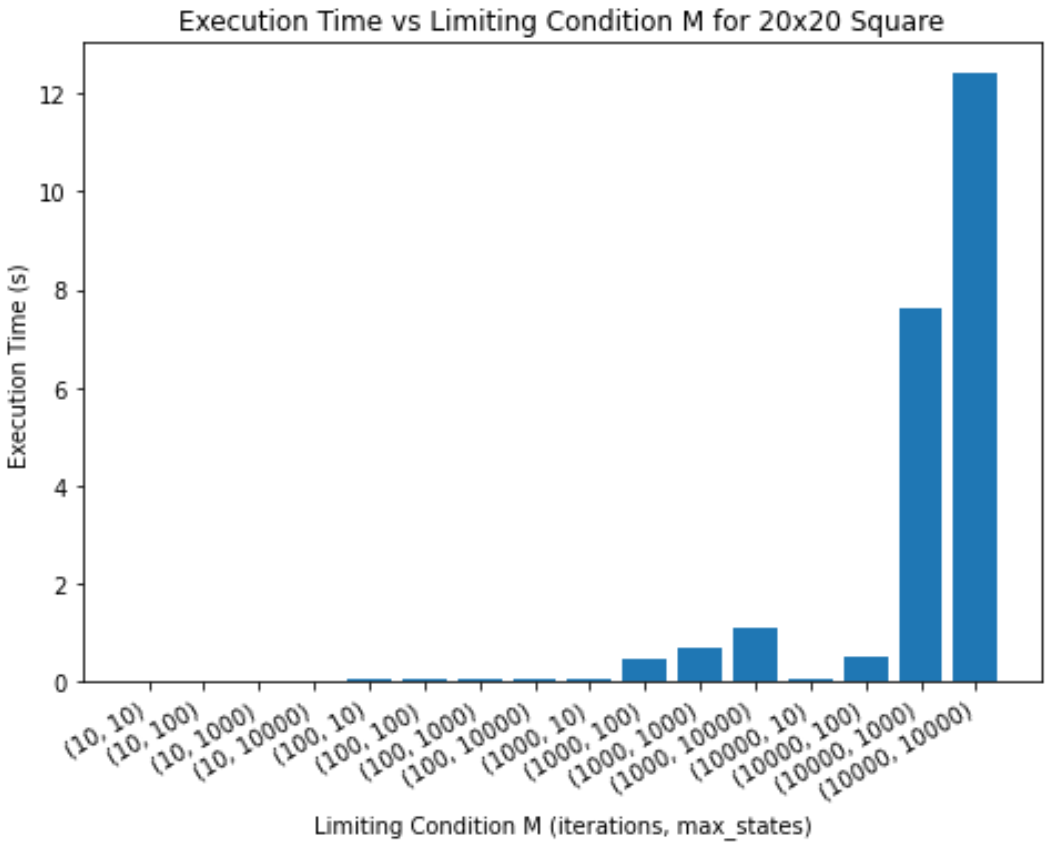
16: σ = 20 optimal = 8





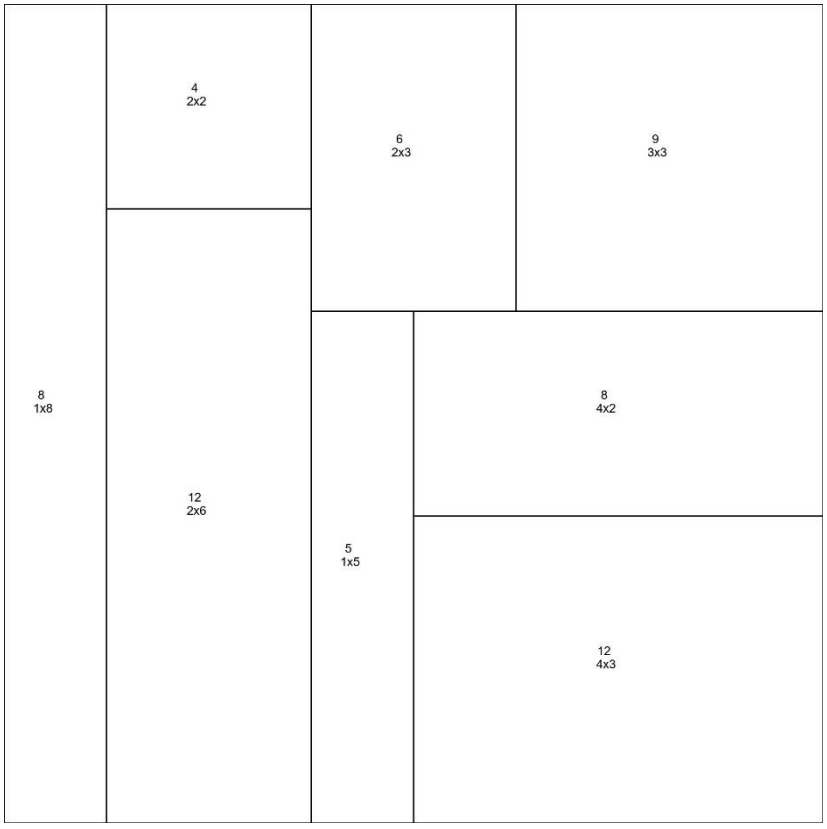
20: σ = 23 optimal = 9



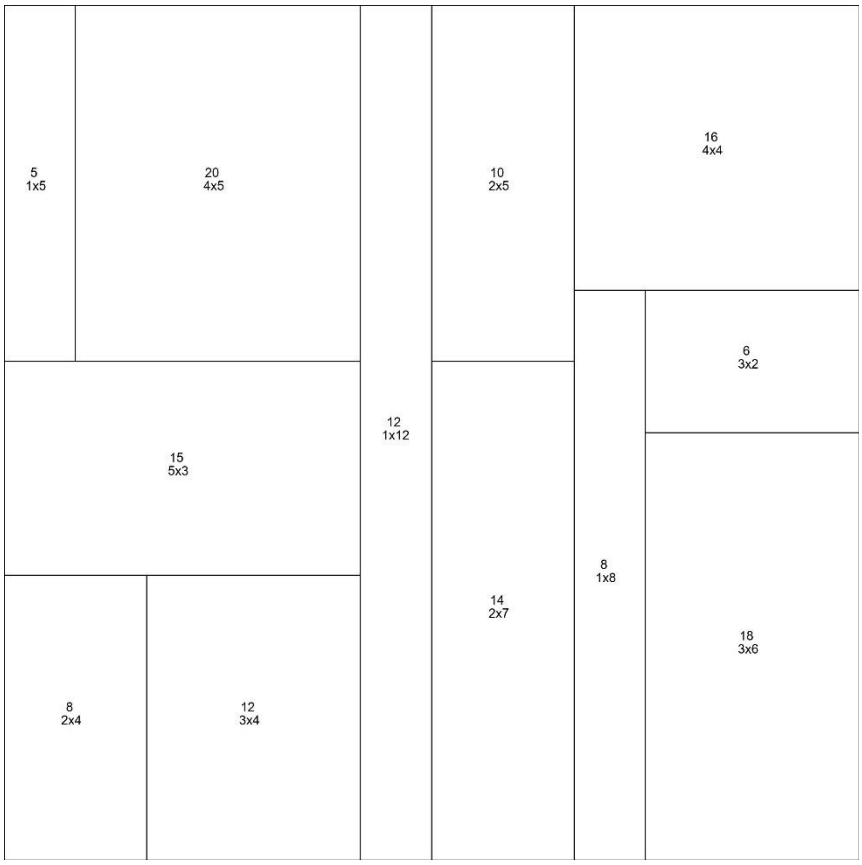

Execution Time vs Limiting Condition M for 20x20 Square
Execution Time (s)
0
2
4
6
8
10
12
(10, 10)
(10, 100)
(10, 1000)
(10, 10000)
(100, 10)
(100, 100)
(100, 1000)
(100, 10000)
(1000, 10)
(1000, 100)
(1000, 1000)
(1000, 10000)
(10000, 10)
(10000, 100)
(10000, 1000)
(10000, 10000)
Limiting Condition M (iterations, max_states)

3.

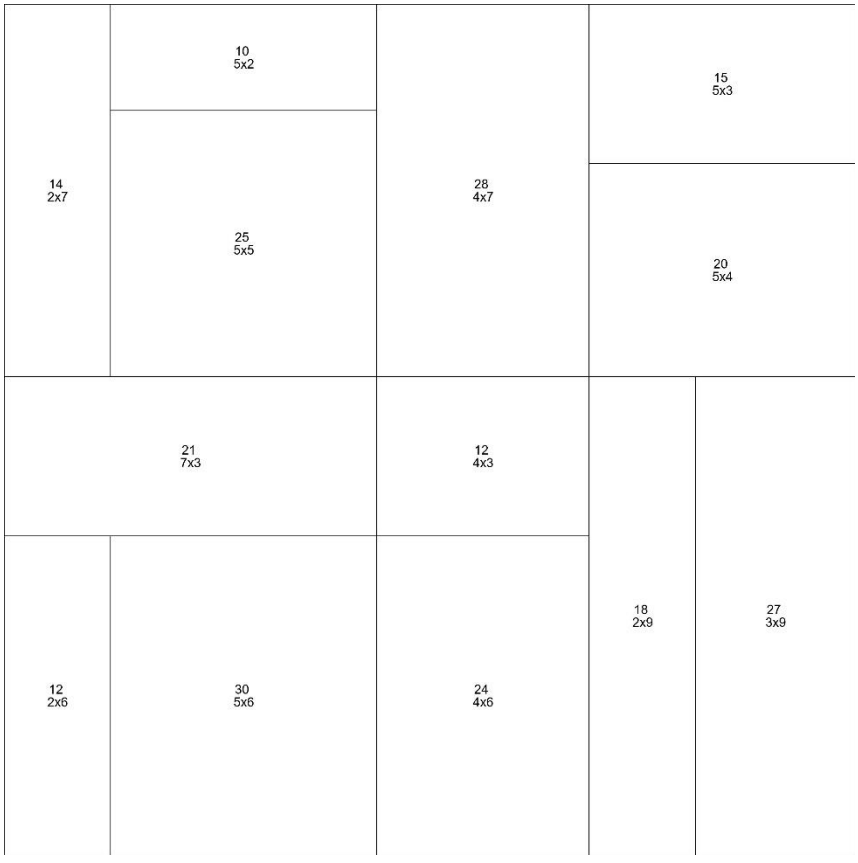
8:


4
2x2
6
2x3
9
3x3
8
1x8
8
4x2
12
2x6
5
1x5
12
4x3

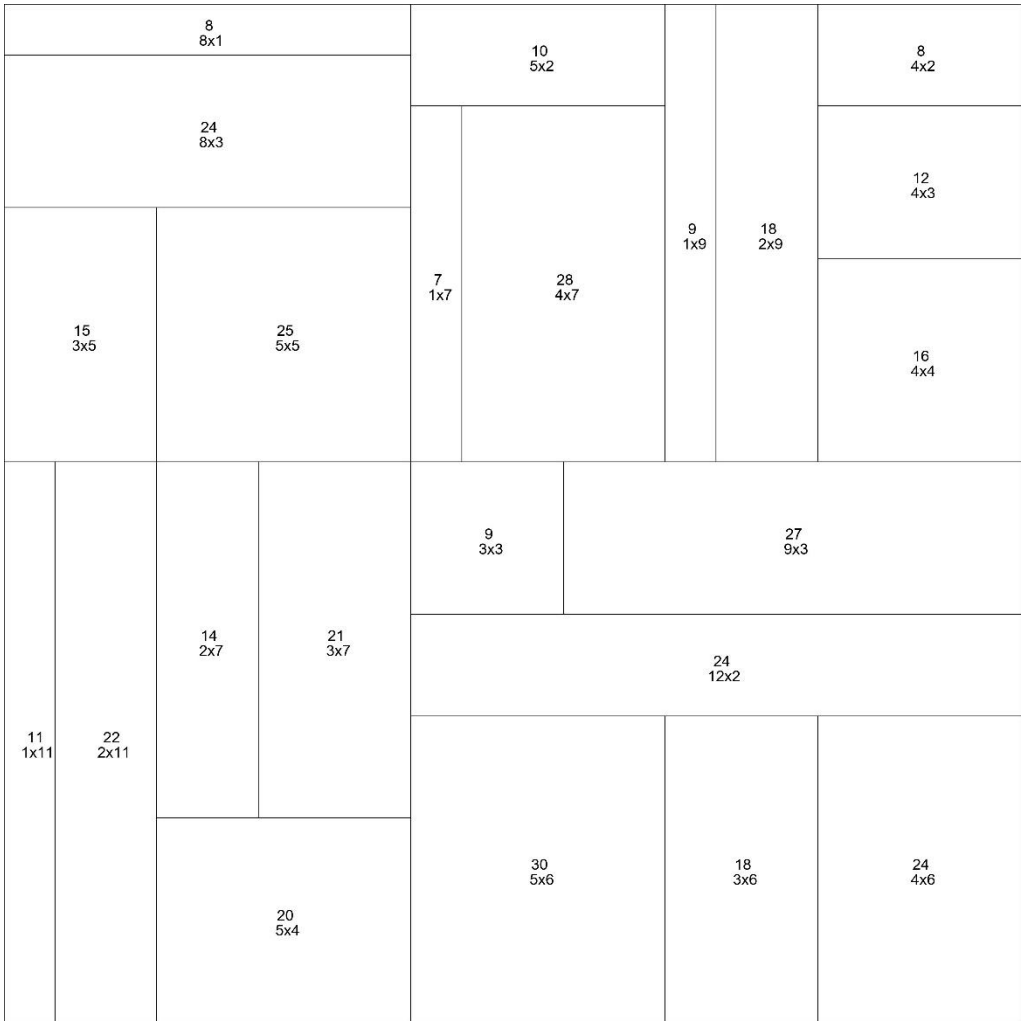
12:


5
1x5
20
4x5
12
1x12
10
2x5
16
4x4
6
3x2
15
5x3
8
1x8
14
2x7
18
3x6
8
2x4
12
3x4

16:


10
5x2
15
5x3
14
2x7
28
4x7
25
5x5
20
5x4
21
7x3
12
4x3
18
2x9
27
3x9
12
2x6
30
5x6
24
4x6

20:



4.

As a general trend, it is safe to say that as the limiting parameters increase, the Mondrian score for any size square decreases, at the cost of execution time, which greatly increases. This makes sense, as allowing the algorithm to explore more nodes (*iterations*) or track more possible follow-up states (*max_states*) should result in better estimations, at the cost of efficiency.

One outlier for this statement is the 20x20 square, as this is the only example where having a lower *max_states* leads to a lower Mondrian score (at *(100, 10)*). This is likely because at *(100, 100)* and *(100, 1000)* the increased number of *max_states* floods the algorithm with transition states of equal Mondrian scores, causing it to explore the wrong subtrees first.

After 10000 *iterations*, the execution time starts skyrocketing, without finding any more optimal states. It is possible that some better scores could be found at higher *iterations*, however it is also known that some most optimal states (4x4 for example), cannot be found by this algorithm as it does not allow transitions to illegal states (a split where the two tiles created have the same dimensions).

5.

First, the solve_mondrian function must be altered to take the two sides of the rectangle as separate inputs (a, b instead of just a). From there, the starting state would be ([(a, b, (0, 0))] a*b). Then, the split function must be changed, also to take a and b instead of just a. Furthermore, the split function must be changed so that it does not only explore splits up to the halfway point of a tile, since this only works for squares due to their symmetrical nature. Once the Mondrian function is altered as well, assigning a * b to minimum, the program should work for axb rectangles.

These changes are quite trivial, since the search method simply splits rectangular tiles into two, creating new states. Therefore, changing the overall shape of the outer rectangle would cause no issue, as only individual tiles are ever traversed, which are rectangular to begin with.