Effectiveness of A* Algorithm for Constrained Tiling: A Particular Case-Study with the Mondrian Tiling Problem

A Thesis to be Submitted

in Partial Fulfilment of the Requirements

for the Degree of

Master of Technology

by

RISHI DEY

Roll No : CS1905

under the supervision of

Dr. Debasis Ganguly

School of Computing, University of Glasgow

Glasgow, UK

Dr. Mandar Mitra

Computer Vision and Pattern Recognition Unit,

Indian Statistical Institute

Kolkata, India

Indian Statistical Institute, Kolkata

Kolkata - 700108, India

Certificate

This is to certify that the dissertation entitled **"Effectiveness of A* Algorithm for Constrained Tiling: A Particular Case-Study with the Mondrian Tiling Problem"** submitted by **Rishi Dey** to Indian Statistical Institute, Kolkata, in partial fulfillment for the award of the degree of **Master of Technology in Computer Science** is a bonafide record of work carried out by him under my supervision and guidance. The dissertation has fulfilled all the requirements as per the regulations of this institute and, in my opinion, has reached the standard needed for submission.

12/07/2021

.....

Debasis Ganguly

School of Computing, University of Glasgow,

Glasgow, UK

Mandar Mitra 12.07.2021

.....

Mandar Mitra

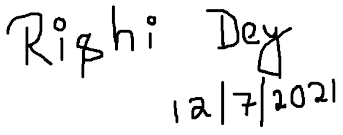
Computer Vision and Pattern Recognition Unit,

Indian Statistical Institute, Kolkata

Declaration

I hereby declare that this thesis report titled "Effectiveness of A* Algorithm for Constrained Tiling: A Particular Case-Study with the Mondrian Tiling Problem" is my own original work carried out as a postgraduate student at Indian Statistical Institute, Kolkata, except the assistance from other sources which is duly acknowledged.

All sources used for this project report have been fully and properly cited. It contains no material which to a substantial extent, has been submitted for the award of any degree/diploma in any institute or has been published in any form, except where due acknowledgment is made.



.....

Rishi Dey

Roll - CS1905

Master of Technology in Computer Science,

Indian Statistical Institute, Kolkata

Dedication

To Maa.....

Acknowledgements

I would like to express my sincere gratitude to my guide **Dr. Debasis Ganguly** for allowing me to work under him and providing me exciting problems to think upon. Throughout the development of this thesis, he was always available for discussion and guidance and has provided support, patience and optimism. I would also like to thank **Dr. Mandar Mitra** for providing me valuable suggestions and feedback. I will be failing in my duties if I don't mention friends and people who mattered and conversing with whom did help me during two years at ISI Kolkata. In this period, I have been enriched and nurtured in numerous ways by interacting with many people.

Finally, I would like to express my special gratitude to my first teacher my **Maa**, who has always encouraged me in all aspects for igniting the love for knowledge within me.

Abstract

The 'Mondrian Tiling' problem is a particular class of constraint optimization problem where a square grid is covered with some non-overlapping integer dimension rectangles, which must not be pairwise congruent. The objective is to use rectangles with similar areas so that difference between the largest and smallest rectangle area, known as score, is minimum. A brute-force approach towards solving this problem enumerates all possible solutions to find the optimal one and incorporates two prevalent NP-Complete problems, making it an exponential algorithm and computationally challenging to compute for large grids. In contrast, our proposed approach employs grids as states and applies a number of (specifically, 4) state transformation operations to improve the states in terms of score. The state-space representation is utilised to explore the states with some strategy to obtain the optimal one. A number of restrictions are applied with the purpose of obtaining a balance between exploration and exploitation of the state-space. The results of our experiments exhibit that the recommended approach is profoundly efficient compared to the former approach, and the obtained scores are close. In contrast to the brute force approach, the state-space search approach can lead to feasible solutions within a relatively small amount of run-time for large grid sizes. It can be deemed as a quick way to provide information about the position of the optimal score.

Contents

List of Figures

List of Tables

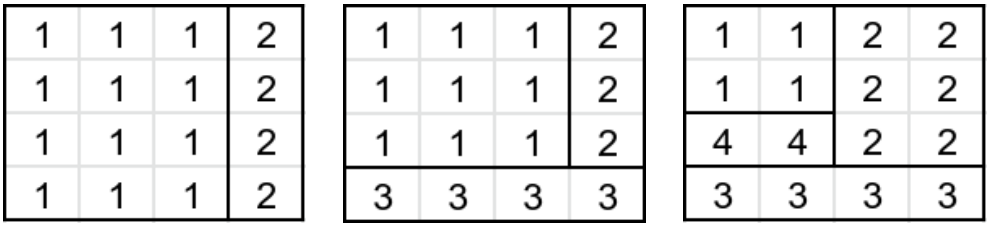
Chapter 1

Introduction

The *Mondrian Tiling puzzle* [6] is a problem based on the artwork of the Dutch artist Piet Mondrian, who is known for the common use of rectangular shapes in his works. Quoting O'Kuhn [14]:

> The idea of the puzzle is that an art critic has ordered Mondrian only to create paintings whose rectangles are all incongruent to one another and only have integer side lengths. Furthermore, he can only use a square canvas whose side length is also an integer. Aggravated, Mondrian still wants to create works whose areas of the rectangles are all as close as possible.

In simple terms, consider an equally spaced two-dimensional grid of size $n \times n$. The whole area of the square grid needs to be filled (or *tiled*) with non-overlapping rectangles with integer dimensions. Moreover, once a rectangle of dimensions $a \times b$ is employed, any other rectangle with dimensions $a \times b$ or $b \times a$ cannot be used. So all the rectangles have to be pairwise non-congruent. After the entire grid area is filled up by different rectangles, a *score* (or *sometimes referred to as defect*) is computed by inspecting the area difference between the largest and smallest rectangles used. This score is also known as the *Mondrian score*. In general the Mondrian score will be different for different tilings of the same $n \times n$ grid. The objective is to find a tiling that has the smallest possible Mondrian score for an $n \times n$ grid.



(a) Score = 8 (b) Score = 6 (c) Score = 4

Figure 1.1: Different configurations of the 4×4 grid with Mondrian scores.

For example, different tilings of a 4×4 grid are illustrated in Figure 1.1. The rectangles used for each tiling are marked with some integer numbers. These integer numbers neither specify the area nor the dimension of rectangles but are used for detection purposes. If k rectangles are used for a tiling, then each one is labelled with $1, 2, \ldots, k$. In Figure 1.1a, the grid is covered using only two rectangles, marked with 1 and 2. For this tiling, the score is $(4 * 3) - (4 * 1) = 8$. This score is improved in Figure 1.1b. Here, three rectangles are used, which are pairwise non-congruent. The largest rectangle is 3×3 and the smallest one is 3×1. Thus, the Mondrian score of this configuration is $(3 * 3) - (3 * 1) = 6$.

Lastly, Figure 1.1c depicts that how four non-overlapping, non-congruent rectangles may be used to cover the grid. The largest rectangle has an area of 6, and the smallest rectangle has an area of 2. So for this tiling the score is $6 - 2 = 4$. This happens to be the optimal Mondrian score for a 4×4 grid.

One important point to note is that a Mondrian tiling exists for all positive integer grid sizes. If n is odd, then two rectangles of dimensions $\frac{(n-1)}{2} \times n$, $\frac{(n+1)}{2} \times n$ can be used to cover the $n \times n$ grid. On the other hand, if n is even, then two rectangles of dimensions $(\frac{n}{2} - 1) \times n$, $(\frac{n}{2} + 1) \times n$ can be used to cover the grid. From these tilings, the following obvious upper bounds on Mondrian score are obtained for given a $n \times n$ grid:

$$\text{Trivial Mondrian score} = \begin{cases} n, & \text{if } n \text{ odd and } n \in \mathbb{Z}^+ \\ 2n, & \text{if } n \text{ even and } n \in \mathbb{Z}^+ \end{cases}$$

A naive computation of the optimal score would require considering all possible valid

configurations of a filled-up grid, which is infeasible from a computational perspective. Indeed, computing a *single* valid tiling (which involves selecting some rectangles such that they cover the grid, are non-overlapping as well as non-congruent) is a challenging task itself.

The currently known approach is a brute force algorithm. It is divided into two parts. The first part tries to find some rectangles in terms of partitions such that the rectangles can cover the entire grid, i.e., the sum of rectangle areas is the same as the grid area. This part is not concerned about whether all the rectangles can fit inside the grid together; it merely generates multiple partitions in an algebraic sense. Next, these algebraic partitions need to be converted into a geometric arrangements of rectangles that fit inside the grid. The second part of the algorithm is responsible for this task, and checks if a partition can be accommodated inside the grid while maintaining the core constraints. If yes, then the score is computed. Unfortunately, both parts of the problem are NP-complete. Current deterministic implementation of the algorithm requires an exponential amount of time to produce results. A detailed discussion of the above algorithm can be found in Section 5.2.

Due to its exponential nature, the existing algorithm consumes much time to provide the optimal score for a given $n \times n$ grid. Moreover, it can not even determine any non-trivial Mondrian score for large grid size; the optimal score is far away.

1.1 Research Questions

In this thesis, three research questions are addressed and investigated in the subsequent chapters. The questions are listed below:

RQ1: *Is a state-space search based approach more suitable for a constrained tiling task?*

RQ2: *What is the suitable set of transformation operations that could be useful for solving a constrained tiling task?*

RQ3: *How does the effectiveness-efficiency trade-off of a state-space search based*

approach compare to that of a brute-force approach?

1.2 Thesis Contributions

In this thesis, our proposed method attempts to alleviate the mentioned shortcomings of the existing approach. It employs a state-space representation where each state is a valid tiling with a score and satisfies the primary constraints of the problem. It then tries to improve the score by locating some grid arrangement with a better score through some geometric transformation operations. Our experiments show that the proposed method can find valid Mondrian scores for large grids within a stipulated period. A thorough investigation of the above algorithm is presented in Section 4.

Chapter 2

Previous Work

Not much work has been done on the Mondrian Art Problem in recent times. It is a relatively unexplored problem. Most of the work on this problem has been conducted by the discrete mathematics community. To the best of out knowledge no empirical analysis has been carried out yet. Some of the related work done on the problem listed below:

- O'Kuhn [14] finds the density of numbers n that satisfy Mondrian score $d(n) \neq 0$ in a given range $[1, x]$ and obtains a nontrivial lower bound for the cardinality of the set $\{n \leq x : d(n) \neq 0\}$. He proves that for all $x \geq 3$ and all $\epsilon > 0$, the following condition holds good:

$$|\{n \leq x : d(n) \neq 0\}| \geq \frac{C_\epsilon \log(\log(x))}{\log(x)} \left(1 + O_\epsilon\Big(\frac{\log(\log(x))}{\log(x)}\Big)\right)$$

  where

$$C_\epsilon = \frac{1}{e^\gamma(2\log(2) + \epsilon)}$$

  and γ is the *Euler-Mascheroni* constant.

- Dalfó et al. [3] compute some bounds on $d(n)$ in terms of the number of rectangles of the square partition. These bounds provide us optimal partitions for some values positive $n \in \mathbb{N}$. They provide sequence of square partitions such that $\frac{d(n)}{n}$ tends to zero for large values of n. Moreover, if $n(x)$ is the number of side lengths x (with $n \leq x$) of squares not having a perfect partition

(i.e., $d(n) = 0$), its density $\frac{n(x)}{x}$ is asymptotic to $\frac{(\log(\log x))^2}{2 \log x}$, which improves the bound given by O'Kuhn[14].

- Bassen [1] shows a computational method for obtaining the least possible Mondrian score. Bassen provides optimal solutions up to a grid size of $n = 32$. These solutions are optimal in terms of score and constructed using a minimal number of rectangles. The algorithm is discussed in Section 5.2 and its performance is compared to the proposed system with respect to speed and optimality, in Section 6.

Chapter 3

Prerequisites

In this chapter, a few famous problems and some techniques for solving them are discussed. These problems and techniques are used in the following chapters.

3.1 Subset Sum

The Subset Sum problem is a decision problem. The fundamental idea of this problem is that given a set S and a number T, it is checked if there exists any subset of S whose element sum adds up to T [9]. This problem is an NP-complete and can be considered as a particular case of another popular decision problem, Knapsack problem. For example, consider $S = \{1, 2, 3, 4, 5, 6\}$, $T = 10$. Then there exists the 5 subsets such that the element sum of each subset adds up to T.

$$\{1, 2, 3, 4\}, \{1, 3, 6\}, \{1, 4, 5\}, \{2, 3, 5\}, \{4, 6\}$$

But if $T = 22$, then there does not exist any such subset of S.

3.1.1 Exhaustive search approach

This algorithm first finds the power set of the set S. Then, for each subset, it checks if the sum of the elements in that subset is T. If $|S| = n$, this algorithm iterates through 2^n subsets. Finding the sum of each subset requires at most n additions. So, the running time of this naive algorithm is $O(n \cdot 2^n)$.

3.1.2 Dynamic programming approach

This problem can be solved using dynamic programming in pseudo-polynomial time. Consider $S = \{x_1, x_2, \ldots, x_n\}$ and assume the elements are sorted in ascending order. A new boolean array $P(i, t)$ is designed which stores TRUE if there is a nonempty subset of $x_1, x_2, \ldots, x_i$ which sums to t, and FALSE otherwise. The existence of any subset of S with sum as T can now be determined by checking $P(n, T)$.

Let A be the sum of the negative values and B the sum of the positive values of S. Clearly, $P(n, T) =$FALSE, if $T < A$ or $T > B$.

Initially, the array $P(i, t)$ can be assigned $P(1, t) \leftarrow (x_1 == t)$ for $A \leq t \leq B$.

Now, $P(i, t)$ has a clear recursive relation:

$$P(i,t) \leftarrow P(i-1,t) \text{ OR } P(i-1,t-x_i) \text{ OR } (x_i == t) \quad \text{for } 2 \leq i \leq n, A \leq t \leq B$$

If the array $P(i, t)$ is constructed for the above example, then it takes the form shown in Table 3.1.

P(i, t)		**t →**										
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
i ↓	**0**	T	F	F	F	F	F	F	F	F	F	F
	1	T	T	F	F	F	F	F	F	F	F	F
	2	T	T	T	T	F	F	F	F	F	F	F
	3	T	T	T	T	T	T	T	F	F	F	F
	4	T	T	T	T	T	T	T	T	T	T	T
	5	T	T	T	T	T	T	T	T	T	T	T
	6	T	T	T	T	T	T	T	T	T	T	T

Table 3.1: Sample array representation of subset sum problem. $P(i, t)$ indicates if there exists any subset of the first i elements, which sums to t.

Now, based on this $P(i, t)$ array, the subsets can be easily identified using the backtracking technique. First, all the values of $P(i, T)$ are checked for $1 \leq i \leq n$. If any particular $P(i, T)$ is TRUE, then $P(i, t)$'s are checked to further locate the subsets for $A \leq t \leq B$.

For example, first, the value $P(6, 10)$ is checked and found TRUE. So, there exists at least one subset with a sum of 10. There can be three cases: either the 6^{th} element belongs to any of such subsets or not. If not, then the value $P(5, 10)$ is checked and

so on. If the 6^{th} element is included, then the value at $P(5, 4)$ is examined. Since $P(5, 4)$ is TRUE, there is at least one subset, including the 6^{th} element.

Now, the 5^{th} and 6^{th} elements can not be part of the same subsets. So the $P(4, 10)$ is checked, and this process is repeated until the left edge of the array is reached. The paths that the above process follows for $P(6, 10)$ are indicated in Figure 3.1. The same method is repeated for $P(i, 10)$ where $1 \leq i \leq 6$

P(i, t)		**t →**										
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
i ↓	**0**	T	F	F	F	F	F	F	F	F	F	F
	1	T	T	F	F	F	F	F	F	F	F	F
	2	T	T	T	T	F	F	F	F	F	F	F
	3	T	T	T	T	T	T	T	F	F	F	F
	4	T	T	T	T	T	T	T	T	T	T	T
	5	T	T	T	T	T	T	T	T	T	T	T
	6	T	T	T	T	T	T	T	T	T	T	T

Figure 3.1: The paths which are traversed for finding the subsets including the 6^{th} element are marked.

Construction of the array takes $O(n \cdot T)$ time. On the other hand, finding the subsets consumers $O(n^2 \cdot T)$ since in worst case the entire array is traversed for each $P(i, T)$.

3.2 State space search

State space search is a process used in the field of artificial intelligence (AI), in which an agent with some specific strategy traverses different configurations or states of a problem. The objective of the agent is to find a *goal state* with some desired property.

A problem is often represented as a state space. A state space is a collection of states that a problem can be in. The state space of a problem can be considered as a graph where the states are the vertices and two states are connected if there is an operation or action that can transform the first state into the second.

A typical state space graph is much too large to generate and store in computer memory. Instead, nodes are generated as they are explored, and typically discarded

thereafter. The solution to a search problem is a sequence of actions that transform the start state to the goal state.

In state space search, a state space is formally represented as a tuple

$$S^* : \langle S, A, Action(s), Result(s, a), Cost(s, a) \rangle$$

in which:

- S is the set of all possible states;
- A is the set of possible actions, not related to a particular state but regarding all the state space;
- $Action(s)$ is a function that establishes which action is possible to perform in a certain state;
- $Result(s, a)$ is the function that returns the state reached performing action a in state s
- $Cost(s, a)$ is the cost of performing an action a in state s. In many state spaces $Cost(s, a)$ is a constant, but this is not true in general.

3.2.1 Uninformed search

Uninformed search, also known as blind search, specifies that the strategies have no additional information about states beyond that provided in the problem definition [16][13]. All they can do is generate successors and distinguish a goal state from a non-goal state. So there is no estimation of how far the goal state is from the current state.

A few uninformed search techniques are:

- Breadth-first search,
- Depth-first search,
- Uniform-cost search [10].

3.2.2 Heuristic search

Heuristic search, also known as informed search, uses problem-specific knowledge beyond the definition of the problem. It can find solutions more efficiently than an uninformed strategy [17][12][5].

In this search strategy, a node t is selected for expansion based on an evaluation function, $f(t)$. The evaluation function is construed as a cost estimate, so the node with the lowest $f(t)$ is expanded first. The choice of f determines the search strategy.

Most informed search algorithms include, as a component of f, a heuristic function denoted $h(t)$:

$h(t) =$ estimated cost of the cheapest path from the state t to a goal state.

Th heuristic function is a problem specific measure and is the most common form in which additional knowledge of the problem is imparted to the search algorithm. If T is a goal state, then $h(T) = 0$.

3.2.2.1 Best-first search

This search technique tries to expand the node that is closest to the goal node, on the grounds that this is likely to lead to a solution quickly. It evaluates nodes by using just the heuristic function; that is, $f(t) = h(t)$. In other words, it makes the locally optimal choice at each state [15].

The best-first tree search is incomplete even in a finite state space and does not always reach the optimal goal state. Moreover, it may turn into unguided *depth-first search* in the worst case.

3.2.2.2 A* search

A* search [7][4] combines the strengths of uniform-cost search and best-first search. It evaluates nodes by combining $g(t)$, the cost to reach the node, and $h(t)$, the

estiamted cost to get from the node to the goal:

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

Since $g(t)$ gives the cost of getting from the start node to node t, and $h(t)$ is the estimated cost of the cheapest path from t to the goal,

$$f(t) = \text{Estimated cost of the cheapest solution through } t$$

A heuristic function is called *admissible* if it never overestimates the cost to reach the goal. If $h(t)$ is admissible for all states t, then A* search is guaranteed to return a least-cost path from start to goal i.e., optimal and complete.

3.3 Exact cover problem

The Exact Cover problem can be formulated in the following way: given a collection S of subsets of a set X, an exact cover of X is a subcollection S^* of S that satisfies two conditions:

- each element in X is contained in exactly one subset in S^* i.e., the subsets in S^* are pairwise disjoint;
- the subsets in S^* cover X i.e., the union of the subsets in S^* is X.

For example, let $S = \{A, B, C, D, E, F\}$ be a collection of subsets of a set $X = \{1, 2, 3, 4, 5, 6, 7\}$ where

$$A = \{1, 4, 7\}$$

$$B = \{1, 4\}$$

$$C = \{4, 5, 7\}$$

$$D = \{3, 5, 6\}$$

$$E = \{2, 3, 6, 7\}$$

$$F = \{2, 7\}$$

Then the subcollection $S^* = \{B, D, F\}$ is an exact cover because each element of X is contained in exactly one of the subsets of S^*.

$$B \cup D \cup F = \{1, 2, 3, 4, 5, 6, 7\} = X$$

$$B \cap D = \emptyset, D \cap F = \emptyset, B \cap F = \emptyset$$

The exact cover problem can also be represented in terms of an incidence matrix. The matrix has one row for each subset in S and one column for each element in X. The entry in a particular row and column is 1 if the corresponding subset contains the corresponding element and is 0 otherwise. So, an exact cover is a selection of rows such that each column contains a 1 in precisely one selected row.

If the above example is represented in matrix form, it looks like Table 3.2.

	1	**2**	**3**	**4**	**5**	**6**	**7**
A	1	0	0	1	0	0	1
B	1	0	0	1	0	0	0
C	0	0	0	1	1	0	1
D	0	0	1	0	1	1	0
E	0	1	1	0	0	1	1
F	0	1	0	0	0	0	1

Table 3.2: Sample matrix representation of an exact cover problem. The matrix has a 1 in position (i, j) if the i$^{\text{th}}$ set contains the j$^{\text{th}}$ element.

From Table 3.2, it is evident that $S^* = \{B, D, F\}$ is an exact cover because each column contains a 1 in exactly one of the subsets of S^*, i.e., each element is contained in precisely one selected subset.

The exact cover problem is decision-based problem [11]. It is NP-complete and one of Karp's 21 NP-complete problems [8].

3.3.1 Algorithm X

Knuth proposed an Algorithm X which can find all the solutions to the exact cover problem [11]. He describes it as an "obvious trial-and-error approach". It is a straightforward recursive, non-deterministic, depth-first, backtracking algorithm.

This algorithm requires the input to an exact cover problem to be provided in the form of a matrix A consisting of 0s and 1s. The purpose is to pick a subset of the rows such that 1 appears in each column exactly once. Following is the pseudo-code for Algorithm X -

```
1: if A has no columns then
2: |   Current partial solution is a valid solution and terminate successfully
3: end if
4: Choose a column, c (in a deterministic way)
5: Choose a row, r, such that A[r, c] = 1 (in a non-deterministic way)
6: Include r in the partial solution
7: for each j such that A[r, j] = 1 do
8: |   for each i such that A[i, j] = 1 do
9: |   |   Delete row i from A
10: |   end for
11: |   Delete column j from A
12: end for
13: Repeat this algorithm recursively on the reduced A
```

The non-deterministic choice of r means that the algorithm recursively repeats over independent subalgorithms; each subalgorithm inherits the current matrix A, but reduces it with respect to a different row r. If column c is entirely zero, there are no subalgorithms and the process terminates unsuccessfully.

3.3.2 Dancing links and DLX

The fundamental idea of dancing links is based on an elegant observation about a doubly linked list of nodes [11]. Suppose x points to an element of a doubly linked list. $L[x]$ and $R[x]$ be pointers to the predecessor and successor of the element pointed by x. Then the operations

$$L[R[x]] \leftarrow L[x], \qquad R[L[x]] \leftarrow R[x]$$

remove x from the linked list. On the other hand, the operations

$$L[R[x]] \leftarrow x, \qquad R[L[x]] \leftarrow x$$

put x back into the list again.

Knuth uses the dancing links technique to demonstrate an efficient implementation of Algorithm X called DLX [11]. A naive implementation of the Algorithm X spends an inordinate amount of time searching for 1's. When selecting a column, the entire matrix has to be searched for 1's. When selecting a row, an entire column has to be searched for 1's. After selecting a row, that row and a number of columns has to be searched for 1's. To improve this search time from complexity $O(n)$ to $O(1)$, a sparse matrix where only 1's are stored is used, implemented as doubly linked lists.

At all times, each node x in the sparse matrix has five pointers $L[x]$, $R[x]$, $U[x]$, $D[x]$, $C[x]$; where

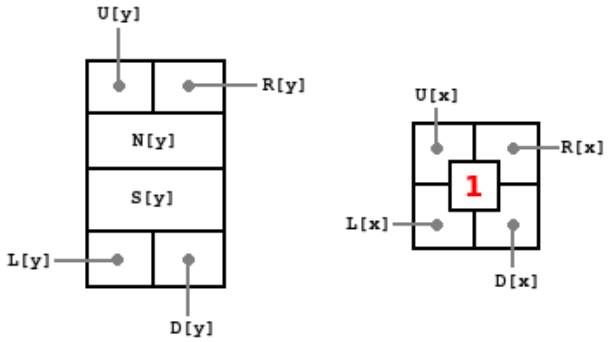
$L[x]$ points to the nearest node with 1 in the same row left to x.

$R[x]$ points to the nearest node with 1 in the same row right to x.

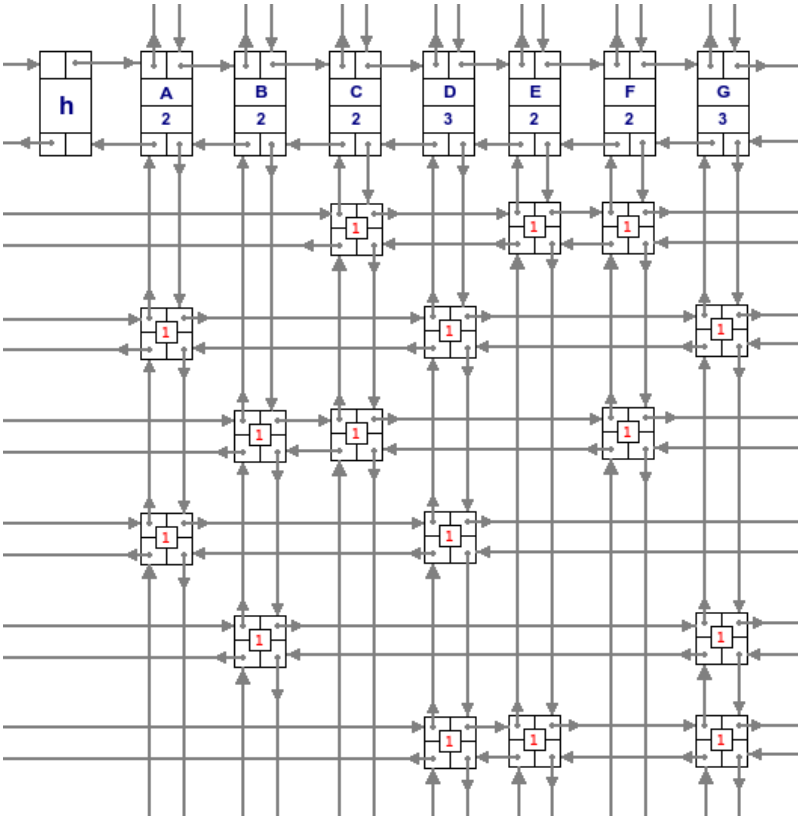
$U[x]$ points to the nearest node with 1 in the same column above x.

$D[x]$ points to the nearest node with 1 in the same column below x.

$C[x]$ points to list header node to which x belongs.



(a) Fields of the head nodes and the data nodes



(b) The head nodes and the data nodes are connected as doubly linked lists

Figure 3.2: Sample mesh representation of DLX algorithm.

Rows of the matrix are circular doubly linked lists connected via the L and R pointers; columns are also circular doubly linked lists attached via the U and D pointers.

Each column list also includes a special node called list *header node* and it forms a special control row consisting of all the columns which still exist in the matrix. A header node y is just like a simple node but has two extra fields $S[y]$ and $N[y]$.

$S[y]$ specifies the number of nodes in the y column.

$N[y]$ contains the name (or index) of the column for identification.

The L and R fields of the list headers link together all columns that still need to be covered. This circular list also includes a special column object called the root, h, which serves as a master header for all the active headers.

For example, the matrix of Table 3.2 is represented by the mesh of 4 way connected nodes shown above in Figure 3.2.

The covering operation removes column c from the header list and removes all rows in the c's own list from the other column lists they are in.

```
1: procedure COVER(c)
2: |   L[R[c]] ← L[c] and R[L[c]] ← R[c]
3: |   for each i ← D[c], D[D[c]], ..., while i ≠ c do
4: |   |   for each j ← R[i], R[R[i]], ..., while j ≠ i do
5: |   |   |   U[D[j]] ← U[j] and D[U[j]] ← D[j]
6: |   |   |   S[C[j]] ← S[C[j]] + 1
7: |   |   end for
8: |   end for
9: end procedure
```

If the column A from Figure 3.2 is covered by using the above algorithm, then the structure takes the form of Figure 3.3.

The operation uncovering takes place in precisely the reverse order of the covering operation, using the technique of dancing links. The uncovering operation for column c is described below.

```
1: procedure UNCOVER(c)
2: |   for each i ← U[c], U[U[c]], ..., while i ≠ c do
3: |   |   for each j ← L[i], L[L[i]], ..., while j ≠ i do
4: |   |   |   S[C[j]] ← S[C[j]] + 1
5: |   |   |   U[D[j]] ← j and D[U[j]] ← j
6: |   |   end for
7: |   end for
8: |   L[R[c]] ← c and R[L[c]] ← c
9: end procedure
```

Thus the DLX algorithm is described in the algorithm 1. The *Search* procedure is invoked initially with $k = 0$.

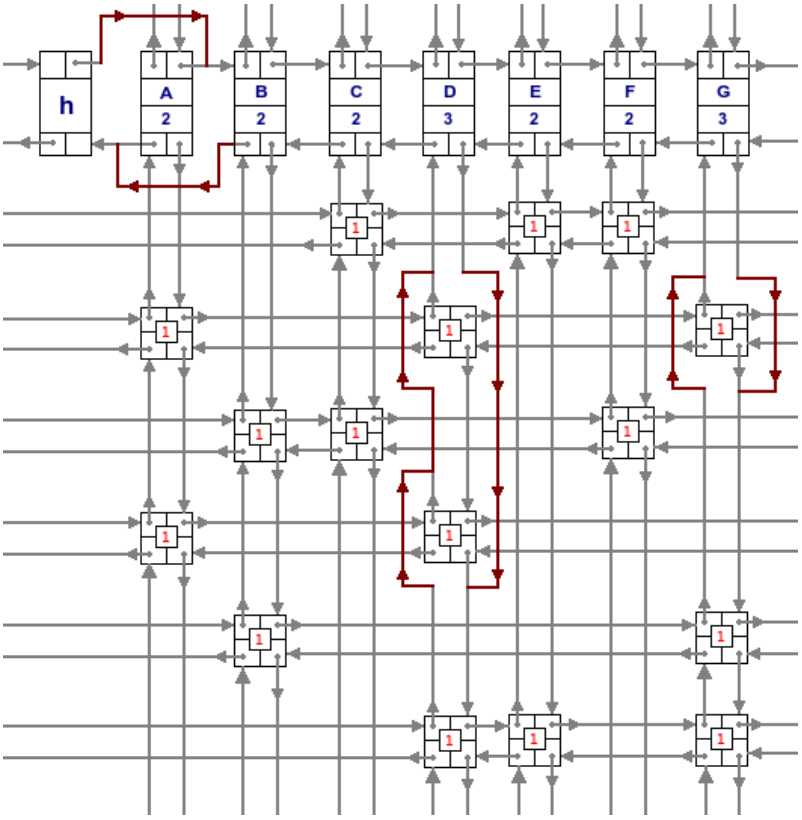


Figure 3.3: Resultant mesh representation after covering the column A. The column head A is not connected to the head list and the data nodes covered by column A are not linked to the remaining nodes.

The operation of printing the current solution is easy: successively print the rows containing $O_0, O_1, \ldots, O_{k-1}$, where the row containing data object O is printed by printing $N[C[O]], N[C[R[O]]], N[C[R[R[O]]]], \ldots$ etc.

Algorithm 1 DLX algorithm

```
1: procedure SEARCH(k)
2:     if R[h] = h then
3:         print the current solution and exit
4:     else
5:         choose a column c
6:     end if
7:     COVER(c)
8:     for each r ← D[c], D[D[c]], ..., while r ≠ c do
9:         O_k ← r
10:        for each j ← R[r], R[R[r]], ..., while j ≠ r do
11:            COVER(j)
12:        end for
13:        SEARCH(k + 1)
14:        r ← O_k
15:        c ← C[r]
16:        for each j ← L[r], L[L[r]], ..., while j ≠ r do
17:            UNCOVER(j)
18:        end for
19:        COVER(c)
20:    end for
21:    UNCOVER(c)
22: end procedure
```

Chapter 4

Methodology

4.1 State

Before describing our method in detail, we first define a *state* in context of the Mondrian Tiling problem. A state is a square grid of size $n \times n$ tiled using some rectangles while maintaining the following two primary conditions.

1. The rectangles used to fill the grid must be non-overlapping, and have integer dimensions.

2. The rectangles must be pairwise non-congruent (i.e., if a rectangle of size 2×3 is used, then another rectangle of size 2×3 or 3×2 cannot be used)

The Mondrian score of a state is the difference in area between the largest and smallest rectangles used for tiling. Thus, each state is a valid solution to the Mondrian Tliling problem.

1	1	2	2
1	1	2	2
4	4	2	2
3	3	3	3

(a) Valid state of 4×4 grid

1	1	1	2
1	1	1	2
4	4	4	2
3	3	3	3

(b) Invalid state of 4×4 grid

Figure 4.1: Sample states of 4×4 grid.

In Figure 4.1a, the rectangles of sizes $2 \times 2, 1 \times 2, 3 \times 2$ and 4×1 are used to fill up

the grid; this is an example of a valid state because no two rectangles are pairwise congruent. Also, the Mondrian score of this state is $(3 * 2) - (1 * 2) = 4$. On the other hand, Figure 4.1b is an invalid state since the rectangles of sizes 3×1 and 1×3 are congruent to each other.

4.2 State space representation

As mentioned in Section 3.2, the state space is the set of all states. Some actions are applied to the states to reach the goal state.

The existing approach by Bassen [1] first finds a set of partitions in the algebraic representation. Then it tests if the partitions can be converted into a valid solution (i.e., covering an empty grid) to the problem in the geometric representation. It uses the help of two different representations. So, finding the partitions, which is NP-complete itself, is not good enough, and most of the time, partitions do not lead to a solution. Also, the conversion of algebraic representations into geometric ones is quite a difficult task. An elaborate discussion of this approach can be found in Section 5.2.

The shortcomings of the existing approach can be alleviated by using the state-space representation. At every step, the latter deals with many states, and each state is a reasonable solution to the problem with Mondrian score. Also, each state can be visualized in terms of a geometric point of view. So, there is no need for any conversion. This kind of representation aims to find a state (i.e., optimal state) with Mondrian score smaller than all the other states to reach the goal state through some actions.

4.3 Generate initial valid state

It is evident now that the states are at the core of the state-space representation. So, to progress with this type of representation, the first essential task is to produce

an initial state.

The fundamental part of forming any initial state is to cover an empty square grid of size $n \times n$ while preserving the two primary constraints. The process of covering an entire grid can further be decomposed into two stages. In the first stage, the bottom-most row of an empty grid is covered up using different rectangles. In the second stage, the remaining part of the grid is filled up using a method similar to a watershed algorithm.

Stage I: Covering the bottom-most row: The basic idea behind covering the bottom-most row of an $n \times n$ grid is to pick random rectangles i.e., rectangles with random lengths and breadths $\leq n$, to fit in the grid, then place those rectangles in the bottom-most row. This covering process may start from either the left or the right side of the grid. As the process progresses, the number of uncovered boxes (each 1×1 grid square can be thought of as a *box* and n^2 boxes form the complete $n \times n$ grid) in the bottom-most row decreases. Suppose the number of uncovered boxes is less than a threshold. In that case, a rectangle can be picked with one random side, where another side is the same as the remaining uncovered boxes so that placing this rectangle will cover the entire bottom-most row. One important point to remember while choosing a random rectangle is that the new rectangle must be non-congruent to already pre-selected rectangles.

An example of the above process is illustrated in Figure 4.2. The process starts with an empty 12×12 grid, and the number of uncovered boxes in the bottom-most row is initially 12. A random 5×3 rectangle is placed on the bottom-most right portion of the grid. Similarly, another rectangle with a length, breadth of 6 and 4, respectively, is chosen and placed next to the previous rectangle. While choosing the next rectangle, it is essential to ensure that the new rectangle cannot be congruent to the two selected rectangles, i.e., it cannot have dimensions of $5 \times 3, 3 \times 5, 6 \times 4, 4 \times 6$. The subsequent random rectangle is of order 3×2. Now, the number of uncovered boxes is 2, which is relatively small. So, a random rectangle is picked with a random

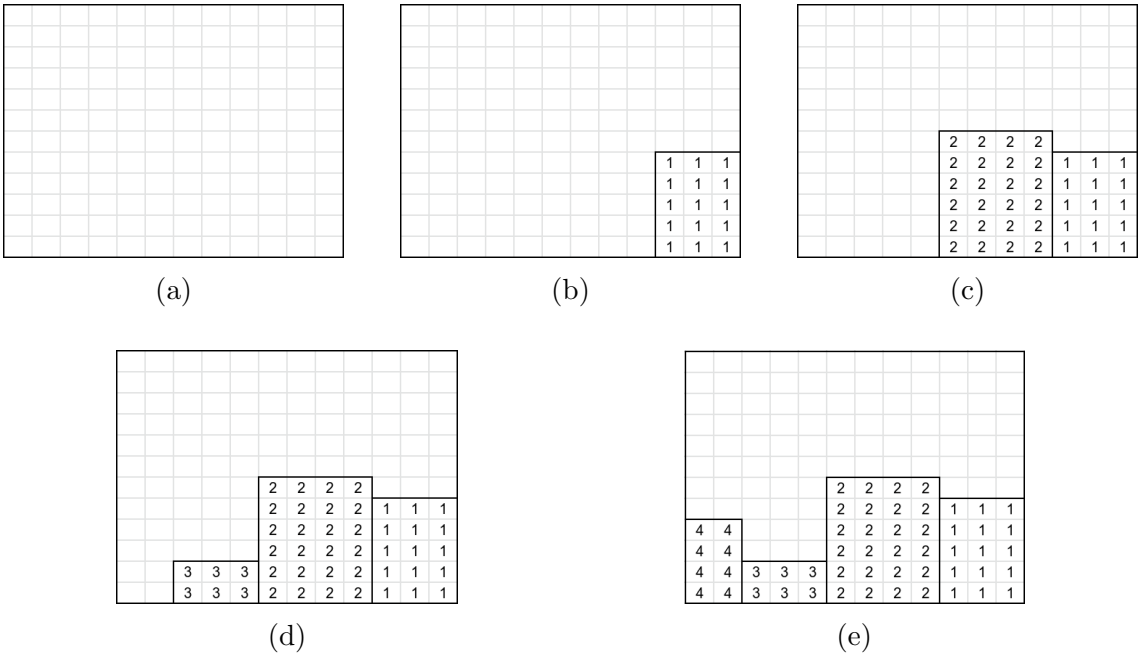


Figure 4.2: Step by step process of covering the bottom-most row.

length of 4 and breadth as the number of uncovered boxes, which is 2. Now, the entire bottom-most row is covered, and this stage is complete.

Stage II: Covering remaining part of the grid: After the previous stage, the grid may not be filled up, and it is time to cover up the remaining uncovered portion of the grid. At the end of Stage I, the grid looks similar to a combination of hills and valleys in between those hills. Therefore, the basic objective of this stage is to fill up the valleys; this will eventually lead to the covering of the entire grid. The process of filling up valleys consists of two steps. At first, the valleys are identified and, then the valleys can be filled up by applying a method similar to a watershed algorithm.

All individual boxes of the grid are provided coordinates based on their position (i.e., row and column). Then, the coordinates of the uncovered boxes are examined to identify the valleys. A collection of boxes is considered part of a valley if the adjoining boundary boxes are covered or edge of the grid. For example, in Figure 4.2e, the uncovered boxes in between the rectangles marked by 2, 4 and just above the rectangle marked by 3 are part of a valley because these uncovered boxes

have adjoining boundaries from rectangles marked by 2, 3, and 4.

At one time, multiple valleys may be recognized in an instance of the grid. Among these, the deepest valley is first attempted for covering up. To cover up a valley, imagine pouring water from the top of the grid on the valley. The water that gets stuck in the valley represents the boxes that are to be covered up, and the dimensions of the new rectangle can be easily computed from these boxes. Before covering the new rectangle permanently, it must be checked if this rectangle is non-congruent to the existing rectangles. If yes, then the rectangle can be covered permanently. Otherwise, either the water level of the valley can be slightly increased (in an abstract sense, the water level may rise to higher than the adjoining hill) to ensure non-congruence, or another valley can be selected.

After covering one valley in this way, the valleys that may be different from before or recently formed are identified, and another subsequent valley is tried to be covered up. This process is iterated until the complete grid is covered.

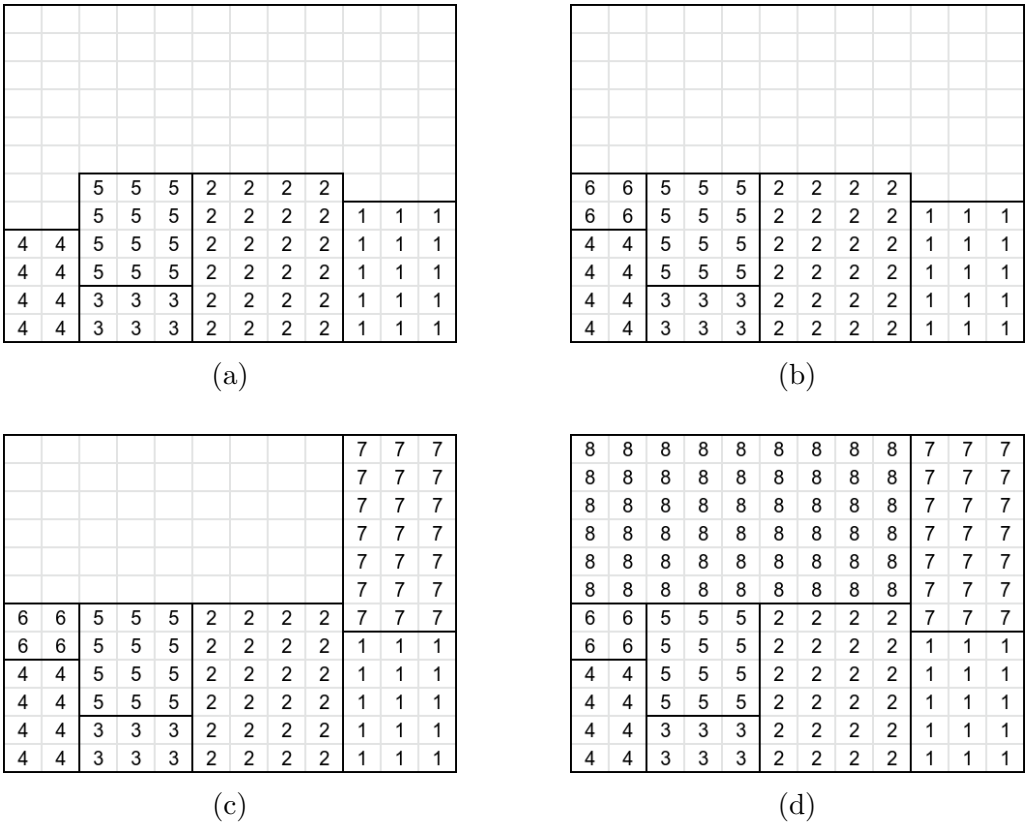


Figure 4.3: Step by step process of covering the remaining portion of the grid.

An example of the above process is illustrated in Figure 4.3. Figure 4.3a is a continuation of the grid from Figure 4.2e. There are hills and valleys present in Figure 4.2e, and from this figure, it can be recognized that there are two valleys present. The first valley can be found within rectangles marked by $2, 3, 4$, whereas another valley is among rectangles marked $1, 2$, and the right edge of the grid. Since the former valley is relatively deeper, abstract water is dropped on this valley and floods the uncovered boxes, and so the water level rises to the rectangle marked by 4. This flooded region of uncovered boxes indicates a new possible rectangle of dimension 2×3. However, since already a rectangle with marked 3 exists with precisely the same dimensions, this new rectangle cannot be covered. Therefore, the water level is slightly increased to match the height of the rectangle marked with 2, and a new possible rectangle of dimension 4×3 is produced. This rectangle is not congruent to any existing rectangle and is used to cover the valley mentioned above.

In the next step, two valleys are found within rectangles marked $1, 2$, and the right edge of the grid and rectangles marked $4, 5$, and the left side of the grid. The latter valley is covered up using a rectangle of dimension 2×2.

Next, one shallow valley is found with rectangles marked $1, 2$, and the right edge of the grid. This valley can be covered up using a small rectangle of dimension 1×3. Nevertheless, that will lead to a poor Mondrian score as the area of the smallest rectangle will become only 3, which is relatively inadequate. That is why the entire upper portion of the rectangle marked 1 is covered up to prevent a further generation of any narrow valley. The remaining uncovered rectangular part is covered using a 6×9 rectangle.

The 12×12 grid is now entirely covered and thus has become a valid state. Therefore, the *Mondrian* score of this state is $(6 * 9) - (2 * 2) = 50$. It is not a very good score, but this score can be further improved.

4.4 Improve score of a valid state

The Mondrian score of an initial state is often inadequate due to the presence of either a very large or small rectangle. For example, in Figure 4.3, the large rectangle marked with 8 and the small rectangle marked with 6 make the score poor. If, in some way, the large rectangle can be made relatively smaller or the small rectangle can be made relatively bigger, then the score would definitely decrease. This suggests that the score of a state can be improved using some operations. These operations are known as *state operations*. Different state operations are applied to states successively to improve the Mondrian score. A detailed description of different state operations can be found in Section 4.4.1.

4.4.1 State operations

The Mondrian score of a state can be improved by shrinking the largest rectangle area or expanding the smallest rectangle area. The main objective of employing the state operations is to improve the score. These operations are exercised only on valid states. Each operation is designed to convert a state to another state with the same or a comparatively lower score and, most importantly, without violating the primary two constraints of any state. Four different operations can be applied to any state; however, the operations may not always reduce the score. The four different state operations are

1. Merge operation
2. Split operation
3. Merge-Split operation
4. Split-Merge operation

4.4.1.1 Merge operation

The primary motivation behind the merge operation is to increase the smallest rectangle area, thus improving the score. At first, the rectangles used to fill the grid

with relatively small area sizes are located in this operation. These rectangles are sorted based on the area size in ascending order. Then, starting from the smallest rectangle, it is checked if the selected rectangle can be merged with any of its adjoining neighbours. Finally, it is examined if the merging process of two rectangles strictly produces another rectangle that is non-congruent to all other existing rectangles apart from the two-parent rectangles. Thus, the child rectangle has an area larger than the parent rectangles, so the minimum area of rectangles increases, improving the Mondrian score.

1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	8	8	10	10	10
1	2	2	3	5	5	5	8	8	10	10	10
1	2	2	3	5	5	5	7	7	7	7	7
1	2	2	3	5	5	5	7	7	7	7	7
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	4	4	4	4	4	4	4	4	4
1	2	2	4	4	4	4	4	4	4	4	4

Figure 4.4: Before merge operation on a 12×12 state with score of 18.

In Figure 4.4, the maximum and minimum areas are 24 and 6, respectively. So the Mondrian score is $24 - 6 = 18$. Here, the smallest rectangle of dimension 2×3 is the one marked with 10. So it can be merged with adjoining rectangles marked with $7, 8, 9$. However, merging with rectangles marked with 7 and 8 does not produce a proper rectangle; instead, it forms an L-shaped structure. Therefore, the rectangles marked with 9 and 10 are merged and create another rectangle of dimension 6×3. The resultant state is shown in Figure 4.5.

Now, although the maximum area is still 24, the minimum area has become 10. So, the Mondrian score is $24 - 10 = 14$, which is a significant improvement over the previous score of 18.

1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	7	7	7	7	7
1	2	2	3	5	5	5	7	7	7	7	7
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	4	4	4	4	4	4	4	4	4
1	2	2	4	4	4	4	4	4	4	4	4

Figure 4.5: After merge operation on a 12×12 state with score of 14.

4.4.1.2 Split operation

The primary motivation behind the split operation is to decrease the largest rectangle area, thus improving the score. At first, the rectangles used to fill the grid with relatively high area sizes are located in this operation. These rectangles are sorted based on the area size in descending order. Then, starting from the largest rectangle, it is checked if the selected rectangle can be split into two new child rectangles so that the difference between the children's areas is minimum. At the same time, it is examined if the child rectangles are non-congruent to all other existing rectangles apart from the parent rectangle. Thus, the child rectangles have areas smaller than the parent rectangle, so the maximum area of rectangles decreases, improving the Mondrian score.

In Figure 4.6, the maximum and minimum areas are 24 and 6, respectively. So the Mondrian score is $24 - 6 = 18$. Here the largest rectangle of dimension 3×8 is the one marked with 7. So it can be split in different ways. However, splitting the parent into $3 \times 5, 3 \times 3$ rectangles causes a minimum gap of areas as well as non-congruent to other rectangles. Therefore, two-child rectangles with dimensions 3×5 and 3×3 are created. The resultant state is shown in Figure 4.7.

Now, although the minimum area is still 6, the maximum area has become 18. So, the Mondrian score is $18 - 6 = 12$, which is a significant improvement over the

1	1	3	3	3	4	4	5	5	5
1	1	3	3	3	4	4	5	5	5
1	1	3	3	3	4	4	5	5	5
1	1	3	3	3	4	4	5	5	5
1	1	3	3	3	4	4	6	6	6
1	1	3	3	3	4	4	6	6	6
1	1	7	7	7	7	7	7	7	7
1	1	7	7	7	7	7	7	7	7
1	1	7	7	7	7	7	7	7	7
2	2	2	2	2	2	2	2	2	2

Figure 4.6: Before split operation on a 10×10 state with score of 18.

1	1	3	3	3	4	4	5	5	5
1	1	3	3	3	4	4	5	5	5
1	1	3	3	3	4	4	5	5	5
1	1	3	3	3	4	4	5	5	5
1	1	3	3	3	4	4	6	6	6
1	1	3	3	3	4	4	6	6	6
1	1	7	7	7	7	7	8	8	8
1	1	7	7	7	7	7	8	8	8
1	1	7	7	7	7	7	8	8	8
2	2	2	2	2	2	2	2	2	2

Figure 4.7: After split operation on a 10×10 state with score of 12.

previous score of 18.

4.4.1.3 Merge-Split operation

The primary motivation behind the merge-split operation is to increase the smallest rectangle area, thus improving the score. At first, the rectangles used to fill the grid with relatively small area sizes are located in this operation. These rectangles are sorted based on the area size in ascending order. Then, starting from the smallest rectangle, it is checked if the selected rectangle can be merged with any of its adjoining neighbours. Merge-split operation explicitly allows the formation of an L-shaped structure while merging, unlike the merge operation, which does not. Since only rectangular shapes are allowed in a grid due to the primary constraint, one of

the other adjoining rectangles that do not participate in merging has to sacrifice by splitting up to accommodate space for the L-shaped structure to become a proper rectangle. Hence, the name is merge-split because a split operation follows a merge operation. This operation produces two new rectangles; one results from the merge followed by split operation, while another from split operation only.

Finally, it is examined if the newly formed child rectangles are non-congruent to all other existing rectangles apart from the three-parent rectangles. Thus, the child rectangles have an area larger than the parent rectangles, so the minimum area of rectangles increases, improving the Mondrian score.

1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	8	8	10	10	10
1	2	2	3	5	5	5	8	8	10	10	10
1	2	2	3	5	5	5	7	7	7	7	7
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	4	4	4	4	4	4	4	4	4
1	2	2	4	4	4	4	4	4	4	4	4

Figure 4.8: Before merge-split operation on a 12×12 state with score of 19.

In Figure 4.8, the maximum and minimum areas are 24 and 5, respectively. So the Mondrian score is $24 - 5 = 19$. Here the smallest rectangle of dimension 1×5 is the one marked with 7. So it can be merged with adjoining rectangles marked with $6, 8, 10$. Now, merging with rectangles marked with 7 and 10 produces an L-shaped structure. To convert this L-shaped structure into a proper rectangle, the rectangle marked with 8 splits up and donate some portion, enabling the L-shaped to become a proper rectangle. Therefore, the two new rectangles marked with 7 and 8 are created with dimensions 4×2 and 3×5. The resultant grid is shown in Figure 4.10.

Now, although the maximum area is still 24, the minimum area has become 8. So, the Mondrian score is $24 - 8 = 16$, which is an improvement over the previous

1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	8	8	7	7	7
1	2	2	3	5	5	5	8	8	7	7	7
1	2	2	3	5	5	5	7	7	7	7	7
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	4	4	4	4	4	4	4	4	4
1	2	2	4	4	4	4	4	4	4	4	4

Figure 4.9: Intermediate step of merge-split operation on a 12×12 state.

1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	8	8	9	9	9
1	2	2	3	5	5	5	7	7	7	7	7
1	2	2	3	5	5	5	7	7	7	7	7
1	2	2	3	5	5	5	7	7	7	7	7
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	4	4	4	4	4	4	4	4	4
1	2	2	4	4	4	4	4	4	4	4	4

Figure 4.10: After merge-split operation on a 12×12 state with score of 16.

score of 19.

4.4.1.4 Split-Merge operation

The primary motivation behind the split-merge operation is to increase the smallest rectangle area, thus improving the score. At first, the rectangles used to fill the grid with relatively small area sizes are located in this operation. These rectangles are sorted based on the area size in ascending order. Then, starting from the smallest rectangle, it is checked if the selected rectangle can be divided into multiple segments so that the segments can be merged with its adjoining neighbours. Hence, the name is split-merge because a merge operation follows a split operation. This operation

produces multiple new rectangles, whereas the initially selected rectangle is wholly destroyed.

Finally, it is examined if the newly formed child rectangles are non-congruent to all other existing rectangles. Thus, the child rectangles have areas larger than the parent rectangles. The selected rectangle with a relatively smaller area is dissolved, so the minimum area of rectangles increases, improving the Mondrian score.

1	4	4	4	4	4	4	8	8	9	9	9
1	4	4	4	4	4	4	8	8	9	9	9
1	4	4	4	4	4	4	8	8	10	10	10
1	4	4	4	4	4	4	8	8	10	10	10
1	2	2	3	5	5	5	8	8	10	10	10
1	2	2	3	5	5	5	8	8	10	10	10
1	2	2	3	5	5	5	7	7	7	7	7
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	11	11	11	11	11	11	11	11	11
1	2	2	11	11	11	11	11	11	11	11	11

Figure 4.11: Before split-merge operation on a 12×12 state with score of 19.

In Figure 4.11, the maximum and minimum areas are 24 and 5, respectively. So the Mondrian score is $24 - 5 = 19$. Here the smallest rectangle of dimension 1×5 is the one marked with 7. This rectangle is divided into two segments of dimensions $1 \times 2, 1 \times 3$. These two segments are consumed by the adjoining rectangles marked with 8 and 10, respectively.

Therefore, the two new rectangles marked with 8 and 10 are reformed with dimensions 7×2 and 5×3, whereas the rectangle marked with 7 does not exist anymore. The resultant grid is shown in Figure 4.13.

Now, although the maximum area is still 24, the minimum area has become 6. So, the Mondrian score is $24 - 6 = 18$, which is a minor improvement over the previous score of 19.

The merge-split and split-merge operations may look similar, but the working procedure and outputs are entirely different. One rectangle is wholly terminated in

1	4	4	4	4	4	4	8	8	9	9	9
1	4	4	4	4	4	4	8	8	9	9	9
1	4	4	4	4	4	4	8	8	10	10	10
1	4	4	4	4	4	4	8	8	10	10	10
1	2	2	3	5	5	5	8	8	10	10	10
1	2	2	3	5	5	5	8	8	10	10	10
1	2	2	3	5	5	5	7	7	7	7	7
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	11	11	11	11	11	11	11	11	11
1	2	2	11	11	11	11	11	11	11	11	11

Figure 4.12: Intermediate step of split-merge operation on a 12×12 state.

1	4	4	4	4	4	4	8	8	9	9	9
1	4	4	4	4	4	4	8	8	9	9	9
1	4	4	4	4	4	4	8	8	10	10	10
1	4	4	4	4	4	4	8	8	10	10	10
1	2	2	3	5	5	5	8	8	10	10	10
1	2	2	3	5	5	5	8	8	10	10	10
1	2	2	3	5	5	5	8	8	10	10	10
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	3	6	6	6	6	6	6	6	6
1	2	2	11	11	11	11	11	11	11	11	11
1	2	2	11	11	11	11	11	11	11	11	11

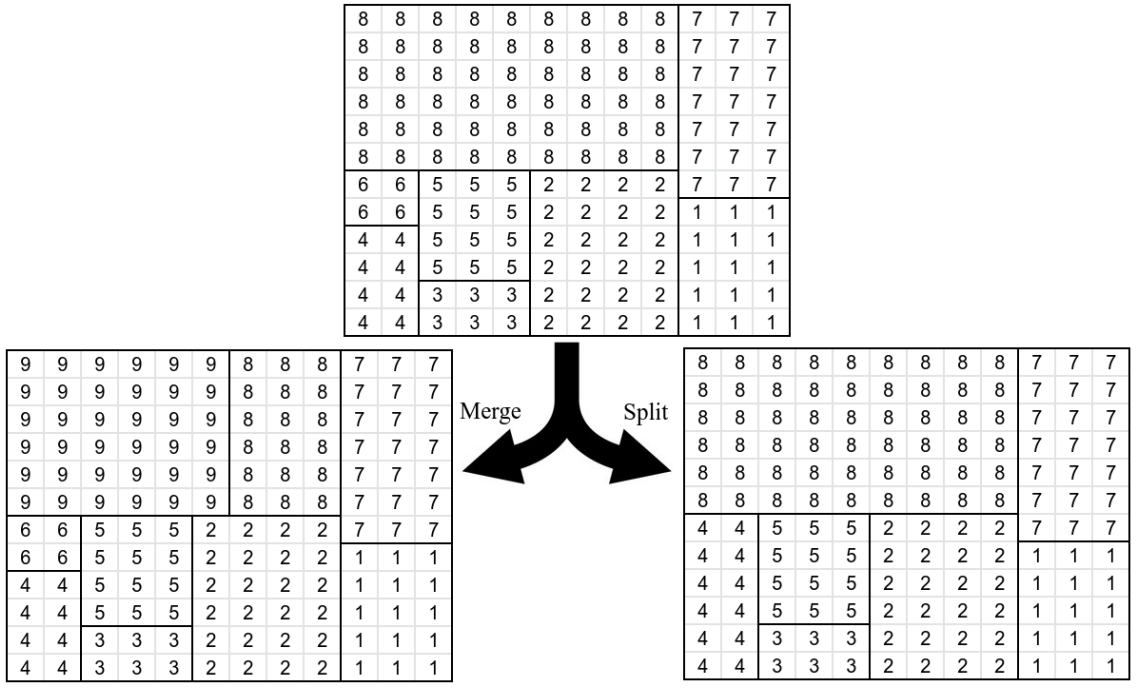
Figure 4.13: After split-merge operation on a 12×12 state with score of 18.

split-merge, whereas this type of termination is not there in merge-split.

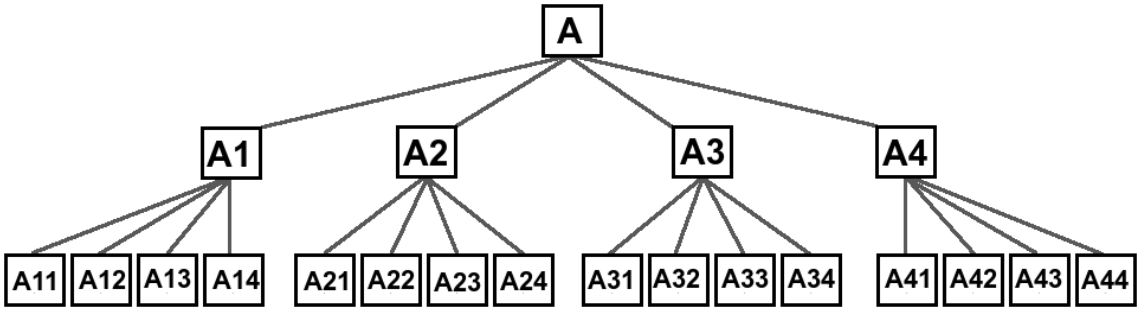
4.5 Apply state space search algorithm

By now, it should be apparent that in this thesis paper, the Mondrian Tliling problem is dealt with in terms of states. Previous sections discuss how to construct a state and some state operations that may reduce the score. The aim is to find a state with a score lower than all other states.

The four state operations, designed in the previous sections, can be applied to a given state. Consider, an initial state, A (say), is generated by the process mentioned



(a)



(b)

Figure 4.14: Sample state space: (a) result of applying Merge and Split operation on a sample 12×12 state (b) sample tree formed by applying successive operations on state A

in Section 4.3. Then the four state operations are applied to A and produce four new states, namely A_1, A_2, A_3 and A_4. Again, the same operations can also be applied to these states, resulting in the creation of 16 states. This process can be iterated many times. If some states are exhibited multiple times, those can be banished immediately, without applying any operations.

Nevertheless, a directional tree of branching factor 4 is accumulated where each node is a unique state and edges represent the state operations. The purpose behind this type of tree is to reach as many states as possible while keeping track of the

state with the lowest score explored yet. By randomly generating an initial state, different trees are formed for each initial state.

A state space search algorithm can be employed on this type of tree to obtain the goal state (a state with a score lower than all other states in the tree) efficiently. However, in order to do that, a precise mapping must be established between a state space search algorithm and the current system.

A standard state space algorithm is usually expressed by a weighted graph consisting of nodes, edges, weights of the edges, and heuristic values (wherever applicable). In the current system, the nodes, edges, and heuristic function are represented by the states, the four-state operations, and the Mondrian score of a state. In contrast, the weights of the edges are simply 1.

The motivation behind using a state space search algorithm is to have a specific policy while exploring the tree of states. For example, if the states A_1, A_2, A_3, and A_4 are spawned from a state A, then which state to visit next may affect searching performance and make it easier or difficult to reach the goal state. The strategy of the search algorithm resolves this problem of traversing. Thus, the policy of the search algorithm explicitly determines the exploration and exploitation of the tree.

A particular case study : A* search algorithm

As discussed earlier in Section 3.2.2.2, A* search algorithm is an absolute balance of exploration and exploitation because it tries to locate the goal node based on foresight and experience. It evaluates a node t using $g(t)$, the cost to reach the node and $h(t)$ the cost to get from the node to the goal.

In the proposed system, the heuristic function h is described as the Mondrian score of a state. The cost function g to reach the current state is formulated as number of state operations consecutively applied to the initial state to reach the current state. It can be regarded as the depth of the current state from the initial

state. Thus, the evaluation function f for a given state t becomes-

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

where

$$h(t) = \text{Mondrian score of the state } t$$

$$g(t) = \text{Number of operations applied to initial state to obtain } t$$

Based on the above evaluation function, the following states are explored. A priority queue is utilised to get a state with minimum evaluation functional value in $O(1)$ time at each step. Since the tree of states grows exponentially, it is not easy to hold all the upcoming states in the priority queue. That is why the size of the priority queue is predetermined to prevent an outburst of the number of states. Also, if the depth of a branch of the tree exceeds a threshold, then it is pruned to prevent further exploitation of a branch. This action is practised to introduce exploration rather than exploiting a single branch. These two parameters can be adjusted as required.

Since the size of the priority queue is limited, only the states with relatively good scores are stored with the hope that these will eventually lead to states with better scores. When the priority queue is not entirely exhausted at the primary stage, all the resultant states from state operations are inserted. However, the moment the priority queue is finished, the new states cannot be inserted. At this moment, the states are inserted into the queue using intelligent tactics. A state can be inserted if it possesses a better score than the state with the worst score stored in the queue. Therefore, a stored state with the worst score is removed from the queue to make room for a better state and keep the queue size constant.

A concise algorithm is presented below.

Algorithm 2 Finding the state with the lowest score using A* algorithm

Input: Two parameters $maxdepth, maxsize$
Output: The state with the lowest Mondrian score

```
1:  procedure SearchAlgorithm(maxdepth, maxsize)
2:      t ← Generate an initial state
3:      g(t) ← 0
4:      h(t) ← score(t)
5:      f(t) ← g(t) + h(t)
6:      Initialize priority queue
7:      Insert(queue, t)
8:      best_state ← t
9:      while Queue is non-empty do
10:         s ← Pop(queue)
11:         if g(s) < maxdepth then
12:             for X ∈ State Operations do
13:                 a ← Apply X(s)
14:                 if a is already explored then
15:                     Continue
16:                 else
17:                     g(a) ← g(s) + 1
18:                     h(a) ← score(a)
19:                     f(a) ← g(a) + h(a)
20:                     if score(a) < score(best_state) then
21:                         best_state ← a
22:                     end if
23:                     if size(queue) ≤ maxsize then
24:                         Insert(queue, a)
25:                     else
26:                         b ← Worst State(queue)
27:                         if score(a) < score(b) then
28:                             Remove(queue, b)
29:                             Insert(queue, a)
30:                         end if
31:                     end if
32:                 end if
33:             end for
34:         end if
35:     end while
36:     return best_state
37: end procedure
```

Chapter 5

Experiment Setup

5.1 Objective of the experiment

The experiment aims to demonstrate that the proposed system is more efficient than the existing brute force approach, and the effectiveness of the former is comparable to that of the latter approach. The proposed system is more efficient because it trades with some states that are valid solutions to the problem at any point in time. So, it can provide adequate solutions, which may not be optimal from the very beginning. In contrast, the existing approach finds some algebraic partitions and transforms those partitions into geometric representation to check if an empty square grid can be filled with the partitions. Unfortunately, both these tasks are NP-complete and require a substantial amount of time for completion. So, it cannot even produce any non-trivial Mondrian score for a large grid size; the optimal score is far away. At the same time, the proposed system can quickly present a good score for a large grid size.

On the other hand, the existing brute force approach fundamentally iterates every possibility available to solve the problem, so it is moderately challenging to beat its optimal solutions. Although the proposed scheme does not necessarily check all possible states to avoid being NP-complete itself, it does provide analogous solutions to the previous approach. A detailed discussion on the existing approach can be found in the next section.

5.2 Baseline description : Brute force approach

The existing brute force approach systematically enumerates all possible candidates for the solution and checks whether each candidate satisfies the problem [2].
The workflow of this approach can be decomposed into 2 phases. For a given $n \times n$ gird, in the first phase, a set of partitions are found in the algebraic sense where the sum of each partition element is n^2. Then in the next phase, these partitions are examined if they can be used to cover up the empty $n \times n$ grid.

5.2.1 Phase 1: Applying subset-sum problem

Before going to the depth, first, it must be understood which rectangles can fit inside a $n \times n$ grid and which rectangles should be incorporated. Any rectangle between 1×1 to $n \times n$ can fit inside a $n \times n$ grid. However, there is no point in using any rectangle with one side being n while another one greater than $\lfloor \frac{n}{2} \rfloor$ because if one rectangle of dimension $n \times \lfloor \frac{n}{2} \rfloor$ is used, then the remaining area is $n^2 - n\lfloor \frac{n}{2} \rfloor = n\lceil \frac{n}{2} \rceil$. So the Mondrian score is always $\geq n$. Thus the suitable range of rectangles lies between 1×1 and $\lfloor \frac{n}{2} \rfloor \times n$. One crucial point to remember that there must not be any congruent rectangles in the set of suitable rectangles while constructing. For example, the set of suitable rectangles for $n = 6$ is $A = \{(1,1), (1,2), (1,3), (1,4), (1,5), (1,6), (2,2), (2,3), (2,4), (2,5), (2,6), (3,3), (3,4), (3,5), (3,6)\}$ with corresponding areas $B = [1, 2, 3, 4, 5, 6, 4, 6, 8, 10, 12, 9, 12, 15, 18]$, where each tuple represents a rectangle of that dimension. An algorithm of the above process can be found below in algorithm 3.

From algorithm 3, the set of suitable rectangles with their corresponding areas is collected. Now the next task is to divide this set into multiple partitions. The sum of each partition area must be equal to the n^2 i.e., $n \times n$ grid area because each partition will be considered a possible solution and rectangles in the partition to cover the grid. For example, the above set A can be partitioned in 373 ways where each partition area sum is $6 * 6 = 36$, which is relatively high for a small $n = 6$.

Therefore, two parameters, namely, the *bound* and the *maximum number of*

Algorithm 3 Finding the set of suitable rectangles for $n \times n$ grid

Input: n
Output: The set of suitable rectangles with areas

$t \leftarrow \{\}$
$s \leftarrow$ Empty list
$i \leftarrow 1$
while $i \leq \lfloor \frac{n}{2} \rfloor$ **do**
    $j \leftarrow 1$
    while $j \leq n$ **do**
        if (i, j) not in t and (j, i) not in t **then**
            $t \leftarrow t \cup \{(i, j)\}$
            Append ij to the list s
        end if
        $j \leftarrow j + 1$
    end while
    $i \leftarrow i + 1$
end while
return t, s

rectangles, are introduced to reduce this number. The parameter bound prevents creating partitions where the largest and smallest rectangle difference is significantly high. Hence, all the partitions now have a maximum area difference among rectangles less than the bound. Thus, this parameter can be considered an upper bound to the Mondrian score obtained later.

The other parameter, the *maximum number of rectangles*, limits the number of rectangles that can be used in a partition. This parameter can be further utilized to find optimal Mondrian score using any restricted number of rectangles only.

Now, the subset sum algorithm (discussed in Section 3.1) can be applied on the set of suitable rectangles, and it returns a set of partitions. Moreover, these partitions now can go through the parameters mentioned above, which reduces the number of eligible partitions to progress to the next phase. However, the subset-sum is an NP-complete problem, and it is computationally expensive.

Instead of imposing the bound parameter on the partitions, it can be directly applied to the set of suitable rectangles. This operation breaks the set into multiple subsets, each containing suitable rectangles with the largest and smallest area difference less than the *bound*.

The sizes of these subsets are significantly smaller than the original set. Therefore, the subset-sum algorithm can be applied to each of these subsets separately, requiring less time in total than the original set due to the size. Then, all the resultant combined partitions can go through the other parameter and further progress to the next phase. The concise algorithm of the above method is described in algorithm 4.

For example, if the parameter *bound* is set as 6, the previously mentioned set A is decomposed into 7 subsets, ultimately leading to 18 partitions without applying the second parameters. The sizes of the subsets $(A_1 - A_7)$ are smaller than the size of A ($\because |A| = 15$).

$$A_1 = \{(1,1),(1,2),(1,3),(1,4),(2,2),(1,5),(1,6),(2,3)\} \text{ and } |A_1| = 8$$

$$A_2 = \{(1,2),(1,3),(1,4),(2,2),(1,5),(1,6),(2,3),(2,4)\} \text{ and } |A_2| = 8$$

$$A_3 = \{(1,3),(1,4),(2,2),(1,5),(1,6),(2,3),(2,4),(3,3)\} \text{ and } |A_3| = 8$$

$$A_4 = \{(1,4),(2,2),(1,5),(1,6),(2,3),(2,4),(3,3),(2,5)\} \text{ and } |A_4| = 8$$

$$A_5 = \{(1,6),(2,3),(2,4),(3,3),(2,5),(2,6),(3,4)\} \text{ and } |A_5| = 7$$

$$A_6 = \{(3,3),(2,5),(2,6),(3,4),(3,5)\} \text{ and } |A_6| = 5$$

$$A_7 = \{(2,6),(3,4),(3,5),(3,6)\} \text{ and } |A_7| = 4$$

Algorithm 4 Finding the subsets of rectangles after applying the *bound*

Input: The set of suitable rectangles, the parameter *bound*
Output: The subsets of suitable rectangles

```
1: procedure BreakTheSet(t, bound)                    ▷ t is the set of rectangles
2:     t ← list(t)                                    ▷ Converting t into list to access index-wise
3:     s ← {}
4:     i ← 1
5:     j ← 2
6:     while i ≤ size(t) do
7:         a = t[i][1] * t[i][2]
8:         b = t[j][1] * t[j][2]
9:         c = t[j − 1][1] * t[j − 1][2]
10:        if b − a > bound then
11:            if c ≠ b then
12:                while c − a > bound do
13:                    i ← i + 1
14:                    a = t[i][1] * t[i][2]
15:                end while
16:                temp ← empty list
17:                for k = i to (j − 1) do
18:                    append t[k] to the list temp
19:                end fors ← s ∪ temp
20:            end if
21:        end ifj ← j + 1
22:    end while
23:    c = t[j − 1][1] * t[j − 1][2]
24:    a = t[i][1] * t[i][2]
25:    while c − a > bound do
26:        i ← i + 1
27:        a = t[i][1] * t[i][2]
28:    end while
29:    temp ← empty list
30:    for k = i to (j − 1) do
31:        append t[k] to the list temp
32:    end for
33:    s ← s ∪ temp
34:    return s
35: end procedure
36: procedure FindAllPartitions(s, maxRectangles, n)
37:    p ← {}
38:    for x ∈ s do
39:        q = SUBSET-SUM(x, n²)
40:        for y ∈ q do
41:            if size(y) ≤ maxRectangles then
42:                p ← p ∪ {y}
43:            end if
44:        end for
45:    end for
46:    return p
47: end procedure
```

5.2.2 Phase 2: Applying exact cover problem

This phase receives a collection of partitions from the previous phase. At first, these partitions are sorted based on their maximum area difference, i.e., Mondrian score in ascending order. Now each of these partitions is tested if it can fit inside an empty $n \times n$ grid. Since the partitions are sorted, even if one partition fits inside flawlessly, the process is terminated. Thus, that particular filled-up grid is the optimal solution to the $n \times n$ Mondrian problem.

The process of fitting a partition inside an empty $n \times n$ grid is more challenging than it seems. A partition usually contains multiple rectangles with various dimensions. Also, these rectangles may occupy any contiguous spaces in the grid with their original or transposed position. So, arranging these rectangles in some way so that the entire grid is covered is a complicated task and requires an exponential amount of time.

The Dancing links technique (discussed in Section 3.3.2) can alleviate this obstacle of examining a partition. However, to apply the Dancing links, a partition first must be transformed to an *exact cover problem* (discussed in Section 3.3). Thus, each partition is converted into a matrix representation of the exact cover problem, and then the Dancing links technique is applied to test the fitting of partitions.

Before moving on to the transformation process, it should be noted that a rectangle of dimensions $a \times b$, where $a \neq b$, can fit inside an empty $n \times n$ grid in $2(n-a+1)(n-b+1)$ ways. Similarly, a square of side a can fit in $(n-a+1)^2$ ways. Thus, a 2×3 rectangle can fit inside a 3×3 empty grid in $2(3-2+1)(3-3+1) = 4$ ways, as illustrated in figure[5.1].

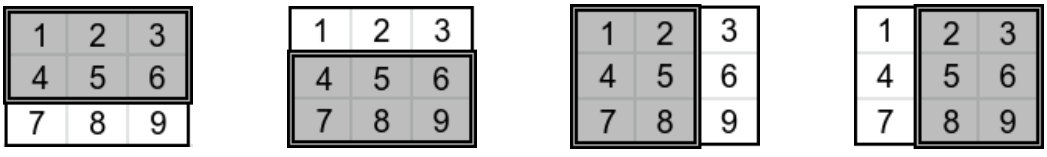


Figure 5.1: Sample fittings of a 2×3 rectangle can fit inside a 3×3 grid.

Consider a partition P of a $n \times n$ grid that contains m rectangles of dimensions $(a_1 \times b_1), (a_2 \times b_2), \ldots, (a_m \times b_m)$. Now, the first rectangle can cover the grid in $2(n - a_1 + 1)(n - b_1 + 1)$ ways. Similarly, the second rectangle cover in $2(n - a_2 + 1)(n - b_2 + 1)$ ways. Therefore, all the rectangles in P can cover the grid combinedly in $2\sum_{i=1}^{m}(n - a_r + 1)(n - b_r + 1)$ ways. Let's call this quantity r.

$$r = 2\sum_{i=1}^{m}(n - a_r + 1)(n - b_r + 1)$$

There are a total n^2 number of small boxes in the $n \times n$ grid. Imagine a matrix with $n^2 + m$ columns, one for each of the small boxes of the grid and one for each of the rectangles from P. Construct all possible rows representing a way to place a $c \times d$ rectangle on the grid; each row contains a 1 in the column identifying the rectangle, and cd 1s in the columns identifying the small boxes it covers or 0 otherwise. Repeat this task for all the m rectangles from P. So there are precisely r number of rows in the matrix. Hence this matrix constitutes a valid exact cover problem.

When applied to this matrix, if a solution exists, the Dancing links technique obtains exactly m number of rows because each of these rows must have exactly a 1 in one of the columns representing the rectangles. Furthermore, each obtained row specifies a rectangle from the partition P and the grid boxes it covers. Therefore, the positions of rectangles are non-overlapping and also cover up the entire grid. Additionally, these rectangles are not pairwise congruent due to the phase 1 construction. Hence, these rows form a real solution to the Mondrian problem and provide a geometric view of the covered grid.

An example of the process is illustrated in Figure 5.2. Here, the grid size is 3×3, and the partition is $\{(1, 3), (2, 3)\}$. The rectangles can cover the grid in 6 and 4 ways. So the matrix has the $9 + 2 = 11$ columns and $6 + 4 = 10$ rows. All the possible ways 2×3 rectangle can fit inside the grid are expressed in the first four rows with the columns corresponding to the covered boxes marked as 1. Also, all these rows have 1 in column R_1, which represents this rectangle. Similarly, the last

six rows represent the possible ways 1×3 rectangle can fit with R_2 as 1. The second and fourth row together can form a solution to the 3×3 Mondrian problem with a score of $6 - 3 = 3$.

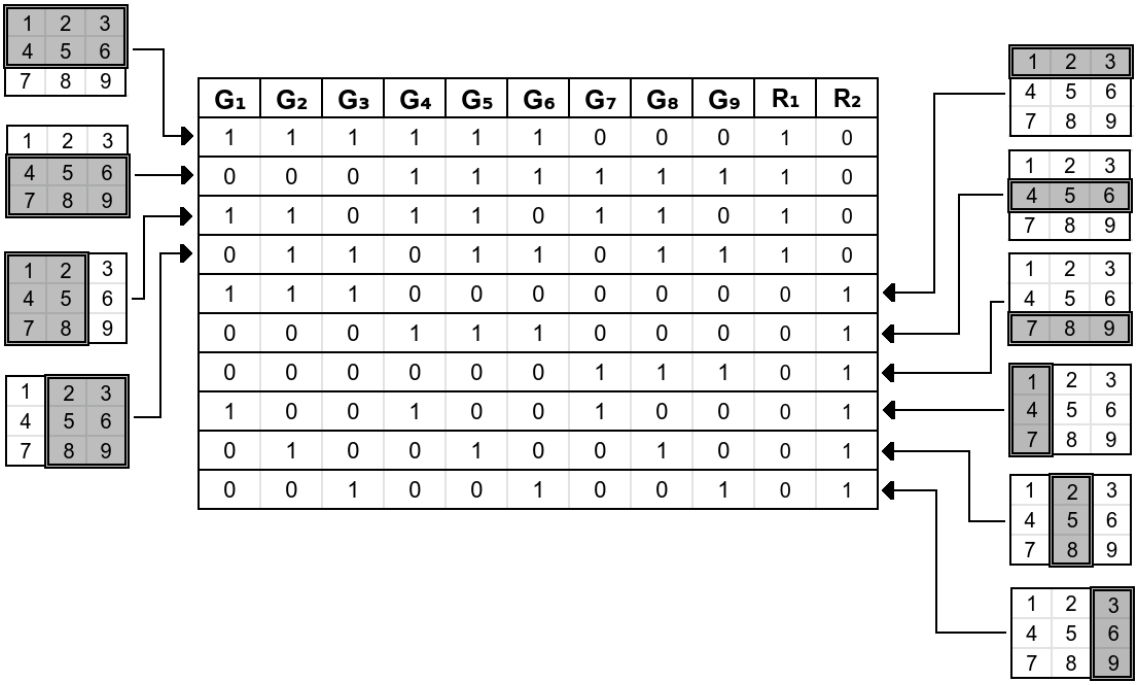


Figure 5.2: Sample transformation of a partition to exact cover problem using matrix representation. The G columns specify each box the grid, whereas R columns designate rectangles of the partition.

5.2.3 Computational challenge

As discussed in the previous sections, the existing approach engages two challenging obstacles; both are NP-complete. Consequently, the approach becomes exponential in nature. The first stage delivers multiple partitions. This number multiplies as the grid size increases. Moreover, the next stage requires exponential time to check each partition and continues its examination until a proper partition is encountered. Thus, the testing process is quite cumbersome. Therefore, this approach may not offer any solution for large grid sizes.

5.3 Proposed method description

In Section 4, it is portrayed that the initial states are generated by using random rectangles, and later a state space search algorithm like A* is employed to improve scores by exploring the state space. The outcome of the search process is dependant on the original state. So different initial states may lead to diverse regions of the state space. Therefore, instead of spawning a single state, numerous introductory states are prepared. These states are stored in the priority queue based on the scores. For experiment purposes, 500 states are constructed initially.

The parameter *max-depth* prevents protracted traversal of state-space along any branch, not aiding in improving the score. This parameter can neither be kept very large nor very small. Large values lead to excessive growth of branches, and small values restrict broad exploration. So it is usually assigned an intermediate value and specifically 100 in the experiment.

Another parameter, *max-size*, limits the size of the priority queue. It restrains the number of states considered to be explored in future so that only good ones are stored in the queue. It is generally kept as a large number to accommodate as many states as possible. It is assigned as 100000 in the experiment, which shows the best result. The result of the experiment is presented in the next section.

Chapter 6

Results and Further Analysis

6.1 Overall result

In the earlier Section 4.3, the process of generating an initial state is discussed with an example. The following section showcases some state operations which may improve the scores of states. Lastly, Section 4.5 describes how state operations are applied consecutively on different states, and gradually, a tree of states is built that grows in size.

Figure 6.1 illustrates how the scores are getting improved over time. Each state in the figure is a resultant of a particular state operation on the preceding state. The first one is the initially generated state, as shown in Section 4.3, whereas the last is the best in terms of score. One important point to note is that multiple operations can be executed in each state, leading to the formation of a tree. Here only the path from the initial state to the best state is depicted for clarity purpose.

Consider the initial 12×12 state in Figure 6.1a with a score of 50, which is below par than the trivial score for a 12×12 grid. Now the split operation is applied to it, which breaks the rectangle with marked 8 into two smaller rectangles. So, the largest area is reduced to 36, which improves the score significantly. The merge operations may yield a better score of 24 ($\because 6 * 6 - 6 * 2 = 24$) from the newborn state by combining the rectangles marked with $4, 6$ and $5, 3$. Nonetheless, the split operation is employed again because it would eventually guide to good scores. So greedily

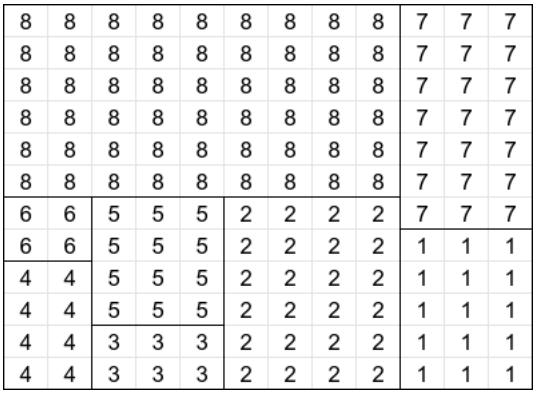
choosing the next visiting state is not the correct way. So, search algorithms like A* are considered because it is a hybrid of both past and future. Table 6.1 contains all necessary information about the states of Figure 6.1, like which state operations are used, the participating rectangles, improvement of scores and many others.

The best score found by the proposed system is 8, which is very close to the optimal score of 7. There are 10 rectangles used in the optimal 12×12 grid solution, whereas the best solution found by the proposed system utilizes only 4 rectangles. Also, the proposed system finds the solution faster, intuitively and employs fewer rectangles, which can also be considered an additional constraint. Please note that all state operations have been used to reach the best solution.

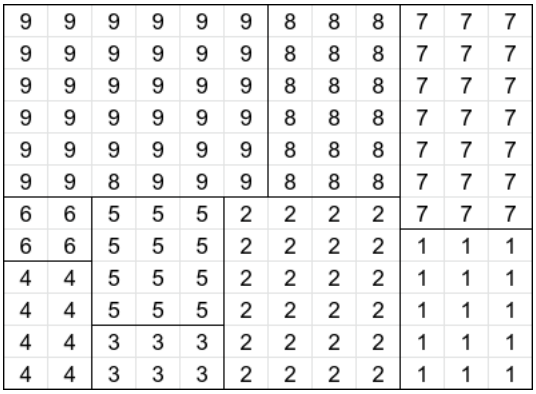
Parent State				**State Operation**			**Child State**			
State	Largest area	Smallest area	Score	Operation	Participating rectangles	Resultant rectangles	State	Largest area	Smallest area	Score
(a)	54	4	50	Split	8	8, 9	(b)	36	4	32
(b)	36	4	32	Split	9	9, 10	(c)	30	4	26
(c)	30	4	26	Split	9	9, 11	(d)	24	4	20
(d)	24	4	20	Merge-Split	2, 3	2, 3	(e)	21	4	17
(e)	21	4	17	Merge	4, 6	4	(f)	21	6	15
(f)	21	6	15	Merge	8, 10	8	(g)	24	10	14
(g)	24	10	14	Split-Merge	4, 5, 11	4, 5	(h)	24	14	10
(h)	24	14	10	Merge	1, 7	7	(i)	36	14	22
(i)	36	14	22	Merge-Split	8, 9	8, 9	(j)	36	8	28
(j)	36	8	28	Split-Merge	2,3, 5	2, 5	(k)	36	8	28
(k)	36	8	28	Merge	2, 8	8	(l)	36	16	20
(l)	36	16	20	Merge	4, 5	4	(m)	40	32	8

Table 6.1: Improvement of Mondrian scores by applying successive state operations on 12×12 states.

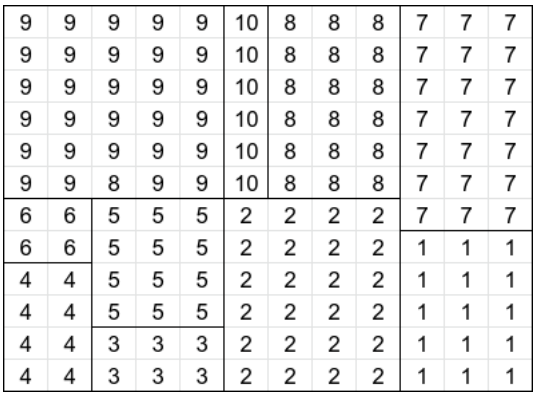
Now, the performance of the proposed approach is compared against the existing approach for different grid sizes and recorded in Table 6.2. For a $N \times N$ grid, the parameters *bound* and *maximum number of rectangles*, mentioned in Section 5.2.1, are assigned as N because optimal Mondrian scores are usually less than N and contain less than N rectangles. Since the proposed system generates the initial state randomly, the results shown in Table 6.2 are computed over multiple iterations for the same N.



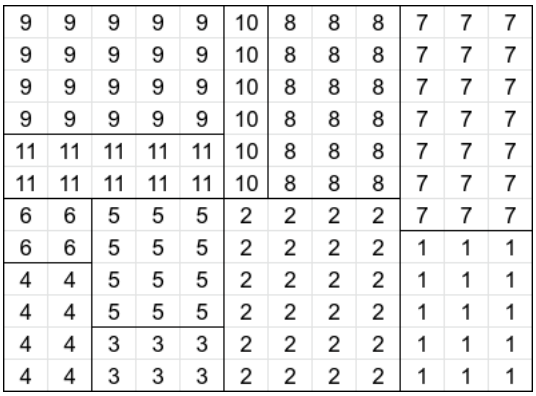
(a) Score = $(6 * 9) - (2 * 2) = 50$



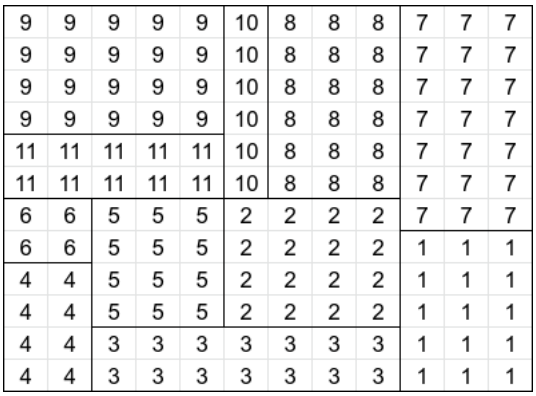
(b) Score = $(6 * 6) - (2 * 2) = 32$



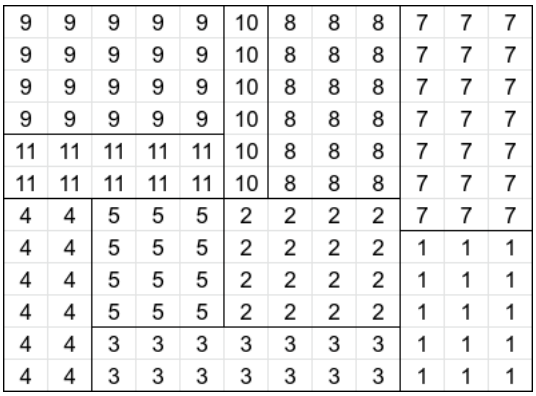
(c) Score = $(6 * 5) - (2 * 2) = 26$



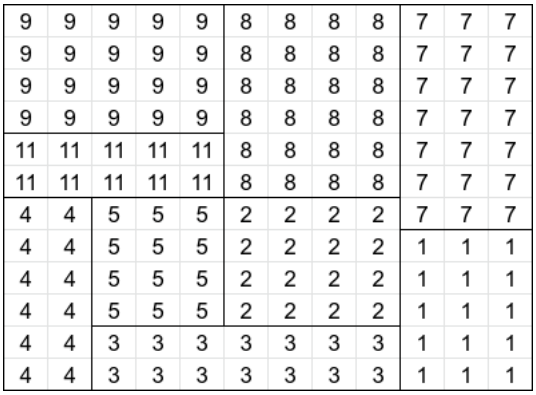
(d) Score = $(6 * 4) - (2 * 2) = 20$



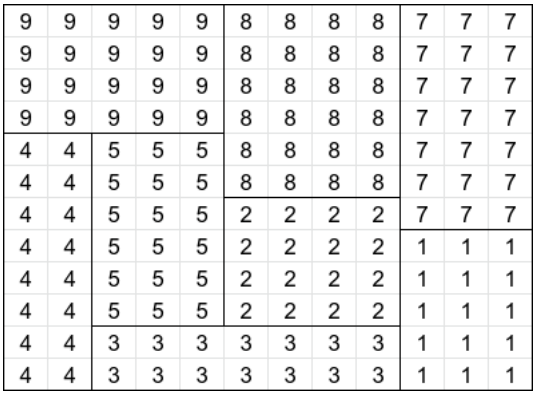
(e) Score = $(7 * 3) - (2 * 2) = 17$



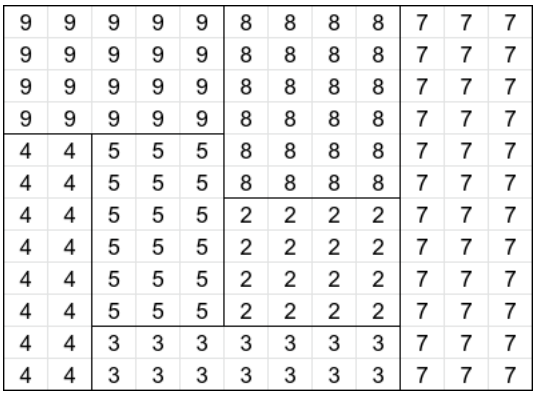
(f) Score = $(7 * 3) - (6 * 1) = 15$



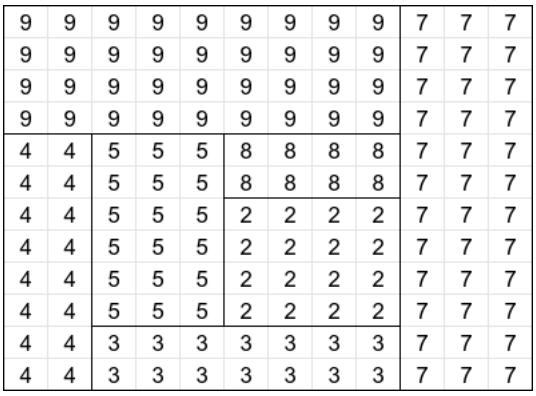
(g) Score = $(6 * 4) - (2 * 5) = 14$



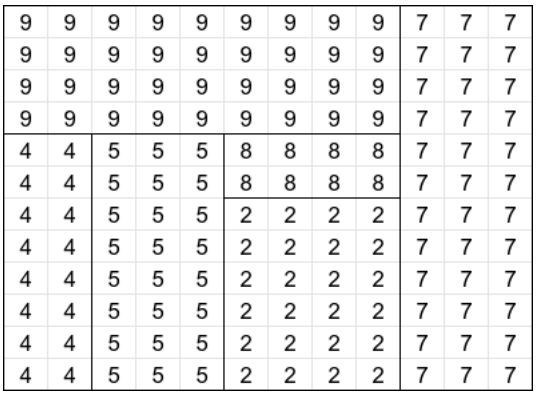
(h) Score = $(6 * 4) - (2 * 7) = 10$



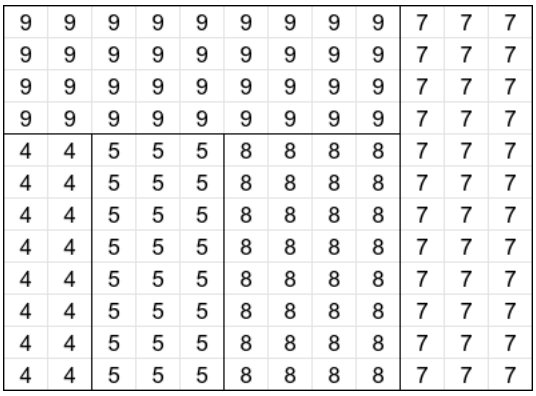
(i) Score $= (12 * 3) - (2 * 7) = 22$



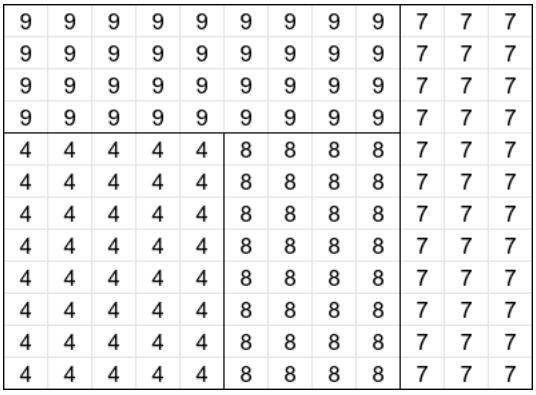
(j) Score $= (12 * 3) - (2 * 4) = 28$



(k) Score $= (12 * 3) - (2 * 4) = 28$



(l) Score $= (12 * 3) - (8 * 2) = 20$



(m) Score $= (8 * 5) - (8 * 4) = 8$

Figure 6.1: Improvement of Mondrian scores among various 12×12 states after applying state operation successively.

For small values of N, the scores are exactly the same. As N increases, the difference between the scores slightly increase, but not far beyond. The ratio of scores somewhat lies between $1, 4$. However, the time taken to compute the scores differ in a colossal margin. Although the ratio of computation times stay near 1 for smaller N, the ratio grows in leaps and bounds as N crosses 17. For N less than 50, the system computes the best score within 1 minute. On the other hand, the brute force approach takes almost 42 hours (156494 seconds) to obtain the score of grid size $N = 31$ due to its exponential nature. Moreover, it cannot retrieve scores for N greater than 50 within 60 hours. In contrast, the system acquires non-trivial scores in a span of 5 minutes for larger values of N.

Thus, the proposed approach may not obtain an optimal Mondrian score, but it can produce a non-trivial solution, even for large grid sizes, way faster than the existing brute force approach.

N	Existing Approach			Proposed approach			Comparison		
	Score [S1]	No. of rectangles	Time (Sec) [T1]	Score [S2]	No. of rectangles	Time (Sec) [T2]	Score difference [S2-S1]	Score ratio [S2 : S1]	Time ratio [T1 : T2]
3	2	3	<1	2	3	<1	0	1	∼1
4	4	4	<1	4	4	<1	0	1	∼1
5	4	3	<1	4	3	<1	0	1	∼1
6	5	5	<1	5	5	<1	0	1	∼1
7	5	5	<1	5	5	<1	0	1	∼1
8	6	6	<1	6	6	<1	0	1	∼1
9	6	3	<1	6	3	<1	0	1	∼1
10	8	6	<1	8	6	<1	0	1	∼1
11	6	8	1	6	8	<1	0	1	∼1
12	7	10	1	8	4	<1	1	1.14	∼1
13	8	6	2	10	8	1	2	1.25	2
14	6	6	2	8	6	2	2	1.33	1
15	8	5	4	8	5	3	0	1	1.33
16	8	8	6	10	11	4	2	1.25	1.5
17	8	12	91	12	11	2	4	1.5	45.5
18	8	10	116	12	11	3	4	1.5	38.67
19	8	10	106	15	16	13	7	1.88	8.15
20	9	10	557	15	12	19	6	1.67	29.32
21	9	10	231	15	10	4	6	1.67	57.75
22	9	12	1723	15	14	9	6	1.67	191.44
23	8	7	176	16	13	6	8	2	29.33
24	9	11	1219	19	14	4	10	2.11	304.75
25	10	14	2076	20	14	19	10	2	109.26
26	9	13	1631	21	19	9	12	2.33	181.22
27	10	16	2329	21	18	17	11	2.1	137
28	9	15	2107	19	18	11	10	2.11	191.55
29	9	11	1386	22	17	10	13	2.44	138.6
30	11	15	2463	24	17	7	13	2.18	351.86
31	11	19	156494	25	23	8	14	2.27	19561.75
32	10	10	2914	24	21	4	14	2.4	728.5
40	12	14	5138	36	20	27	24	3	190.3
50	13	21	191341	42	32	78	29	3.23	2453.09
60	-	-	-	54	36	67	-	-	-
70	-	-	-	74	41	161	-	-	-
80	-	-	-	116	37	175	-	-	-
90	-	-	-	121	43	243	-	-	-
100	-	-	-	155	36	152	-	-	-
150	-	-	-	270	64	208	-	-	-

Table 6.2: Comparison of brute force approach with the proposed system is shown. The scores for $N > 50$ cannot be computed within 60 hours of time by the existing approach, but is computed by the recommended approach within 5 minutes.

6.2 Ablation analysis

At every stage of the search algorithm, states go through the 4 state operations. The impact of an operation can be measured by inspecting how much it reduces scores

for different states. An experiment is conducted to assess the relative performances of each operation. Firstly, the best scores for different grid sizes are recorded where all operations are employed combined. Next, again the best scores for different grid sizes are measured; but unlike last time, one operation is skipped, and others are used. This process is repeated for all the operations. Now, there are five sets of best scores for every individual size, where one is computed with all operations, and the rest are computed omitting one operation at a time.

The significance of an operation can be quantified by checking the scores without that particular operation. If the deviation of these scores is pretty high from other scores, then the operation has a more significant impact on the system. If low, then the importance of the operation is not immense. The purpose is to rank the operations according to their influence.

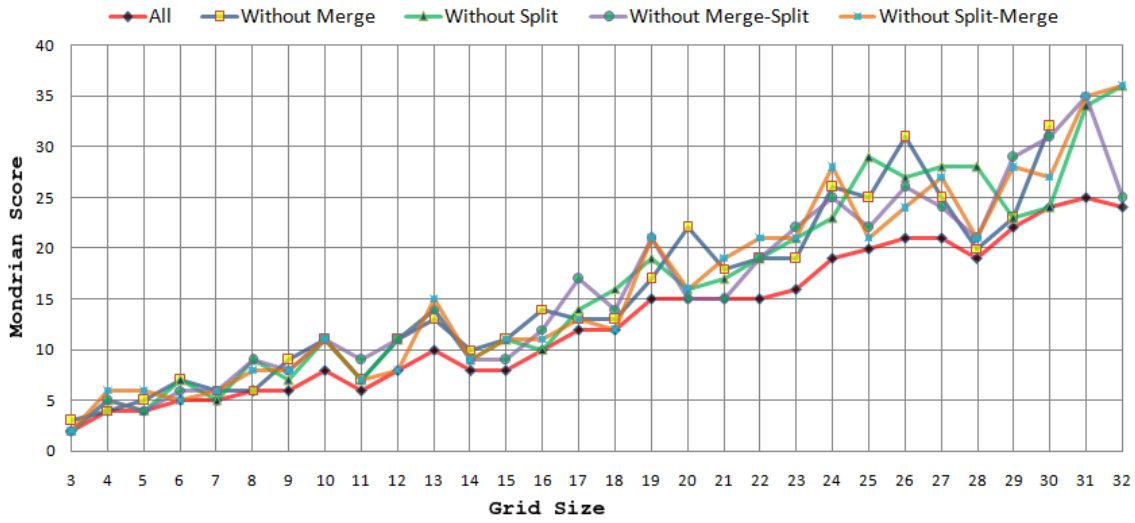


Figure 6.2: Comparison of relative performance of the state operations.

The result of the above experiment is presented in Figure 6.2. The lines showcase the five sets of scores. The line which maps to the scores with all operations combined stays lowest. The reason is pretty simple. The other lines crisscross each other frequently, even sometimes intersect the lowermost line of the best scores, but never cross it. So no clear ranking can be observed, and it is fair to say all the operations contribute somewhat equally.

Chapter 7

Conclusion and Future Scope

According to the study results, it is clear that the proposed approach is way more efficient than the existing brute force approach. The former can compute a non-trivial score for a large grid size quickly, whereas the latter takes a massive amount of time for the same. The effectiveness of both approaches is comparable. Therefore, it is a trade-off between efficiency and effectiveness. Moreover, it can be considered a fast way to measure the Mondrian score by sacrificing the optimality a little bit. Also, it can provide some idea about the optimal score and be treated as upper bound while applying the brute force approach, reducing time consumption.

So far now, the valid states which preserve the primary constraints are dealt with, and the state space is explored only for those feasible states. However, restricting the system to only feasible states is causing half the state space to be still unknown and undiscovered. This unexplored portion of state space consists of infeasible (or invalid) states that do not follow primary constraints. The future goal is to search this unknown region, eventually leading to valid states with significantly improved scores, which is not reachable now.

Some other variants of the core problem can also be defined by introducing another goal. For example, another goal can be to pack as many rectangles as possible to maximize the ratio of the number of rectangles to the score. Some modifications may be required to the current system to adapt to the new rules and

goals.

Bibliography

[1] Hannes Bassen. *Further Insight into the Mondrian Art Problem*. URL: https://nyccami.org/wp-content/uploads/2017/12/Bassen-Further-Insight-into-the-Mondrian-Art-Problem.pdf.

[2] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[3] C. Dalfó, M. A. Fiol, and N. López. *New results for the Mondrian art problem*. 2020. eprint: 2007.09639. URL: https://arxiv.org/abs/2007.09639.

[4] Rina Dechter and Judea Pearl. "Pearl, J.: Generalized best-first search strategies and the optimality of A*. Journal of the ACM (JACM) 32, 505-536". In: *J. ACM* 32 (July 1985), pp. 505–536. DOI: 10.1145/3828.3830.

[5] George W. Ernst and Allen Newell. "Some Issues of Representation in a General Problem Solver". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 583–600. ISBN: 9781450378956. DOI: 10.1145/1465482.1465579. URL: https://doi.org/10.1145/1465482.1465579.

[6] G. Hamilton. *Mondrian Art Puzzles*. URL: https://mathpickle.com/project/mondrian-art-puzzles/.

[7] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136.

[8] Richard M. Karp. "Reducibility among Combinatorial Problems". In: *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*. Ed. by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. Boston, MA: Springer US, 1972, pp. 85–103. ISBN: 978-1-4684-2001-2. DOI: 10.1007/978-1-4684-2001-2_9. URL: https://doi.org/10.1007/978-1-4684-2001-2_9.

[9] Jon Kleinberg and Eva Tardos. *Algorithm Design*. USA: Addison-Wesley Longman Publishing Co., Inc., 2005. ISBN: 0321295358.

[10] Donald E. Knuth. "A generalization of Dijkstra's algorithm". In: *Information Processing Letters* 6.1 (1977), pp. 1–5. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(77)90002-3. URL: https://www.sciencedirect.com/science/article/pii/0020019077900023.

[11] Donald E. Knuth. *Dancing Links*. URL: https://arxiv.org/abs/cs/0011047.

[12] A. Newell, J. C. Shaw, and H. A. Simon. "Empirical Explorations of the Logic Theory Machine: A Case Study in Heuristic". In: *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*. IRE-AIEE-ACM '57 (Western). Los Angeles, California: Association for Computing Machinery, 1957, pp. 218–230. ISBN: 9781450378611. DOI: 10.1145/1455567.1455605. URL: https://doi.org/10.1145/1455567.1455605.

[13] Nils J. Nilsson. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill Pub. Co., 1971. ISBN: 0070465738.

[14] Cooper O'Kuhn. *The Mondrian Puzzle: A Connection to Number Theory*. URL: https://arxiv.org/abs/1810.04585. (accessed: 16.03.2021).

[15] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. USA: Addison-Wesley Longman Publishing Co., Inc., 1984. ISBN: 0201055945.

[16] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. `https://cs.calvin.edu/courses/cs/344/kvlinden/resources/AIMA-3rd-edition.pdf`(visited 2021-06-01). Pearson Education, Inc., 1995.

[17] Herbert A. Simon and Allen Newell. "Heuristic Problem Solving: The Next Advance in Operations Research". In: *Operations Research* 6.1 (1958), pp. 1–10. ISSN: 0030364X, 15265463. URL: `http://www.jstor.org/stable/167397`.