Tiling Problem Using Two Different Approaches

Mohamed Hesham
Faculty Of Computer Science
Misr International University
mohamed2300428@miuegypt.edu.eg

Ramy Slait
Faculty Of Computer Science
Misr International University
ramy2301480@miuegypt.edu.eg

Antoni Ashraf
Faculty Of Computer Science
Misr International University
antoni2304892@miuegypt.edu.eg

Nabil Ramy
Faculty Of Computer Science
Misr International University
nabil2300799@miuegypt.edu.eg

Seif Makled
Faculty Of Computer Science
Misr International University
seif2304145@miuegypt.edu.eg

Ashraf Abdel Raouf
Faculty Of Computer Science
Misr International University
ashraf.raouf@miuegypt.edu.eg

***Abstract*—The tiling problem is to determine the number of ways in which a 2 × n rectangular grid can be tiled entirely using 2 × 1 dominoes. In this study, two algorithmic strategies are compared: a recursive divide-and-conquer solution and brute force backtracking. Both methods are explained and instantiated, the computational complexity issues at stake are studied, and these methods' efficiency are assessed empirically. Our findings show that even though both methods have exponential time complexity for this problem, the divide and conquer (recurrence) method tends to be easier to implement and understand. While the recursive method makes use of the structure of the problem to provide a simpler solution, brute force encompasses all the solutions by searching through all the combinations. We explain each method's weakness and advantage in relation to the tiling problem.**

***Index Terms*—Tiling problem, brute force, divide and conquer**

I. Introduction

One of the best-known computer science and combinatorics problems is the "tiling problem": In how many manners can one tile a $2 \times n$ grid with 2×1 dominoes standing vertically or horizontally, without overlapping and without leaving a gap between them? The problem is a trivial illustration of recursive problem solution and has applications in physics, mathematics, and algorithm construction [1], [2].

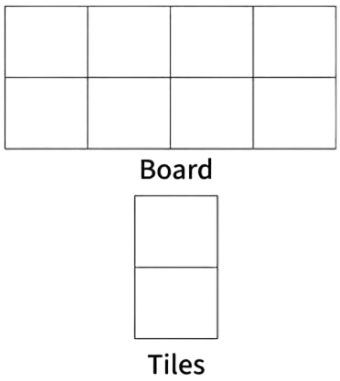

Fig. 1: Example of domino tiling on a 2×4 grid.

There are a large number of tilings with extremely rapid growth with respect to grid size n, and exhaustive listing becomes computationally costly. Recursive (divide and conquer) methods and brute force backtracking are two typical methods of reducing this issue [3], [4]. By systematic evaluation of all possible placements, the brute force method ensures that all possible tiles are evaluated. Through defining the solution to an $n \times n$ grid in terms of smaller solutions, the recursive solution employs the problem's structure [1], [2]

The recursive approach tends to be simpler and more straightforward to program, while brute force offers an easy means of listing all of the solutions, although both approaches are of exponential time complexity for the problem. We here outline, programmed, and compared the two strategies and evaluated their advantages and disadvantages.

II. Brute Force Approach

The domino tiling problem brute force algorithm uses exhaustive search to list every possible arrangement of dominoes on a $2 \times n$ grid [3]. The approach is certain to find an optimal solution through systematic backtracking and verification of every possible solution. Brute force offers an explicit representation of the problem structure and a baseline from which more advanced algorithms can be compared, although it is costly computationally for larger grids [4].

The overall idea of this solution is to recursively put dominoes in all orientations (horizontal and vertical) in all empty cells with the constraint that each domino should cover exactly two adjacent cells without affecting the already put dominoes.

A. Approach

The brute force method employs a recursive backtracking algorithm that tests all feasible solutions. The algorithm keeps a $2 \times n$ boolean matrix to mark filled squares and to lay down dominoes.

Key components of the approach include the following.

- **State Representation**: A $2 \times n$ boolean matrix where `true` indicates an occupied cell

- **Decision Points**: At each empty cell, attempt both horizontal and vertical domino placements
- **Constraint Checking**: Ensure dominoes don't exceed grid boundaries or overlap existing placements
- **Backtracking**: Undo placements when no valid continuation exists

Starting in cell location $(0, 0)$, the algorithm moves through all cells and over all cells that are already filled. It recursively solves the subproblem with the remaining cells by attempting to place a domino in each of the two orientations for each empty cell.

B. The Algorithm

The implementation consists of several components working together to ensure all valid tilings. Here, We break down the algorithm into logical components:

Function Header and Parameters:

The core recursive function is defined as follows:

Algorithm 1 Function Signature

procedure PLACEDOMINO($row, col, grid$)
    row: current row position to examine
    col: current column position to examine
    $grid$: reference to $2 \times n$ boolean matrix tracking cells
end procedure

The function takes three parameters: the current position (row, col) and a reference to the grid. The grid variable is passed by reference to avoid copy overhead and also to retain state changes across recursive calls.

Global Variables:

The algorithm uses the following global variables:

Algorithm 2 Global Variables

```
const int ROWS = 2          ▷ Fixed grid height
int tilingCount = 0         ▷ Counter for valid tilings
```

Cell Navigation Logic:

The algorithm efficiently finds the next empty cell to process. This navigation ensures organized exploration:

Algorithm 3 Finding Next Empty Cell

while $row < rows$ **and** $grid[row][col]$ **do**
    $col \leftarrow col + 1$
    if $col = cols$ **then**
        $col \leftarrow 0$
        $row \leftarrow row + 1$
    end if
end while

This loop skips over already occupied cells, moving left to right within each row, then advancing to the next row.

Base Case Detection:

The recursive process ends when every cell has been processed, indicating that a complete and valid tiling has been achieved.

Algorithm 4 Complete Tiling Detection

if $row = rows$ **then**
    $tilingCount \leftarrow tilingCount + 1$
    return
end if

Horizontal Domino Placement:

For each empty cell, the algorithm first attempts horizontal placement if the adjacent cell is available:

Algorithm 5 Horizontal Placement

if $col + 1 < cols$ **and** $\neg grid[row][col + 1]$ **then**
    $grid[row][col] \leftarrow true$
    $grid[row][col + 1] \leftarrow true$
    PLACEDOMINO($row, col, grid$)
    $grid[row][col] \leftarrow false$ ▷ Backtrack
    $grid[row][col + 1] \leftarrow false$
end if

Vertical Domino Placement:

Similarly, the algorithm attempts vertical placement if the cell below is available:

Algorithm 6 Vertical Placement

if $row + 1 < rows$ **and** $\neg grid[row + 1][col]$ **then**
    $grid[row][col] \leftarrow true$
    $grid[row + 1][col] \leftarrow true$
    PLACEDOMINO($row, col, grid$)
    $grid[row][col] \leftarrow false$ ▷ Backtrack
    $grid[row + 1][col] \leftarrow false$
end if

Main Function:

The main function initializes the grid and starts the recursive function call:

Algorithm 7 Main Function

procedure MAIN
    input n ▷ number of columns
    $grid \leftarrow$ new boolean matrix of size $2 \times n$ initialized to `false`
    PLACEDOMINO($0, 0, grid$)
    output "Possible tilings: " + $tilingCount$
end procedure

C. Complexity Analysis

Time Complexity: It uses at most two options of placements per empty space, which is vertical and horizontal, making the running time exponential in $O(2^n)$.

Space Complexity: The algorithm's overall space complexity uses $O(n)$ space for the recursion stack and $O(n)$ space for the $2 \times n$ grid parameter.

D. Step-by-Step Trace for 2×2 Grid

To illustrate the process, consider the execution trace for a 2×2 grid:

TABLE I: Execution trace for 2×2 grid

Step	Action	Grid State	Result
1	Initial state	$\begin{pmatrix} F & F \\ F & F \end{pmatrix}$	Start at (0,0)
2	Try horizontal at (0,0)	$\begin{pmatrix} T & T \\ F & F \end{pmatrix}$	Move to (1,0)
3	Try horizontal at (1,0)	$\begin{pmatrix} T & T \\ T & T \end{pmatrix}$	Solution 1 found
4	Backtrack to (0,0)	$\begin{pmatrix} F & F \\ F & F \end{pmatrix}$	Try vertical
5	Try vertical at (0,0)	$\begin{pmatrix} T & F \\ T & F \end{pmatrix}$	Move to (0,1)
6	Try vertical at (0,1)	$\begin{pmatrix} T & T \\ T & T \end{pmatrix}$	Solution 2 found

Note: T=True (occupied), F=False (empty)

III. Divide and Conquer Approach

The divide and conquer technique of the domino tiling problem uses a recursive method to reduce the problem into subproblems [1], [2]. Unlike the brute force technique of keeping grid states explicitly, it utilizes patterns for the manner dominoes can be arranged to estimate the number of tiling arrangements [2].

The basic principle here is that there are only two possible methods of starting to occupy the grid in any given column position: either one vertical domino or two horizontal dominoes stacked on top of each other. The realization creates a basic recursive breakdown [1], [5].

A. Approach

Based on a recurrence relation with knowledge of the problem structure, the divide and conquer approach applies a recursive approach. The method expresses the number of tiling arrangements in terms of the board width rather than explicitly tracing the grid.

The main components of the approach are:

- **Problem Decomposition**: Break the $2 \times n$ grid into smaller subproblems
- **Choice Analysis**: Investigate the two potential domino placement strategies at each column
- **Recurrence Relation**: Define the solution in terms of smaller examples of the same problem
- **Base Cases**: Define the base cases which terminate the recursion

The algorithm takes the leftmost column of the grid and calculates the ways to fill it, then recursively finds the remainder of the subgrid. This minimizes backtracking and explicit management of grid state [2], [3].

B. The Algorithm

The implementation consists of a single recursive function that captures the problem's structure. We break down the algorithm into its essential components:

Function Header and Parameters:

The core recursive function is defined as follows:

Algorithm 8 Function Signature

```
procedure DIVIDEANDCONQUERTILING(n)
    n: the number of columns in the 2 × n grid
    returns: integer count of possible tilings
end procedure
```

The function takes a single parameter representing the grid width and returns the total count of valid tilings.

Base Cases:

The recursion requires well-defined base cases to terminate properly:

Algorithm 9 Base Cases

```
if n = 0 then
    return 1    ▷ Empty grid has exactly one way to tile (do nothing)
end if
if n = 1 then
    return 1    ▷ Only one vertical domino fits in a 2 × 1 grid
end if
```

Choice Analysis:

For any $2 \times n$ grid, we have exactly two ways to begin tiling:

Algorithm 10 Tiling Choices

```
Choice 1: Place one vertical domino in the first column
        ▷ This covers both cells in column 1, leaving a 2 × (n − 1) subproblem
Choice 2: Place two horizontal dominoes (one in each row)
    ▷ This covers columns 1 and 2, leaving a 2 × (n − 2) subproblem
```

Recurrence Relation:

This analysis leads directly to the recurrence relation:

Algorithm 11 Recurrence Formula

```
T(n) = T(n − 1) + T(n − 2)
    ▷ Where T(n) represents the number of ways to tile a 2 × n grid
```

Algorithm 12 Divide and Conquer Tiling Function

```
procedure DIVIDEANDCONQUERTILING(n)
    if n = 0 then
        return 1
    end if
    if n = 1 then
        return 1
    end if
    return DIVIDEANDCONQUERTILING(n − 1) + DIVIDEANDCONQUERTILING(n − 2)
end procedure
```

Core Recursive Function:

The main recursive function implements the recurrence:

Main Function Implementation:

The main function demonstrates how to use the recursive approach:

Algorithm 13 Main Function for Divide and Conquer

```
procedure MAIN
    input n                                  ▷ number of columns
    result ← DIVIDEANDCONQUERTILING(n)
    output "Number of ways to tile 2 × " + n + " grid: " + result
end procedure
```

C. Complexity Analysis

Time Complexity: The time complexity of $O(2^n)$ is seen for the recursive solution because each function call is divided into two calls recursively, creating a binary tree of depth n. The actual running time will vary with the overlapping subproblems.

Space Complexity: The maximum space utilized by the algorithm is $O(n)$ for the recursion stack because the deepest recursion is n.

D. Step-by-Step Trace for $n = 4$

To illustrate the divide and conquer process, consider the execution trace for a 2×4 grid:

TABLE II: Execution trace for 2×4 grid using divide and conquer

Call	Function Call	Recursive Breakdown	Result
1	$T(4)$	$T(3) + T(2)$	$3 + 2 = 5$
2	$T(3)$	$T(2) + T(1)$	$2 + 1 = 3$
3	$T(2)$	$T(1) + T(0)$	$1 + 1 = 2$
4	$T(1)$	Base case	1
5	$T(0)$	Base case	1

IV. Comparison and Results

Both methods get the problem of domino tiling solved effectively; However, they get implemented differently and are of different performance characteristics. In this section here, it is explicitly shown how the divide and conquer and brute force methods are compared [4], [5].

A. Performance Analysis

TABLE III: Algorithmic Characteristics Comparison

Aspect	Brute Force	Divide and Conquer
Time Complexity	$O(2^n)$	$O(2^n)$
Space Complexity	$O(n)$	$O(n)$
Implementation	Complex backtracking logic	Simple recursive formula
Output	Enumerates all solutions	Counts total solutions
Optimization	Limited pruning possible	Limited optimization

B. Strengths and Limitations

Brute Force Approach:

Strengths:

- Finds all possible tilings
- Easy to understand and modify

Limitations:

- Very slow for large n
- Uses a lot of memory
- Harder to debug for big grids

Divide and Conquer Approach:

Strengths:

- Simple and compact code
- Low memory usage
- Relates to Fibonacci numbers

Limitations:

- Only gives the count, not the patterns
- Still slow without optimization
- Recurrence may be less obvious

V. Conclusion

In this paper, we have contrasted and compared two approaches to the domino tiling problem: brute force backtracking and divide and conquer. The brute force approach, while exhaustive and capable of generating all possible tiling arrangements, is handicapped by its resource hungry and excessive memory requirements, making it inappropriate for large grid sizes. On the other hand, the divide and conquer approach employs only mathematical recurrence, which gives an improved time complexity in calculating the count of tilings without actually counting the arrangements themselves.

We find that, for all but the most practical problems, the recursive divide and conquer method should be preferred due

to its simplicity and efficiency [1], [2]. However, the brute force method does hold merit where exhaustive listing of all tilings is a required prerequisite. Ultimately, the selection of method should be guided by the specific demands of the problem at hand, balancing the trade-offs of completeness against computability [3], [5].

REFERENCES

[1] D. E. Knuth, "The Art of Computer Programming, Volume 4A: Combinatorial Algorithms," Addison-Wesley Professional, 2011.

[2] R. L. Graham, D. E. Knuth, and O. Patashnik, "Concrete Mathematics: A Foundation for Computer Science," Addison-Wesley Professional, 1994.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms, Third Edition," MIT Press, 2009.

[4] S. S. Skiena, "The Algorithm Design Manual, Second Edition," Springer, 2008.

[5] J. Kleinberg and E. Tardos, "Algorithm Design," Addison-Wesley, 2005.