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 goal of the tiling problem is to determine how many ways there are to cover a 2 × n rectangular grid entirely with 2 × 1 dominoes. This study contrasts two algorithmic strategies: a recursive divide and conquer strategy and brute force backtracking. We outline and put into practice both approaches, examine the computational difficulties involved, and assess their effectiveness experimentally. Our findings demonstrate that while both strategies have exponential time complexity for this problem, the divide and conquer (recurrence) strategy is frequently easier to apply and understand. While the recursive method makes use of the problem's structure to produce a more simple solution, brute force ensures all solutions by investigating all possible combinations. We go over each method's advantages and disadvantages in relation to the tiling problem.**

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

I. Introduction

A well-known problem in computer science and combinatorics is the "tiling problem": How many different ways are there to cover a $2 \times n$ rectangular grid with 2×1 dominoes arranged vertically or horizontally, with no overlap or spaces between them? This problem is a basic illustration of recursive problem solving and has applications in physics, mathematics, and algorithm design [1], [2].

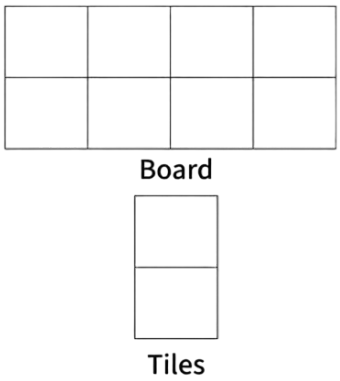

Fig. 1: Example of domino tiling on a 2×4 grid.

The number of potential tilings rapidly increases with grid length n, making exhaustive enumeration computationally costly. Recursive (divide and conquer) methods and brute force backtracking are two popular strategies for resolving this issue [3], [4]. By methodically examining every potential placement, the brute force approach ensures that every legitimate tile is taken into account. By expressing the solution for a grid of length n in terms of smaller subproblems, the recursive method makes use of the problem's structure [1], [2].

The recursive approach is frequently less complicated and simpler to implement, while brute force offers a direct method to list all solutions, even though both approaches have exponential time complexity for this problem. In this work, we outline, put into practice, and contrast both strategies while evaluating their advantages and disadvantages.

II. Brute Force Approach

The brute force method for solving the domino tiling problem uses exhaustive search to list every possible arrangement of dominoes on a $2 \times n$ grid [3]. This approach ensures that the best solution will be found by methodically going back and examining each possible arrangement. The brute force approach offers a clear understanding of the problem's structure and acts as a baseline for comparing more complex algorithms, despite being computationally costly for large grids [4].

The basic idea of this method is to recursively arrange dominoes in every conceivable orientation (horizontal and vertical) at every empty space while adhering to the requirement that each domino cover precisely two neighboring cells without overlapping with dominoes that have already been placed.

A. Approach

The brute force solution utilizes a recursive backtracking strategy that explores all possible solutions. The algorithm maintains a $2 \times n$ boolean grid to track occupied cells and to place 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 at position $(0, 0)$, the algorithm moves through each cell, avoiding those that are already occupied. It solves the remaining subproblem recursively by attempting to position a domino in each of the two possible 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 × 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 copying overhead and to maintain 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 ← col + 1
    if col = cols then
        col ← 0
        row ← 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 ← 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 ¬grid[row][col + 1] then
    grid[row][col] ← true
    grid[row][col + 1] ← true
    PLACEDOMINO(row, col, grid)
    grid[row][col] ← false                ▷ Backtrack
    grid[row][col + 1] ← 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 ¬grid[row + 1][col] then
    grid[row][col] ← true
    grid[row + 1][col] ← true
    PLACEDOMINO(row, col, grid)
    grid[row][col] ← false                ▷ Backtrack
    grid[row + 1][col] ← 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 ← new boolean matrix of size 2 × n initialized
  to false
    PLACEDOMINO(0, 0, grid)
    output "Possible tilings: " + tilingCount
end procedure
```

C. Complexity Analysis

Time Complexity: The brute force algorithm uses up to two placement options (horizontal and vertical) at each empty position, resulting in an exponential time complexity $O(2^n)$.

Space Complexity: The algorithm's overall space complexity is $O(n)$ since it 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 method for the domino tiling challenge uses a recursive technique to split the problem into smaller subproblems [1], [2]. Unlike the brute force approach that explicitly tracks grid states, this method uses patterns in how dominoes can be arranged to estimate the number of tiling configurations [2].

This method's basic rule is that there are only two ways to start filling the grid at any given column position: either with a single vertical domino or with two horizontal dominoes stacked on top of one another. This observation produces a basic recursive breakdown [1], [5].

A. Approach

Based on a recurrence relation that catches the structure of the problem, the divide and conquer solution employs a recursive approach. The method models the number of tiling configurations as a function of board length rather than clearly tracking the grid.

Key components of the approach include:

- **Problem Decomposition**: Break down the $2 \times n$ grid into smaller subproblems
- **Choice Analysis**: At each column, analyze the two possible domino placement strategies
- **Recurrence Relation**: Express the solution in terms of smaller instances of the same problem
- **Base Cases**: Define the fundamental cases that terminate the recursion

The algorithm considers the leftmost column of the grid and determines the number of ways to fill it, then recursively solves the remaining subgrid. This approach eliminates the need for explicit grid state management and backtracking [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 recursive algorithm has exponential time complexity $O(2^n)$ because each function call branches into two recursive calls, creating a binary tree of depth n. The actual runtime is determined by the number of overlapping subproblems.

Space Complexity: The algorithm uses $O(n)$ space for the recursion stack in the worst case, as the maximum depth of 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

The domino tiling problem is successfully solved by both methods, although they have different uses and performance traits. A thorough comparison between the divide and conquer and brute force strategies is given in this section [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 analyzed and compared two approaches to the domino tiling problem: brute force backtracking and the divide and conquer method. The brute force approach, while exhaustive and capable of generating all possible tiling arrangements, is limited by its high computational and memory requirements, making it impractical for large grid sizes. On the other hand, the divide and conquer approach leverages mathematical recurrence, offering a more efficient solution for counting the number of tilings, though it does not enumerate the actual arrangements.

Our results demonstrate that, for most practical scenarios, the recursive divide and conquer method is preferable due to its simplicity and efficiency [1], [2]. However, the brute

force approach remains valuable for applications where the enumeration of all possible tilings is required. Ultimately, the choice of method should be guided by the specific needs of the problem at hand, balancing the trade-offs between completeness and computational feasibility [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.