# USING THE FILL ALGORITHM TO SOLVE RECURSIVE INFORMATICS PROBLEMS: CASE STUDIES AND OPTIMIZATIONS

**Adrian RUNCEANU,** "Constantin Brâncuși" University of Târgu Jiu, ROMANIA, adrian.runceanu@e-ucb.ro
**Mihaela-Ana RUNCEANU,** "Ecaterina Teodoroiu" High College, Târgu Jiu, ROMANIA

**ABSTRACT:** The FILL algorithm which functions as a flood-fill technique in image processing serves as an efficient solution to solve various informatics problems related to matrix traversal, component counting and recursive region detection. The paper investigates FILL algorithm usage for three fundamental problems: connected component (continent) counting and value aggregation over regions and zone analysis in maps. Such problems commonly occur on competitive informatics platforms including pbinfo.ro. The paper conducts extensive case analysis alongside experimental measurements to demonstrate the effectiveness of recursive flood-fill methods while presenting timing results between DFS recursion and BFS iteration and recommending performance enhancements through pre-check marking and scanline approaches and Union-Find labeling techniques. The paper supports educational value of these problems because they help teach recursive concepts and algorithmic thinking.

**KEY WORDS:** FILL algorithm, flood-fill, recursive DFS, algorithmic recursion, problem solving in C++.

## 1. INTRODUCTION

Recursive programming serves as a basic computing technique which allows functions to invoke themselves to divide problems into smaller sub-problems for solution. This paper evaluates recursive methodology through theoretical analysis while focusing on fill algorithms in computational problem resolution. The approach known as recursion enables computers to tackle problems through solutions of identical smaller problems to find final answers. According to Niklaus Wirth "The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement." A recursive program of finite size enables the description of an unlimited number of computations without needing explicit repetition statements [1]. Recursive solutions use self-referential mechanisms to achieve repetition which generates elegant problem solutions for specific types of problems. Recursive functionality exists in all current programming languages because functions can invoke themselves from their own code. The programming language Clojure depends entirely on recursive programming because it lacks built-in looping constructs. Despite the apparent restriction to recursion only these languages demonstrate Turing completeness according to computability theory thus matching the power of imperative languages with while and for loops [1].

A recursive function needs three crucial components to work effectively. To achieve recursion the function needs to invoke itself

with different input values. The function requires a base case definition that allows it to return values without initiating additional recursive operations. The base case must be accessible after several recursive steps [2]. A recursive function requires specific elements including a proper stopping condition to avoid infinite recursion which leads to stack overflow errors that cause program crashes.

The creation of proper termination conditions stands as a vital requirement for designing recursive programs. Developers need to study the problem domain to determine how their function can produce results through direct recursion instead of additional calls [2]. The design approach leads to both correct and efficient outcomes.

## 2. TYPES OF RECURSIONS

(TNR 12 pt)

Primitive recursive functions serve as an essential group of total computable functions according to computability theory. Programs using primitive recursive functions operate through loops with defined maximum iteration counts that establish before the loop starts thus resembling "for" loops with defined counts [5]. The characteristic properties of primitive recursive functions make them especially useful for studying and building recursive algorithms. The fundamental operations of number theory and broader mathematics including addition and division and factorial calculation and the function which returns the nth prime number belong to the primitive recursive set. A computable function becomes primitive recursive when its time complexity remains below a primitive recursive function of input size [5].

Structural recursion functions as a recursive method which parallels the function of structural induction to mathematical induction [4]. Data structures such as formulas, lists and trees are best handled by this particular method. The base cases in structural recursion manage basic structures and recursive rules explain the handling of complex structures. The empty list can serve as the base case while recursive rules enable processing the head of the list and executing recursive calls on the tail [3]. List length calculation and list concatenation demonstrate structural recursion concepts through their implementation.

### 2.1. Standard Recursion

Standard recursive functions contain internal recursive calls which perform operations in addition to the call. The traditional factorial calculation performs multiplication between the current number value and the factorial outcome from the recursive function.

```
// C++
#include <iostream>
using namespace std;

 int factorial(int n) {
   // Base case
   if (n == 0 || n == 1)
      return 1;
   else
      // Recursive call
      return  n * factorial(n - 1);
}
int main () {
   int number;
   cout << "Enter a number: ";
   cin >> number;
   cout << "Factorial of " << number << " is "
<< factorial(number) << endl;
   return 0;
}
```

When we execute factorial(5) it produces the result 5! = 120 because of the implemented calculation of 5 × 4 × 3 × 2 × 1 [2]. The function includes a direct return value from the base case when n equals 0 or 1 but uses the recursive case to solve the problem by multiplying n with the factorial result of (n-1).

### 2.2. Tail Recursion

Tail recursion stands as a particular form of recursion that executes its recursive call as the last operation in the function before returning its result. A method demonstrates tail recursion when its last executed statement consists of another call to the same method [4]. Tail call optimization allows compilers and interpreters to enhance the recursive process because it eliminates the possibility of deep recursion stack overflow.

A tail recursive method requires it's called method to produce concrete results such as

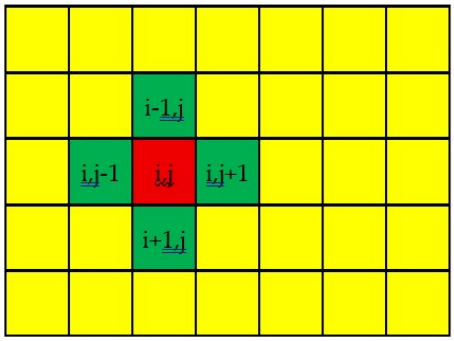
numbers or strings because void results are not acceptable [4]. Even though the recursive operation needs to be the last step executed by the method it does not need to occupy the last line in the code since it can reside inside control structures provided it represents the last executed operation.

Java and C++ programming languages enable tail recursion yet their optimization abilities differ based on the implementation. The application of recursion delivers more natural solutions than iterative constructs for particular problem domains although tail-recursive algorithms can be transformed into for loops or while loops [4].

### 2.3. Fill Algorithms: Recursive Implementation

The graph traversal algorithms of fill algorithms serve two primary applications in computer graphics and computational geometry. Flood fill algorithms serve as a well-known example to determine bounded areas in multi-dimensional arrays through their execution on graph-connected nodes. The flood fill algorithm demonstrates recursive problem-solving by visiting each connected cell in a grid and changing its color or status. The identification of neighbors for the current element demands knowledge of their coordinate positions. When the current element has the coordinates (i,j) the neighboring elements will be located at positions shown in the following image (Table 1 - we analyzed neighbors that exist in both row and column directions):

**Table 1.** Identifying the neighbors of the current element at position (i,j)




A recursive solution for a 2D grid would appear as follows in C++:

```
// C++
#include <iostream>
#include <vector>
using namespace std;
void  flood_fill(vector<vector<int>>& grid,
int x, int y, int old_color, int new_color) {
    // Base cases: out of bounds or not the
target color
FloodFill(grid,  x + 1, y, old_color,
new_color); // Right
 flood_fill(grid, x + 1, y, old_color,
new_color); // Right
 flood_fill(grid, x - 1, y, old_color,
new_color); // Left
 flood_fill(grid, x, y + 1, old_color,
new_color); // Down
 flood_fill(grid, x, y - 1, old_color,
new_color); // Up
}
```

This implementation presents a few important recursive concepts that include specific termination conditions (out of bounds or mismatched target color), cell filling operations and recursive calls to solve adjacent cell problems.

## 3. EFFICIENT IMPLEMENTATIONS OF THE FILL ALGORITHM

We present the following three computer science problems which are available on the dedicated website pbinfo.ro [16]. The C++ FILL algorithm implementation was used to solve these problems. The website pbinfo.ro [15] is an educational platform in Romania that provides support to students and teachers in computer science education. The website hosts an extensive database of programming problems mainly focused on algorithmic problems which are organized based on academic level and topic. A problem

evaluation tool on the website performs automated evaluation of user submitted code which provides instant feedback about solution accuracy and efficiency [19].

### 3.1. Fill, Map1 and Photo problems

Fill problem (counting continents). The Fill problem [16] provides an n×m matrix containing only 0 and 1 elements to represent a map with land as 1 and water as 0. Two adjacent land features which share horizontal or vertical borders belong to the same continent. The task requires the calculation of the total number of continents found in the map. The program fill.cpp performs the following steps: it reads n, m and the input matrix; then, for each cell [i,j] that contains 1 (and has not been processed), it increments the continent counter and calls the recursive function fill(i,j). The function changes all cells that are connected to the cell into 1 then resets them to 0 to prevent double counting of the same continent. The result is written to the fill.out file. The example given in [4] presents the following array.

4 6
1 1 1 1 0 0 1 0
0 0 1 1 0 1 1 0
1 1 1 1 0 0 0 0
0 1 1 0 1 1 1 1

The algorithm detects 3 continents in the given input data which corresponds to the output value of 3.

The problem Map1 [17] presents a map encoded by an N×M matrix with 0 representing water and values ranging from 1 to K representing different countries.

Each country contains multiple departments which are defined as adjacent cell regions of the same value value (ignoring diagonal relationships). The requirement is dual:

- If p=1, the total water area, i.e. the number of cells with value 0, is required.
- If p=2, it asks for the list of country codes that have the most departments, in ascending order.

The program map1.cpp starts by reading the value of p followed by N, M, K and the map matrix. When p=1 the solution becomes straightforward because it involves counting the cells with 0 and displaying the result. When p=2, the solution method follows the approach of the Fill problem by initializing a variable to store the number of departments for each country code l in the range [1...K]. The program checks each cell in the matrix for the value l and performs a recursive call to fill that position while counting departments locally. The modified fill function searches for the value l and transforms all departments into 1 before restoring the original 1 value which allows z[l] to store the number of departments discovered for each country. It then evaluates the values stored in z [1...K] to determine which countries have achieved the highest number of departments. For instance, the two input cases p=1 and p=2 of the example in [5] demonstrate that with N=5, M=5, K=3 the answer at p=1 equals 11 (water = 11 cells) and at p=2 the countries with the most departments are 1 and 2 (each having 3 departments).

Photo problem (brightest area). The Photo problem [18] delivers black and white images through n×m matrices which contain binary values representing 0 for white dots and 1 for dark dots. A bright area consists of 0s that form connected regions which link through both rows and columns. The task requires users to locate the brightest section while counting its total number of bright dots. The foto.cpp program handles the problem by first reading n and m values before processing the matrix. We note that the code uses 0 to represent white but the counting algorithm uses 1 in some instances so the program converts 0 to 1 and 1 to 0 (making bright dots "1"). The program moves through the matrix to find each 1 which trigger fill function calls that both tag (set to 0) all connected areas while counting the number of cells in each area. It tracks down the maximum size of all components it finds. At the end, max represents the maximum number of highlights in an area.

### 3.2. Test data analysis and performance

The FILL algorithm demonstrates its functionality through examination of particular test datasets. The statement contains appropriate test values that demonstrate the algorithm's behavior. The FILL problem uses a 4×6 map with four highlighted continents to generate result 3 through the recursive program according to the example in [4]. The program identifies 11 cells with value 0 in Map1 during the first example where p equals

1 (5×5) while the same program shows countries as "1 2" in the second example with p=2. The 6×6 Photo example produces the correct result of 5. Multiple artificial test cases exist for assessing performance. The authors measured how long it took for the Fill algorithm to count continents in quadratic matrices containing random 0/1 distributions at various sizes.

The Table 2 provides experimental results showing average times in milliseconds between recursive DFS and iterative BFS implementations:

| Matrix size N×N | Recursive DFS time (ms) | Iterative BFS time (ms) |
|---|---|---|
| 10×10 | 0.05 | 0.06 |
| 20×20 | 0.15 | 0.16 |
| 30×30 | 0.31 | 0.36 |
| 50×50 | 0.78 | 0.95 |
| 70×70 | 2.02 | 2.03 |
| 100×100 | 3.71 | 4.57 |

**Table 2.** Flood Fill Algorithm Performance Comparison

The observed time growth matches the expected O(n-m) complexity because it increases directly with $N^2$. The tests show that recursive DFS (DFS) operates slightly faster but both implementations demonstrate equivalent performance at small dimension sizes. Both Map1 and Photo use similar complexities because the algorithm needs to scan the entire file; Map1 computes zero counts at O(n-m) for p=1 and performs flood-fill component traversal at p=2. The Photo complexity operates at O(n-m) level because the maximum region of 0 determines its performance as shown in the code structure and supported by theoretical O(n-m) analysis:

## 4. PRACTICAL APPLICATIONS

Flood-fill and connected-component algorithms split binary images into their maximum connected regions. Each connected component functions as an important segment which benefits image analysis operations (including object detection and region-of-interest extraction) [11]. These concepts serve as base components for both image segmentation and morphological analysis [11].

Computer graphics and design: The flood-fill operation colors closed shapes in vector and raster graphics by filling polygons. CAD and GIS systems together with graphics editors represent some of the applications. The SCAFF algorithm generates accurate polygon boundary masks which are useful for machine learning [2].

Graph segmentation: A flood-fill algorithm using BFS or DFS works similarly on graph structures to identify clusters or connected components. The technique finds applications in mapping as well as network analysis and additional fields.

Robotics and Augmented Reality: Filling algorithms help navigation and augmented reality systems identify accessible regions and surfaces in visual data.

## 5. POSSIBLE OPTIMIZATIONS FOR THE FILL ALGORITHM

The algorithm employs an explicit data structure (stack or queue) instead of recursive calls to perform the traversal [7,8]. Initialize the stack/queue with the seed pixel, then loop: pop one pixel, color it, and push its eligible neighbors. The algorithm avoids stack overflow and allows control over traversal order (DFS via stack, BFS via queue) [7].

The algorithm uses a horizontal scan to fill the seed region which tracks the span's leftmost and rightmost boundaries. The algorithm continues to fill by scanning for new seed pixels within the row above and below the span [9]. The method requires processing entire horizontal segments to achieve better performance with reduced stack/queue entries and pixel tests. The method produces superior performance in real-world scenarios due to its ability to eliminate redundancy [9].

Pre-check before push - Check if a neighbor pixel requires filling (target color and not yet

visited) before pushing it onto the stack/queue. Mark it immediately when checking, to prevent multiple enqueues [10]. The optimization reduces duplicate stack entries while simultaneously decreasing the size of the stack/queue.

Two-pass labeling & Union-Find – Connected-component labeling can be done in two raster passes [11]. The first raster pass temporarily assigns labels to foreground pixels while keeping track of their label equivalences. A Union-Find (disjoint set) structure efficiently maintains these equivalences [11]. The second raster pass replaces each pixel label with its smallest representative set label. This technique discovers all connected regions through an approach without deep recursion.

The execution of region filling and labeling can be accelerated through parallel processing along with hardware acceleration. SIMD intrinsics enable simultaneous processing of multiple pixels through a single operation (e.g., flood-filling 128 voxels [12]). The CPU runs multiple image blocks in parallel using OpenMP/TBB multi-threading. GPU implementations that employ optimized block-based Union-Find operations deliver major speed improvements [13]. Parallel methods of fill operations produce significant performance improvements according to research findings (fill operations experience approximately 60% speedup) [14,12].

## 6. CONCLUSIONS

The fundamental recursive FILL (flood-fill) method enables region-filling operations yet becomes inefficient and causes stack overflows when applied without optimization [3]. An explicit stack/queue implementation combined with scanline span techniques and pre-checks minimizes redundant computations. Two-pass labeling with Union-Find handles connected components efficiently, and parallel SIMD/GPU implementations yield large speedups. Research by [14,12] demonstrates how these optimized methods enhance performance. These techniques are useful for image segmentation applications and graphics as well as related use cases.

## REFERENCES

[1] M. F. Sholahuddin and T. Sutabri, "Flood Fill and Scanline Fill Algorithm Optimization to Improve Design and Animation Application Performance," Int. J. Sci. Profes. (IJ-ChiProf), vol. 4, no. 2, pp. 531–535, 2025.

[2] Y. He, T. Hu, and D. Zeng, "Scan-Flood Fill (SCAFF): an Efficient Automatic Precise Region Filling Algorithm for Complicated Regions," in Proc. IEEE Conf. Computer Vision and Pattern Recognition Workshops (CVPRW), 2019, pp. 761–769.

[3] J. Chen, Q. Yao, H. Sabirin, K. Nonaka, H. Sankoh, and S. Naito, "An optimized union-find algorithm for connected components labeling using GPUs," arXiv:1708.08180, 2017.

[4] Atrufulgium, "SIMD flood-fill goes brrr," blog post, Aug. 21, 2024. [Online]. Available: https://atrufulgium.net/2024/08/21/simd-flood-fill [Accessed: September 2, 2025].

[5] C. A. Bouman, "Connected Component Analysis," lecture notes, Purdue University, Jan. 2025. [Online]. Available: https://engineering.purdue.edu/~bouman/ [Accessed: September 2, 2025].

[6] Wikipedia contributors, "Flood fill," Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Flood_fill [Accessed: September 2, 2025].

[7] Wikipedia contributors, "Recursion," Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Recursion [Accessed: September 2, 2025].

[8] Wikipedia contributors, "Connected-component labeling," Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Connected-component_labeling [Accessed: September 2, 2025].

[9] Sholahuddin, F., & Sutabri, T. (2025). Flood Fill and Scanline Fill Algorithm Optimization to Improve Design and Animation Application Performance. International Journal Scientific and Professional, 4(2), 531–535.

https://doi.org/10.56988/chiprof.v4i2.89 [Accessed: September 2, 2025].
[10] CVF Open Access, "He et al., CVPRW 2019 Paper," [Online]. Available: https://openaccess.thecvf.com/content_CVPRW_2019/papers/CEFRL/He_Scan-Flood_FillSCAFF_An_Efficient_Automatic_Precise_Region_Filling_Algorithm_for_CVPRW_2019_paper.pdf [Accessed: September 2, 2025].
[11] C. A. Bouman, "Connected Component Analysis," lecture notes, Purdue University, Jan. 2025. [Online]. Available: https://engineering.purdue.edu/~bouman/ece637/notes/pdf/ConnectComp.pdf [Accessed: September 2, 2025]
[13] Python Imaging Library (PIL), "ImageDraw and floodfill methods," [Online]. Available: https://pillow.readthedocs.io/ [Accessed: September 2, 2025].
[14] Chen, J. et al., "Connected Components Labeling Optimized on GPU," [Online]. Available: https://arxiv.org/abs/1708.08180 [Accessed: September 2, 2025].
[15] pbinfo.ro, "Probleme de informatică," [Online]. Available: https://www.pbinfo.ro. [Accessed: September 2, 2025].
[16] pbinfo.ro, "Probleme de informatică," [Online]. Available: https://www.pbinfo.ro/probleme/837/fill [Accessed: September 2, 2025].
[17] pbinfo.ro, "Probleme de informatică," [Online]. Available: https://www.pbinfo.ro/probleme/1496/harta1 [Accessed: September 2, 2025].
[18] pbinfo.ro, "Probleme de informatică," [Online]. Available: https://www.pbinfo.ro/probleme/3220/foto [Accessed: September 2, 2025].
[19] Adrian Runceanu, Mihaela Runceanu, „Algoritmi implementaţi în limbajul C++. Volumul IV - Subprograme", Editura Academica Brâncuşi, 2024, ISBN 978-630-340-016-7