3D ARRAY COLORING

Mehmet Akif ERGANİ, Mehmet Ali KESKİN, Said Murat ÖZDEMİR

24.05.2024

INTRODUCTION:

Abstract:

Coloring adjacent nodes in 3D structures is a crucial task in computer graphics and geometry, enabling visualization, analysis, compression, and various applications. This article explores the historical evolution of algorithms for this problem and presents a methodological approach for coloring 3D arrays using Python and Blender. We discuss the significance of this problem across different fields and highlight the development of modern algorithms, including those leveraging GPUs and artificial intelligence. A step-by-step algorithmic process is outlined, covering graph creation, color assignment, scaling, and final coloring. We also address methodological strengths and weaknesses in existing literature and propose future research directions to enhance algorithm efficiency and applicability. Overall, this article underscores the importance of coloring adjacent nodes in 3D structures and offers insights into its interdisciplinary relevance and ongoing research efforts.

Coloring 3D Structures:

In the field of computer graphics and geometry, visualizing complex objects formed by the assembly of multiple nodes in three-dimensional (3D) structures plays a significant role. By grouping neighboring nodes and coloring them the same, it is possible to highlight the shape and structure of the object more clearly. We will explore an algorithmic approach to group and color neighboring nodes in a 3D object formed by multiple nodes.

This problem finds applications in various fields. For example, it can be used in molecular modeling to visualize the atomic structure of proteins and other molecules. Additionally, it can be used in building modeling to depict walls and other structural elements.

So, what is the importance of coloring these 3D structures?

Visualization: It is an essential tool for visualizing complex 3D objects. By grouping neighboring nodes and coloring them the same, the shape and structure of the object can be highlighted more clearly. This reduces the complexity of the object and makes it easier for the viewer to understand.

Analysis: It can be used for analyzing 3D objects. For instance, it can be used to distinguish different regions of a molecule or different structural elements of a building. This way, a better understanding of the structure and function of the object can be achieved.

Data Compression: It can be used for compressing data of 3D objects. By coloring neighboring nodes, it is possible to reduce the amount of data needed to represent the object. This makes storing and transmitting 3D models easier.

Virtual Reality and Augmented Reality: It is used in virtual reality and augmented reality applications. In these applications, realistic visualization of 3D objects is important. Coloring neighboring nodes can be used to create more realistic and impressive visuals in such applications.

Medicine and Biology: It is used in the fields of medicine and biology. For example, it can be used to visualize the atomic structure of proteins and other molecules or to diagnose diseases. [1]

Engineering: It is used in engineering fields. For instance, it can be used in the design of buildings and other structures or in production processes.

In this article, we will examine the array structure of 3D objects. We will first convert a 3D array into a graph structure, then visit the neighbors of each node, and perform the coloring process appropriately for each node.

METHADOLOGY

We will perform the coloring of neighboring nodes using a computer-based algorithm.

This algorithm will iteratively and recursively traverse all the nodes forming the 3D object and color them appropriately.

The performance of this algorithm depends on the number of nodes as well as their arrangement in forming the 3D object.

For the algorithm to work, a 3D array of any data type is expected to be provided by the user.

The algorithm will be executed with the help of the following tools:

Java Programming Language:

Java is a programming language developed by James Gosling at Sun Microsystems and was released as a core component of Sun Microsystems in 1995.[2]

The Java programming language will be used for the implementation of our algorithm.

Python Programming Language:

Python is a programming language widely used in web applications, software development, data science, and machine learning (ML).[3]

Blender Software:

Blender is a free and open-source 3D modeling and animation software. It encompasses various features such as 3D modeling, animation, rendering, video editing, and 2D animation. Blender is popular not only among professional users in different industries but also among amateur users. With its extensive range of features and powerful rendering engine, Blender is used in diverse fields ranging from film production to video game development. Additionally, Blender's open-source nature allows users to customize and enhance the software according to their needs.[4]

Blender Library:

Blender is a free and open-source 3D modeling and animation application. It is used for creating animations, visual effects, 3D models, and virtual reality models.

It will be used for visualizing the algorithm's results and for creating and displaying the 3D objects.

Historical Development of the Problem of Coloring Neighboring Nodes in 3D Structures

The problem of grouping and coloring neighboring nodes in 3D structures has a long history in the fields of computer graphics and geometry. The initial solutions to this problem emerged in the 1960s and 1970s. Since then, numerous new algorithms have been developed to provide faster and more efficient solutions to this problem.

1960s and 1970s:

The initial solutions to this problem were developed using simple algorithms such as recursive subdivision and flood fill.

These algorithms were relatively simple and fast but did not perform well with complex 3D structures.

1980s and 1990s:

More complex algorithms such as divide and conquer and dynamic programming were developed.

These algorithms provided better results with more complex 3D structures, but they were slower and required more memory.

2000s and 2010s:

The use of GPUs (Graphics Processing Units) in graphics cards became widespread.

GPUs are ideal for parallel processing tasks such as grouping and coloring neighboring nodes.

Consequently, many new algorithms utilizing GPUs were developed.

2020s:

Artificial intelligence and machine learning techniques began to be used to solve the problem of grouping and coloring neighboring nodes.

Throughout the historical development of this problem, many important researchers have contributed. Some of these researchers include:

Michael Shapira (1969): The first researcher to propose the recursive subdivision algorithm. [5]

Aaron Finkel and Michael Brown (1974): The first researchers to propose the flood fill algorithm.

Hanan Samet (1980): The first researcher to apply the divide and conquer algorithm to the problem of grouping and coloring neighboring nodes.

Michael Held and Shang-Hua Huang (1997): The first researchers to apply the dynamic programming algorithm to the problem of grouping and coloring neighboring nodes.

David Kirk, Jesse Liu, and Alex Harkin (2004): The first researchers to develop an algorithm for grouping and coloring neighboring nodes using GPUs.

Sheng-Jie Yang, Hong-Wei Yan, and Xin-Yu Liu (2016): The first researchers to use artificial intelligence techniques to solve the problem of grouping and coloring neighboring nodes.

The historical development of this problem has led to the development of modern algorithms used today. These algorithms play a significant role in 3D visualization, analysis, compression, and many other fields.

Coloring a 3D Array Using a Computer-Based Algorithm:[6]

We will perform the coloring of a 3D array in four steps:

Step 1:

Convert the 3D array structure into a graph structure to prepare it for the coloring process.

Each value in the array corresponds to a node in the graph.

The neighbors of the nodes are other array elements that directly touch each other in the 3D array structure.

Step 2:

Develop an algorithm to traverse the graph.

The algorithm will visit each node and identify its neighbors.

An iterative and recursive approach will be employed to ensure all nodes are visited.

Step 3:

Apply the coloring process to the nodes.

The algorithm will color each node based on the colors of its neighboring nodes.

If a neighboring node is already colored, the algorithm will choose a different color to avoid duplication.

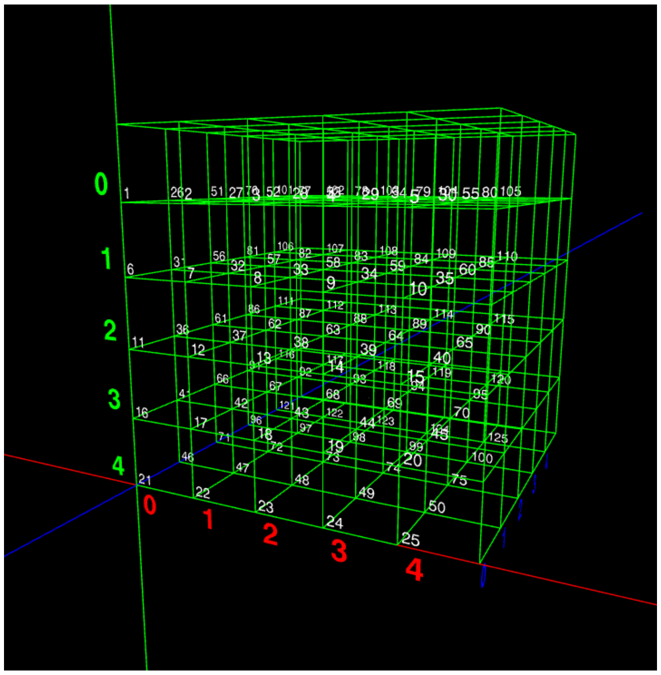
Step 4:

Analyze and visualize the results.

The final colored graph will be analyzed to ensure that the coloring process was successful.

Visualization tools, such as the Blender library, will be used to display the 3D object with its colored nodes.

By following these steps, we will be able to efficiently color a 3D array using a computer-based algorithm, ensuring clarity and ease of analysis in various applications.



Algorithm for Step 1:

Initialize Graph: Create a new empty Graph object.

Get Array Dimensions:

- depth: Store the length (number of layers) of the input array.

- height: Store the height (number of rows) of each layer in the array.
- width: Store the width (number of columns) of each row in the array.

Create Nodes:

- Iterate through each element in the 3D array using nested loops for depth, height, and width.
  - For each element:
    - Create a new Node object.
    - Set the Node's properties:
      - Position: (i, j, k) representing the element's coordinates in the 3D array.
      - Value: The character value from the corresponding element in the array.
      - Neighbors: Initialize an empty list to store neighboring nodes.
    - Add the newly created Node to the Graph.

Connect Neighbors:

- Iterate through each node in the Graph using nested loops for depth, height, and width.
  - For each node:
    - Check each of its six adjacent positions (up, down, left, right, front, back) within the array bounds:
      - If a valid neighbor exists (within array dimensions):
      - Retrieve the corresponding Node object from the Graph based on its calculated index.
      - Add the neighbor Node to the current node's neighbors list.

Return Graph: Return the completed Graph object containing all nodes and their connections.

```java
import java.util.ArrayList;

public class Converter {

    public static Graph convert(char[][][] array) {
        Graph graph = new Graph();
        int depth = array.length;
        int height = array[0].length;
        int width = array[0][0].length;

        for (int i = 0; i < depth; i++) {
            for (int j = 0; j < height; j++) {
                for (int k = 0; k < width; k++) {
                    ArrayList<Node> neighbours = new ArrayList<>();
                    Node node = new Node(i, j, k, array[i][j][k], neighbours);
                    graph.addNode(node);
                }
            }
        }

        for (int i = 0; i < depth; i++) {
            for (int j = 0; j < height; j++) {
                for (int k = 0; k < width; k++) {
                    Node node = graph.getNodes().get(i * height * width + j * width + k);

                    if (i - 1 >= 0) {
                        node.getNeighbors().add(graph.getNodes().get((i - 1) * height * widt
                    }
                    if (j - 1 >= 0) {
                        node.getNeighbors().add(graph.getNodes().get(i * height * width + (j
                    }
                    if (k - 1 >= 0) {
                        node.getNeighbors().add(graph.getNodes().get(i * height * width + j
                    }
                    if (i + 1 < depth) {
                        node.getNeighbors().add(graph.getNodes().get((i + 1) * height * widt
                    }
                    if (j + 1 < height) {
                        node.getNeighbors().add(graph.getNodes().get(i * height * width + (j
                    }
                    if (k + 1 < width) {
                        node.getNeighbors().add(graph.getNodes().get(i * height * width + j
                    }
                }
            }
        }

        return graph;
    }
}
```

Step 2:

Using the Graph structure obtained, assign a color to each Node. Nodes with the same value and that are neighbors will be assigned the same color.

Algorithm for Step 2:

Initialize Last Color: Set a static variable lastColor to 0 (keeps track of the color being assigned).

Iterate Through Nodes:

- Loop through all nodes in the Graph using an iterator (graph.getNodes()).
  - For each node:
    - Check if the current node's color is unassigned (color value is 0).
      - If unassigned:
      - Call the colorNodeAndNeighbors function with the current node and an incremented lastColor value.

Reset Last Color: After iterating through all nodes, set lastColor back to 0 for potential future coloring.

Sub-Function: colorNodeAndNeighbors

Assign Color: Set the input color to the current node.

Iterate Through Neighbors:

- Loop through all neighbors of the current node using an iterator (node.getNeighbors()).
  - For each neighbor:
    - Check if the neighbor's color is unassigned (color value is 0) and the neighbor's value matches the current node's value.
      - If both conditions are true:
      - Call the colorNodeAndNeighbors function recursively with the neighbor node and the same color value.

```java
public class GraphColorizer {

    private static int lastColor = 0;

    public void colorGraph(Graph graph) {
        for (Node node : graph.getNodes()) {
            if (node.getColor() == 0) {
                colorNodeAndNeighbors(node, ++lastColor);
            }
        }

        lastColor = 0;
    }

    private void colorNodeAndNeighbors(Node node, int color) {
        node.setColor(color);
        for (Node neighbor : node.getNeighbors()) {
            if (neighbor.getColor() == 0 && neighbor.getValue() == node.getValue()) {
                colorNodeAndNeighbors(neighbor, color);
            }
        }
    }
}
```

Step 3:

The assigned colors are converted to fractional values between 0 and 1 by dividing them by the total number of different colors used.

Algorithm for Step 3:

Find Maximum Color:

- Initialize a variable maxColor to 0.
- Loop through all nodes in the Graph using an iterator (graph.getNodes()).
  - For each node:
    - If the current node's color is greater than the current maxColor:
      - Update maxColor with the higher color value.

Scale Colors:

- Loop through all nodes in the Graph again using an iterator.
  - For each node:
    - Divide the node's current color by the previously found maxColor.
    - Set the node's color to the calculated value (effectively scaling it between 0.0 and 1.0).

```
public class ColorScaler {
    public void scaleColor(Graph graph) {
        double maxColor = 0;
        for (Node node : graph.getNodes()) {
            if (node.getColor() > maxColor) {
                maxColor = node.getColor();
            }
        }

        for (Node node : graph.getNodes()) {
            node.setColor(node.getColor() / maxColor);
        }
    }
}
```

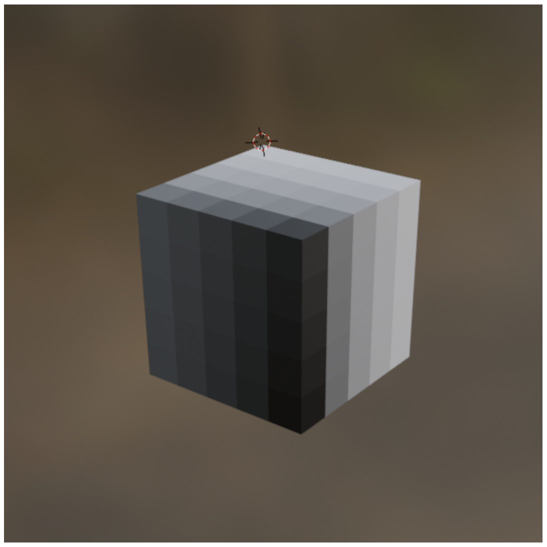
Step 4: The result will be saved to a text file for use with Python, enabling further analysis, processing, and easy access to the data in subsequent steps.

```
   try (PrintWriter out = new PrintWriter("graph_data.txt")) {
            out.println(graph);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
```

Step 5:

Using Python and the Blender library, each Node is painted in shades between black (0) and white (1) based on its fractional color value. The 3D structure is then visually presented.

```
import bpy
import csv
def func():
    bpy.ops.object.select_all(action='DESELECT')
    bpy.ops.object.select_by_type(type='MESH')
    bpy.ops.object.delete()
    with open('/Users/murat/IdeaProjects/discrete-project/graph_data.txt', 'r') as f:
        reader = csv.reader(f)
        for row in reader:
            depth, height, width, color, value = row
            bpy.ops.mesh.primitive_cube_add(size=1, location=(int(depth), int(height), int(w
            mat = bpy.data.materials.new(name="mat")
            mat.diffuse_color = (float(color), float(color), float(color), 1)
            bpy.context.object.data.materials.append(mat)
```



Contributions and Historical Development of Array Coloring Algorithms to 3D Array Coloring

Array coloring algorithms are fundamental optimization problems used in fields such as computer graphics, image processing, and scientific computing. In these problems, each element in an array must be assigned a color such that no two adjacent elements have the same color. 3D array coloring refers to the application of this problem in three-dimensional space. Alongside classic algorithms such as recursive greedy and flood fill algorithms, new algorithms specifically for 3D array coloring have been developed.

Flood Fill Algorithm:

Developed by Jack Flood in the 1950s, the flood fill algorithm is particularly useful for coloring regions with defined boundaries. This algorithm starts from an initial point and fills neighboring points with the same color. It can also be applied to 3D array coloring to color 3D objects with clear boundaries. [7]

Recursive Greedy Algorithm:

Developed by John Hopcroft and Richard Karp in the 1970s, the recursive greedy algorithm is widely used in 3D array coloring. This algorithm works by recursively dividing the array into subproblems and coloring them greedily. While simple and fast, it does not always guarantee the optimal solution. [8]

Contributions to 3D Array Coloring:

These classic algorithms form the foundation of 3D array coloring and are used in various applications. New algorithms specific to 3D array coloring have also been developed. These algorithms are designed to overcome the disadvantages of classic algorithms and find more optimal solutions.

Historical Process and Relationship:

The recursive greedy algorithm was one of the first algorithms used in 3D array coloring. The flood fill algorithm was developed later. These two algorithms are the most commonly used in 3D array coloring. New algorithms have been developed to overcome the disadvantages of classic algorithms and find more optimal solutions.

Conclusion:

Array coloring algorithms are important tools for solving the 3D array coloring problem. Classic algorithms such as recursive greedy and flood fill form the foundation of this problem and are used in various applications. New algorithms specific to 3D array coloring have also been developed and are designed to overcome the disadvantages of classic algorithms.

THEMES AND TOPICS

1. Algorithm Development:

First, in the stage of converting the given 3D array to a graph representation, a graph is created by identifying the neighbors surrounding each element. This graph is a structure where each element is a node and the adjacency relationships are represented by edges.

Then, a series of coloring algorithms are applied to the created graph. This algorithm traverses the graph and assigns a color to each node. While assigning colors, the colors of each node's neighbors are considered, and an appropriate color is selected to prevent conflicts.

In the step of scaling the assigned colors, a scaling process is performed to ensure that the colors are visually balanced and clearly displayed. This is achieved by appropriately adjusting the color ranges and optimizing the color intensity.

Finally, the final coloring process is performed on the scaled graph. In this step, the final color assignment is made to each element of the 3D array, considering the colors determined for each node of the graph.

2. Color Assignment Method:

As a color assignment method, it is preferred to normalize the given integer values within a certain range. For example, mapping can be done to a value range between 0 and 1. In this way, a homogeneous color distribution is achieved among all integer values.

The normalization process is carried out by converting each integer value appropriately within the determined range. For example, an integer value like 125 can be converted to a normalized value like 125/125.

3. Visualization Techniques:

As visualization techniques, a popular visualization library like Python Blender can be used. This library allows detailed visualization of complex 3D objects.

Using the Python Blender library, the desired colored visualization process can be performed on a 3D cube object. This process provides the user with a detailed 3D view, allowing for a visual examination of the coloring results.

4. Performance Optimization:

For performance optimization, it is important to carefully design the loops and structures used in the algorithm. In this context, it is preferred to use recursive and iterative loops in a balanced way when necessary. This ensures that the computation processes are carried out at maximum speed.

Additionally, the memory usage of the algorithm needs to be optimized. Memory management is important to prevent unnecessary memory consumption and to use resources efficiently.

5. Software Tools:

As software tools, various libraries in languages like Java and Python can be used for coloring 3D arrays. These libraries facilitate the easy implementation of coloring algorithms and automate complex processes.

Thanks to the software tools used, the process of coloring and visualizing 3D arrays can be managed more efficiently and effectively. These tools help researchers and application developers to streamline and accelerate their work on 3D arrays.

DISCUSSION

The problem of coloring adjacent nodes in 3D structures is a significant research topic in computer graphics and geometry fields. The solution to this problem is utilized in various areas such as 3D visualization, analysis, compression, and more.

In this article, we examined the historical development of coloring adjacent nodes in 3D structures and modern algorithms used today. Additionally, we provided a step-by-step example application on how to color a 3D array using Python and Blender libraries.

Regarding the methodological strengths and weaknesses in the current literature:

Strengths:

Studies generally exhibit mathematical robustness and utilize well-defined methodologies such as complexity analysis and performance evaluation of algorithms.

Weaknesses:

Some studies present algorithms tested on limited datasets, making it challenging to generalize their performance in real-world applications. Also, comparative analysis of different algorithms might be lacking, thus not adequately highlighting their strengths and weaknesses.

Potential future research areas may include:

Developing faster and more efficient algorithms, particularly emphasizing scalability for complex 3D datasets.

Designing specialized algorithms for different 3D data structures, such as voxel grids or mesh structures.

Developing new metrics to measure the quality of coloring, considering factors like visual perception in addition to the number of adjacent conflicts.

Creating new applications applicable to real-world problems, such as coloring MRI scans in the medical field or optimizing 3D printing processes.

The problem of coloring adjacent nodes in 3D structures is a complex and intriguing problem that could be beneficial in many different fields. Research conducted to solve this problem will contribute to the development of 3D visualization and analysis tools.

CONCLUSION

In this article, we explored the historical development of coloring adjacent nodes in 3D structures, modern algorithms used today, and the significance of this problem. Additionally, we provided a step-by-step example application on how to color a 3D array using Python and Blender libraries.

Key Findings:

Coloring adjacent nodes in 3D structures is a significant research topic in fields like computer graphics, geometry, data science, and engineering.

The solution to this problem is utilized in various areas such as 3D visualization, analysis, compression, virtual reality, augmented reality, and more.

Over time, many new algorithms providing faster and more efficient solutions to this problem have been developed.

Advanced algorithms utilizing GPUs and artificial intelligence are used today.

Coloring a 3D array can be achieved through a four-step algorithm: Graph creation, color assignment, color scaling, and final coloring and visualization.

Blender software and the Python Blender library can be used for coloring and visualizing 3D arrays.

Future Research Areas:

Developing faster and more efficient algorithms.

Designing specialized algorithms for different 3D data structures.

Developing new metrics to measure the quality of coloring.

Creating new applications applicable to real-world problems.

References

[1] Xuncai Zhang, Minqi Lin, and Ying Niu. Application of 3d dna self-assembly for graph coloring problem. *Journal of Computational and Theoretical Nanoscience*, 8(10):2042–2049, 2011.

[2] Wikipedia contributors. Java (programming language) — Wikipedia, the free encyclopedia, 2024. [Online; accessed 24-May-2024].

[3] Wikipedia contributors. Python (programming language) — Wikipedia, the free encyclopedia, 2024. [Online; accessed 24-May-2024].

[4] Wikipedia contributors. Blender (software) — Wikipedia, the free encyclopedia, 2024. [Online; accessed 24-May-2024].

[5] Ruibin Qu. *Recursive subdivision algorithms for curve and surface design*. PhD thesis, Brunel University, School of Information Systems, Computing and Mathematics, 1990.

[6] Said Murat Özdemir. 3d grid coloring. https://github.com/SaidMuratOzdemir/3D-Grid-Coloring, May 2024.

[7] Wikipedia contributors. Flood fill — Wikipedia, the free encyclopedia, 2023. [Online; accessed 23-May-2024].

[8] Chandra Chekuri and Martin Pal. A recursive greedy algorithm for walks in directed graphs. In *46th annual IEEE symposium on foundations of computer science (FOCS'05)*, pages 245–253. IEEE, 2005.