CS575 Project Report

Team Member(s):
B01176039 Vidhya Suram, B01165891 Mridhula Polur

Ⅰ. Algorithms

1. Knuth-Morris-Pratt(KMP):

KMP is an efficient string-searching algorithm that finds all occurrences of a pattern in a text in **linear time**. It avoids re-checking characters by precomputing a **LPS (Longest Prefix-Suffix)** table for the pattern. During the search, when a mismatch occurs, the algorithm uses the LPS table to shift the pattern intelligently, instead of restarting the search from the next character. This ensures the overall time complexity is **O(n + m)** for a text of length *n* and pattern of length *m*.

2. AVL Tree:

An AVL Tree is a height-balanced binary search tree where the **height difference (balance factor)** between the left and right subtrees of any node is at most 1.
After every insertion or deletion, the tree automatically restores balance using **rotations** (LL, RR, LR, RL).
This guarantees that search, insert, and delete operations all run in **O(log n)** time.
You implemented the tree with operations for insertion, balancing, calculating heights, and performing rotations.

3. Graph Coloring(Backtracking & Greedy):

Greedy:

The greedy algorithm colors vertices one by one, always assigning the smallest available color that is not used by any of its colored neighbors.
It is fast with time complexity around O(V + E), but it does not guarantee the minimum number of colors, as the result depends on the vertex order. Suitable for large graphs

Backtracking:

The backtracking solution explores color assignments recursively and only proceeds when the current color choice does not violate adjacency constraints.
This algorithm systematically tries colors for every vertex, making it capable of finding a valid coloring using the minimum possible number of colors.
While it guarantees the optimal solution, its worst-case time complexity is exponential, making it suitable for small to medium graphs.

Ⅱ. Software Design and Implementation:

Software Design:

The software is organized as three independent modules, KMP String Matching, AVL Tree, and Graph Coloring, each with its own visualization interface. The design follows a clear separation between:

Core Algorithm Logic (implemented from scratch):

a. Each algorithm (KMP prefix table, AVL rotations, Backtracking & Greedy coloring) is written manually without using external algorithm libraries.
b. Data structures such as arrays, trees, adjacency lists, and helper functions are implemented directly in Python.

Visualization Layer:

c. A Tkinter-based GUI is used to display step-by-step execution of each algorithm.
d. Input fields allow users to enter strings (for KMP), tree values (for AVL), or edges/nodes (for graph coloring).
e. The canvas/labels dynamically update to show progress, comparisons, rotations, or color assignments.

Performance Measurement:

perf_counter() is used to record algorithm execution time and display it to the user.

Libraries and Tools Used:

- **Python 3** – Used as the primary programming language for implementing all algorithms.
- **Tkinter** – Used to create the graphical user interface, including windows, labels, buttons, and visualization canvases.
- **time.perf_counter** – Used to measure performance and record high-resolution execution times for each algorithm.
- **math / basic utility modules** – Used only for small helper operations; not involved in algorithm logic.

No external data-structure or algorithm libraries were used. All core algorithm logic (KMP, AVL Tree, Graph Coloring) was fully implemented manually from scratch.

Ⅲ. Project Outcomes:

1. **Knuth-Morris-Pratt(KMP):**

   The project demonstrates the successful implementation of the Knuth–Morris–Pratt (KMP) string matching algorithm, which efficiently searches for patterns within a text. By constructing and using the LPS (Longest Prefix Suffix) array, the algorithm avoids redundant comparisons, achieving linear time complexity O(n + m). The implementation accurately identifies all occurrences of a pattern, including overlapping matches, and handles edge cases such as repeated characters, patterns longer than the text, and no-match scenarios.

   Key Outcomes:

   1. **Understanding of Pattern Matching Concepts**
      - Developed a strong understanding of string searching, prefix functions, and the importance of avoiding redundant comparisons.

   2. **Efficient Algorithm Implementation**
      - Implemented KMP from scratch without using built-in string functions.
      - Correctly constructed the LPS array and applied it to search patterns efficiently.

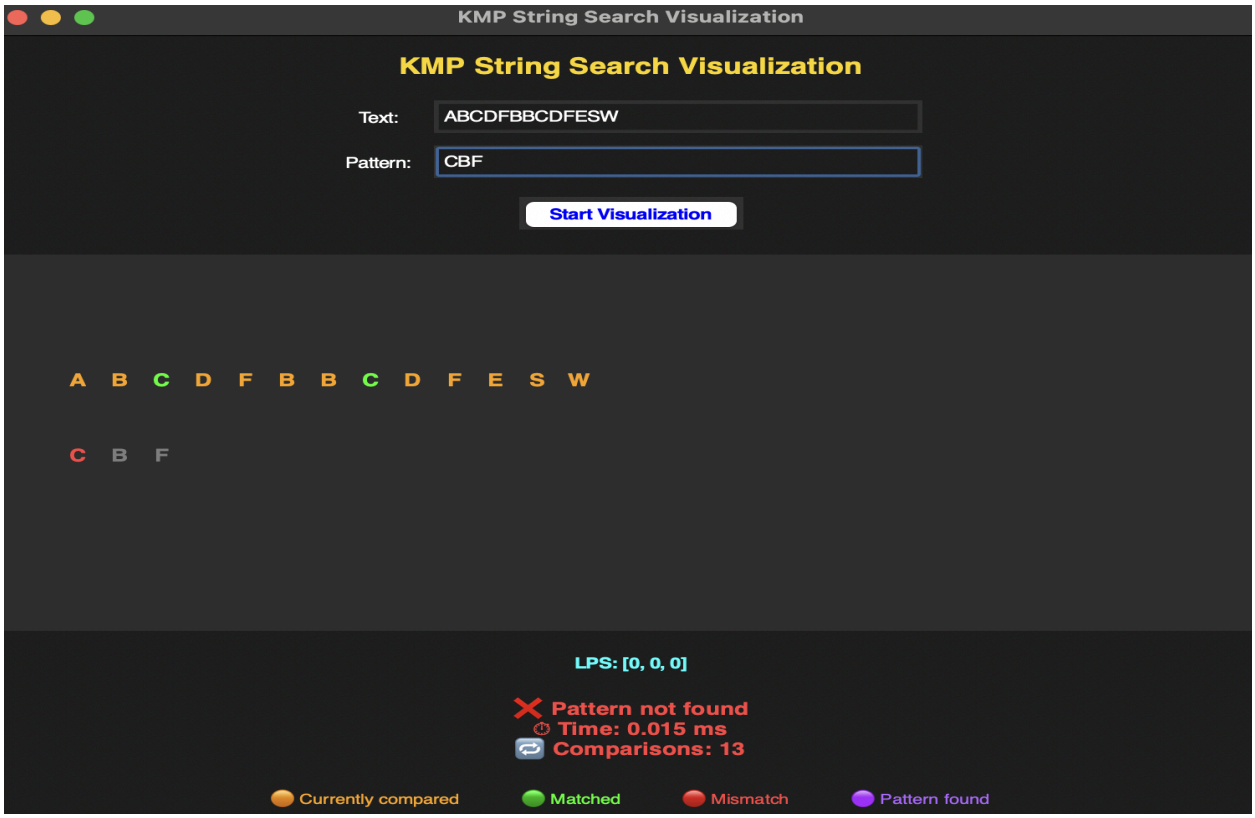
   3. **Finding All Occurrences**
      - The algorithm identifies all instances of a pattern within a text, including overlapping patterns.

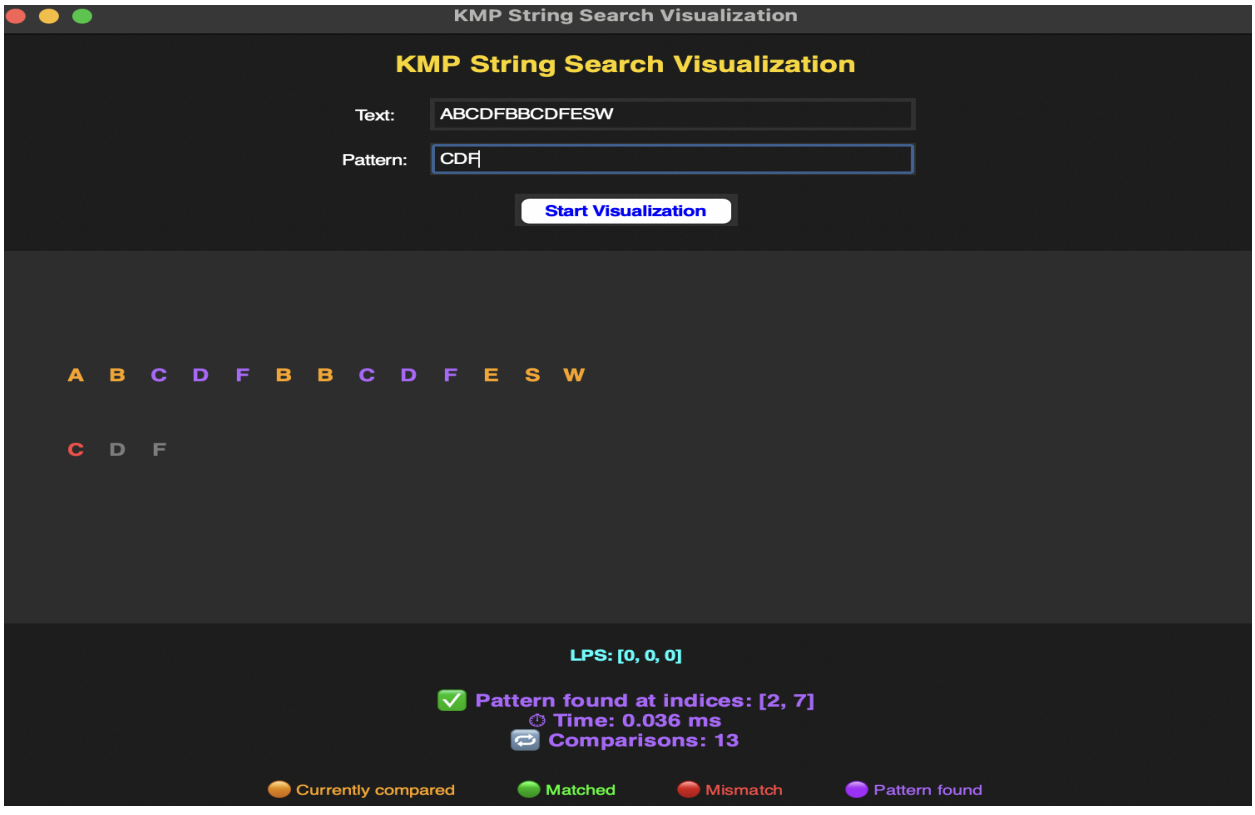
   4. **Edge Case Handling**
      - Successfully handled special scenarios like empty strings, patterns longer than the text, repeated characters, and no-match situations.

   5. **Performance Analysis**
      - Demonstrated the efficiency of KMP compared to the naive string matching algorithm in terms of the number of comparisons and time complexity.

   6. **Learning Outcomes**
      - Gained practical experience in algorithm design, recursion, and string processing.
      - Enhanced understanding of asymptotic analysis and efficient problem-solving techniques.

The following are examples illustrating cases when the pattern is found in the string and when it is not:


KMP String Search Visualization
KMP String Search Visualization
Text:
ABCDFBBCDFESW
Pattern:
CBF
Start Visualization
A B C D F B B C D F E S W
C B F
LPS: [0, 0, 0]
Pattern not found
Time: 0.015 ms
Comparisons: 13
Currently compared
Matched
Mismatch
Pattern found


KMP String Search Visualization
KMP String Search Visualization
Text:
ABCDFBBCDFESW
Pattern:
CDF
Start Visualization
A B C D F B B C D F E S W
C D F
LPS: [0, 0, 0]
Pattern found at indices: [2, 7]
Time: 0.036 ms
Comparisons: 13
Currently compared
Matched
Mismatch
Pattern found

2. AVL Tree:

The project demonstrates the successful implementation of an **AVL (Adelson-Velsky and Landis) Tree**, a self-balancing binary search tree. The AVL tree maintains **height balance** at every node, ensuring that the difference between the heights of the left and right subtrees (balance factor) is **-1, 0, or 1**. This balance guarantees **efficient operations** such as insertion, deletion, and search, all with a **time complexity of O(log n)**.

Key Outcomes:

1. **Understanding of AVL Tree Concepts**
   - Gained strong understanding of **binary search trees (BSTs)** and **self-balancing trees**.
   - Learned how rotations (LL, RR, LR, RL) maintain tree balance.

2. **Implementation of Insert and Delete Operations**
   - Implemented **insertion** with automatic rotations to maintain balance.
   - Implemented **deletion**, handling cases such as removing leaf nodes, nodes with one child, and nodes with two children, while maintaining AVL properties.

3. **Rotations for Balancing**

   Correctly implemented all four types of rotations:

   - **LL (Left-Left)** → Right rotation
   - **RR (Right-Right)** → Left rotation
   - **LR (Left-Right)** → Left rotation on child, then right rotation on root
   - **RL (Right-Left)** → Right rotation on child, then left rotation on root

   ■ Demonstrated how rotations restore balance after insertions and deletions.

4. **Search and Traversal Operations**

   ■ Implemented **search** operation to efficiently locate nodes.
   ■ Implemented **in-order, pre-order, and post-order traversals**, showing sorted order for in-order traversal.

5. **Handling of Edge Cases**

   ■ Successfully managed:
   - Insertion into an empty tree
   - Deletion of root node
   - Balancing after consecutive insertions or deletions
   - Searching for non-existing nodes

6. Performance and Efficiency

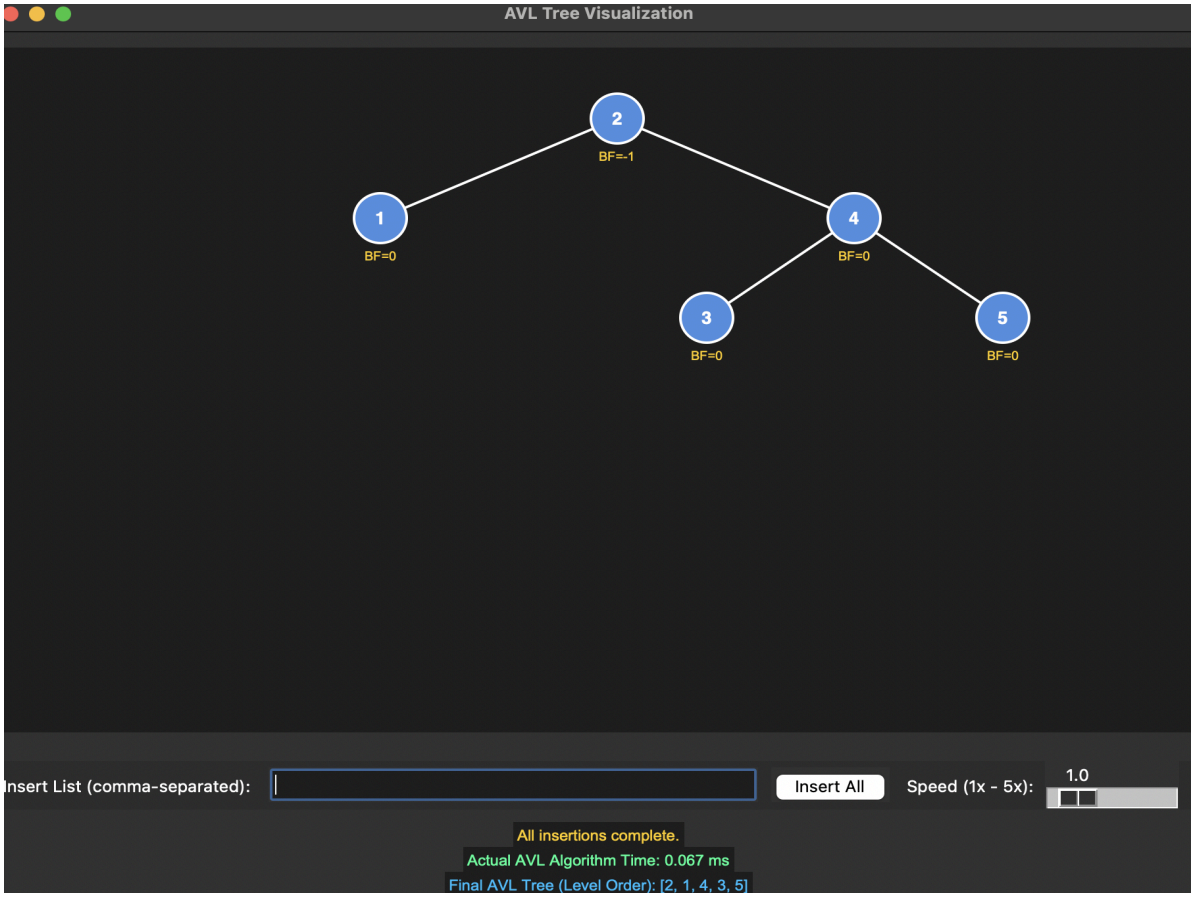
- Verified that all operations maintain **logarithmic time complexity O(log n)**.
- Observed improved efficiency compared to unbalanced BSTs, especially for large datasets.

7. Learning Outcomes

- Gained practical experience in **recursive algorithms** and **tree rotations**.
- Developed a deeper understanding of **balanced data structures** and their importance in real-world applications.
- Learned to **analyze algorithm complexity** and design robust, efficient data structures.

The following is an example of AVL tree construction:
Input: [1,2,3,4,5]

Output: [2,1,4,3,5]



3. Graph Coloring(Backtracking & Greedy):

The project demonstrates the implementation and analysis of graph coloring algorithms using both Backtracking and Greedy approaches. Graph coloring assigns colors to vertices of a graph such that no two adjacent vertices share the same color, which has applications in scheduling, register allocation, and map coloring.

Key Outcomes:

1. **Understanding of Graph Coloring Problem**
    - Gained a strong understanding of the **graph coloring problem**, its constraints, and applications.
    - Learned how to represent graphs using **adjacency lists and matrices**.

2. **Implementation of Backtracking Algorithm**
    - Implemented **Backtracking algorithm** to find a valid coloring using a given number of colors.
    - Capable of finding **all possible solutions** for small graphs.
    - Demonstrated understanding of **recursive solution exploration and pruning** to avoid invalid assignments.

3. **Implementation of Greedy Algorithm**
    - Implemented **Greedy coloring algorithm**, which assigns colors to vertices in a specific order.
    - Efficient for **large graphs**, providing a **fast, although not always optimal**, solution.
    - Showed how vertex ordering affects the number of colors used.

4. **Comparison of Algorithms**
    - Observed that **Backtracking guarantees optimal coloring** but can be **time-consuming** for large graphs.
    - Note that **Greedy is faster** but may not always use the minimum number of colors.
    - Analyzed **trade-offs between efficiency and optimality**.

5. **Handling Various Graph Types**
    - Successfully colored **different types of graphs** including:
        - Sparse graphs
        - Dense graphs
        - Cyclic graphs
        - Complete graphs
    - Demonstrated robustness of algorithms across graph structures.

6. **Visualization (Optional)**
    - Provided visual representation of coloring assignments for better understanding.

- Highlighted differences between Backtracking (step-by-step exploration) and Greedy (immediate assignment) approaches.

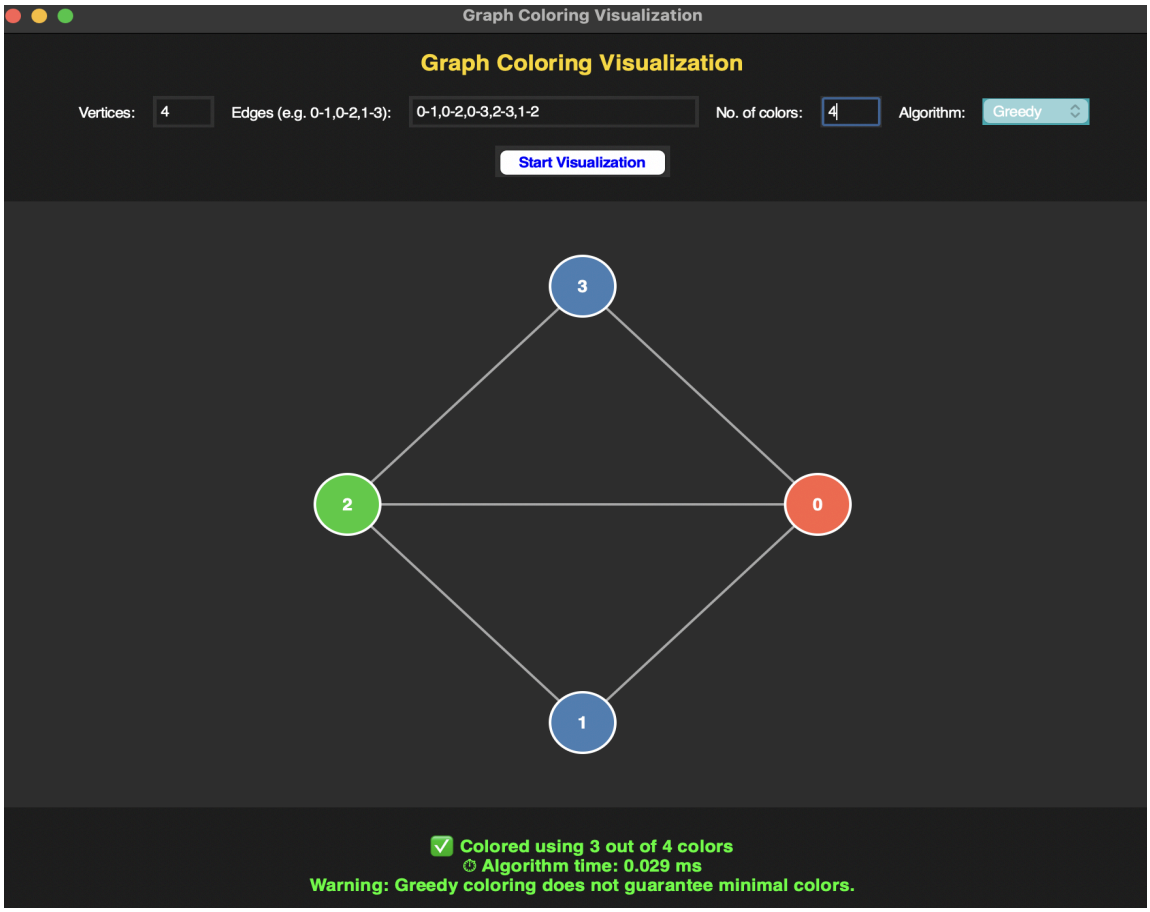
7. **Performance Analysis**

   - Measured **execution time** and **number of colors used** for each algorithm.
   - Showed efficiency differences between Backtracking and Greedy on graphs of varying size and density.

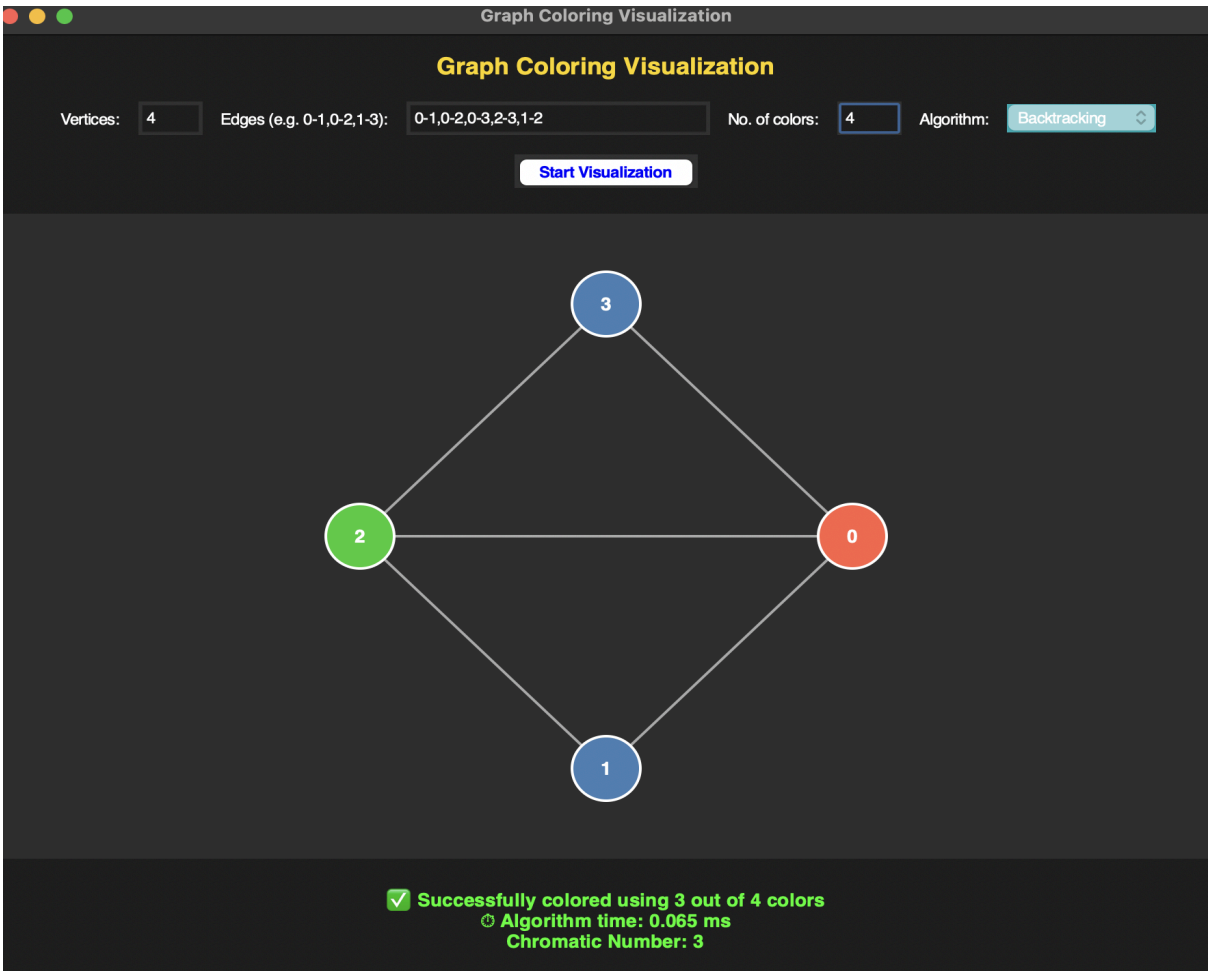
8. Learning Outcomes

   - Gained practical experience in **recursive problem solving**, **algorithm design**, and **graph theory**.
   - Developed skills in **analyzing algorithm complexity** (time and space) and understanding trade-offs.

The following is an example of Graph coloring:
Input → No of vertices: 4, Edges: 0-1,0-2,0-3,2-3,1-2, No of colours:4
Greedy:



Backtracking:



Note: The time taken by Greedy(0.029) is less than the time taken by Backtracking(0.065) in the above example.

References:

- ChatGPT for visualization and test cases.
- Tutorialpoints for images of AVL tree rotations: https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm