

**SCHOOL OF COMPUTER SCIENCE AND ENGINEERING**

**COURSE: CSE 316 OPERATING SYSTEM**

*SUBMITTED TO:* **Dr. Hardeep Kaur**

**TOPIC**

“**Automated Deadlock Detection Tool**”

|  |  |
| --- | --- |
| A. Full name | **Kriti Kumari** |
| Reg No. | **12322865** |
| Roll No | **26** |
| B. Full name | **Deeksha** |
| Reg No. | **12305940** |
| Roll No | **08** |
| C. Full name | **Abhishek Bathnotra** |
| Reg No. | **12319834** |
| Roll No | **29** |

**Table of Contents**

1. **Project Overview**
   * Introduction
   * Objective of the Project
   * Scope of the Project
2. **Module-Wise Breakdown**

* Graphical User Interface Module
* System Monitoring Module
* Process Control Module

1. **Functionalities**
2. **Technology Used**
   * Programming Languages
   * Libraries and Tools
   * Other Tools: [e.g., GitHub for version control]
3. **Flow Diagram**
4. **Revision Tracking on GitHub**
   * Repository Name
   * GitHub Link
5. **Conclusion and Future Scope**
   * Conclusion
   * Future Scope
6. **References**

**Appendix**

* + A. AI-Generated Project Elaboration/Breakdown Report
  + B. Problem Statement
  + C. Solution/Code

**1. Project Overview**

**Introduction**

Deadlocks are a critical problem in operating systems and concurrent computing environments, where multiple processes get stuck indefinitely, waiting for resources held by one another. To address this issue, our project, **Automated Deadlock Detection Tool**, provides an efficient way to identify and resolve deadlocks in a system.

This tool is designed to analyze resource allocation and process dependencies to detect potential deadlocks before they impact system performance. The tool follows a structured approach, incorporating **graph-based deadlock detection algorithms** to examine cycles in resource allocation graphs. If a cycle is detected, the system identifies the processes involved in the deadlock and suggests resolution mechanisms, such as process termination or resource preemption.

The implementation of this tool involves Python programming, leveraging essential libraries for data handling and visualization. GitHub is used for version control, ensuring structured development with multiple revisions and clear commit messages. The project follows a systematic workflow where new features are implemented using branches, tested, and then merged into the main branch.

By integrating **automated deadlock detection**, this project significantly improves system reliability and efficiency. The tool can be beneficial for **operating system students, software engineers, and researchers** who want to understand and mitigate deadlocks in real-time environments.

This report covers the detailed implementation, **module-wise breakdown, functionalities, technologies used, flow diagram, revision tracking, and future scope** of the project. The aim is to provide a **clear, structured, and comprehensive** understanding of the **Automated Deadlock Detection Tool** and its importance in modern computing systems.

**Objective of the Project:**

The **Automated Deadlock Detection Tool** aims to enhance system reliability by identifying and resolving deadlocks in resource allocation scenarios. The primary objectives of this project are:

1. **Automated Deadlock Identification** – Develop a tool that can efficiently detect deadlocks in a system by analyzing resource allocation and process dependencies.
2. **Graph-Based Analysis** – Implement a Resource Allocation Graph (RAG) to model process-resource interactions and use cycle detection algorithms to identify deadlocks.
3. **Real-Time Detection** – Enable continuous monitoring of processes and resources to ensure that deadlocks are detected before they severely impact system performance.
4. **User-Friendly Output** – Provide clear and interpretable results, including visual representations of deadlocked processes and suggested resolution mechanisms.
5. **Efficient Resolution Mechanisms** – Suggest possible solutions to resolve deadlocks, such as process termination, resource preemption, or priority adjustments.
6. **Scalability & Extensibility** – Ensure the tool can handle different system configurations and be extended for future enhancements, including integration with larger system monitoring solutions.
7. **Educational and Practical Use** – Serve as a learning tool for students and a practical aid for software engineers dealing with deadlock situations in real-world applications.

**Scope of the Project:**

The **Automated Deadlock Detection Tool** is designed to be a versatile and efficient solution for identifying and managing deadlocks in resource allocation scenarios. The scope of this project includes:

1. **Deadlock Detection in Multi-Process Systems**
   * The tool can be applied to operating systems, database management systems, and distributed computing environments where multiple processes compete for resources.
2. **Application in Academic and Industrial Domains**
   * It serves as a **learning tool** for students studying **operating system concepts**, particularly in deadlock detection and prevention.
   * It can also be used in **real-world system administration** to monitor and resolve deadlocks in software applications.
3. **Graph-Based Deadlock Analysis**
   * The tool uses a **Resource Allocation Graph (RAG)** and cycle detection techniques to identify deadlocks effectively.
   * It supports **visual representation** of deadlocks for better understanding and debugging.
4. **Real-Time and Batch Processing Support**
   * The tool can operate in **real-time mode**, where it continuously monitors resource allocations, or in **batch mode**, where it analyzes logs for potential deadlocks.
5. **Scalability and Flexibility**
   * The project is designed to handle **small-scale and large-scale process environments** with multiple resource dependencies.
   * It can be extended to **support additional features**, such as deadlock prevention techniques and automated resolution mechanisms.
6. **Integration with System Monitoring Tools**
   * The tool can be integrated with **existing system monitoring frameworks** to provide enhanced deadlock detection capabilities.

**2. Module-Wise Breakdown**

The **Automated Deadlock Detection Tool** consists of several key modules, each responsible for handling specific functionalities. Below is a breakdown of the major modules:

**1. Graphical User Interface (GUI) Module**

**Purpose:**

* Provides an interactive interface for users to visualize process-resource relationships.
* Displays the **Resource Allocation Graph (RAG)** and highlights deadlocks.

**Features:**

* User-friendly dashboard to **input process and resource details**.
* **Graph visualization** for resource allocation and cycle detection.
* **Real-time updates** on deadlock status.
* **Alerts and notifications** when a deadlock is detected.

**2. System Monitoring Module**

**Purpose:**

* Continuously tracks resource allocation and process execution.
* Collects **real-time data** about system state to identify deadlocks.

**Features:**

* Monitors **resource requests and allocations** by processes.
* Detects **circular wait conditions** that indicate deadlocks.
* Maintains **logs of process-resource interactions** for analysis.
* Provides **reporting features** to view past deadlock occurrences.

**3. Process Control Module**

**Purpose:**

* Manages process execution and **handles deadlock resolution**.
* Ensures efficient resource allocation to prevent system slowdowns.

**Features:**

* Implements **deadlock detection algorithms** (e.g., Wait-for Graph, Banker's Algorithm).
* Provides **manual and automatic resolution options** for deadlocks.
* Allows **terminating or rolling back processes** to break deadlocks.
* Optimizes **resource reallocation** for better system performance.

**3. Functionalities**

The **Automated Deadlock Detection Tool** is designed to efficiently detect and resolve deadlocks in a computing system. Below are the key functionalities:

**1. Deadlock Detection**

* **Monitors system resources** and process allocations in real-time.
* Uses **graph-based algorithms** (such as Wait-for Graph) to **identify circular waits**.
* Highlights processes that are in a deadlock state.

**2. Graphical Representation of Process-Resource Allocation**

* Displays a **Resource Allocation Graph (RAG)** to visualize relationships between processes and resources.
* Uses **color-coded indicators** to show active processes and blocked processes.
* Updates dynamically to reflect system changes.

**3. Real-time System Monitoring**

* Continuously **tracks and logs** resource allocation and deallocation.
* Provides **status reports** for system administrators.
* Detects potential deadlocks **before they fully occur**.

**4. Deadlock Resolution Mechanisms**

* Offers **automatic and manual deadlock resolution** strategies.
* Users can **terminate a process or preempt resources** to resolve deadlocks.
* Suggests the **best possible resolution based on system state**.

**5. Logging and Reporting**

* Maintains a **history of deadlock occurrences** for analysis.
* Generates **reports** for administrators to improve resource management.
* Provides insights into **frequently blocked processes** and system inefficiencies.

**6. User-Friendly Interface**

* Simple and intuitive UI for **easy configuration and monitoring**.
* Allows users to **add, remove, and modify processes** dynamically.
* Provides **real-time alerts and notifications** for deadlocks.

**4. Technology Used**

The Automated Deadlock Detection Tool is developed using a combination of programming languages, libraries, and tools that enable efficient monitoring, detection, and resolution of deadlocks in a system.

1. **Programming Languages**

* Python – Used for implementing the core logic, deadlock detection algorithms, and system monitoring features.

2. **Libraries and Tools**

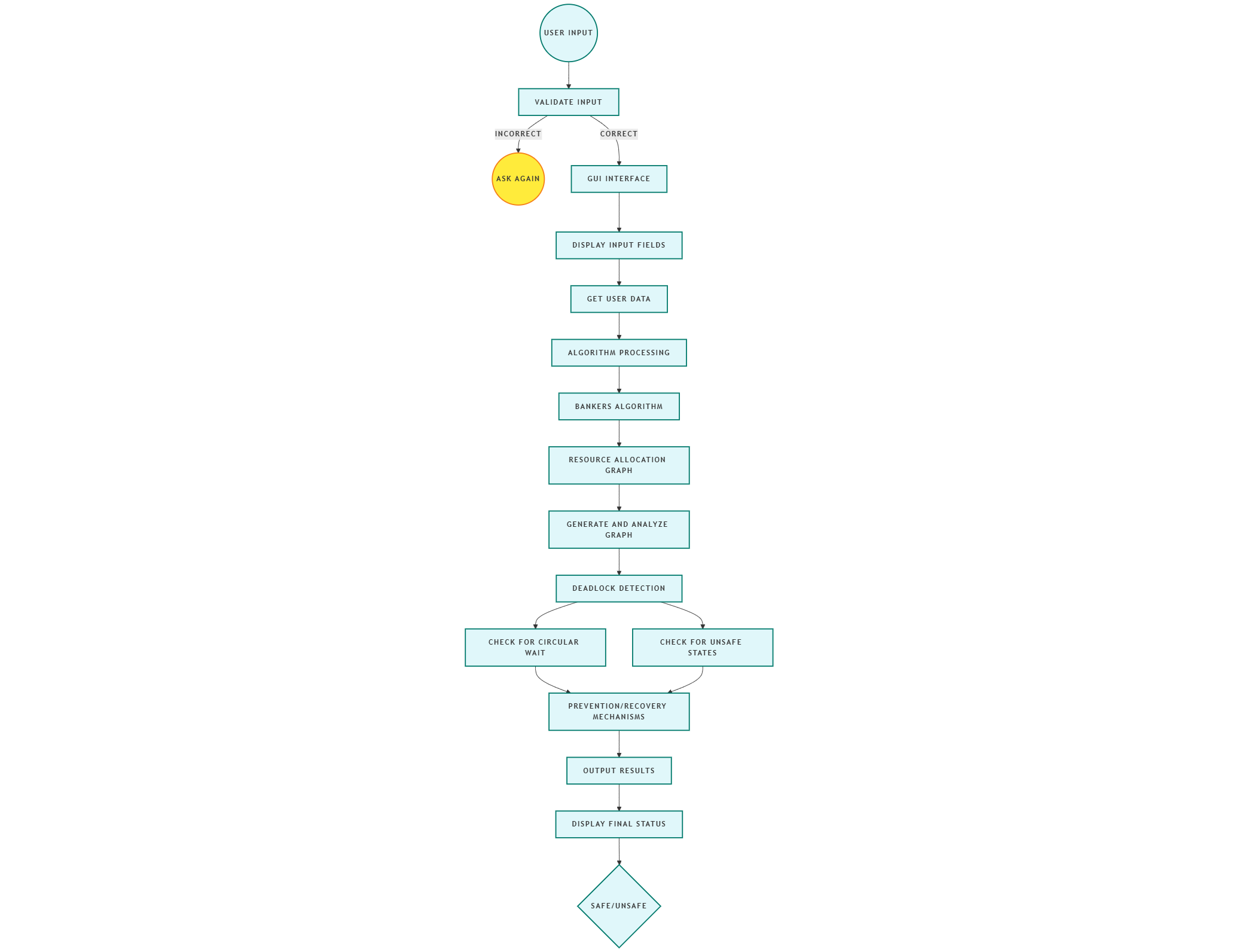
* **NumPy** – Used for matrix operations in the resource allocation and need matrix calculations.
* **NetworkX** – Used for constructing and analyzing resource allocation graphs (RAG) to visualize process-resource dependencies.
* **Matplotlib** – Used for visualizing process-resource allocation and deadlock situations.
* **Tkinter** – For developing the Graphical User Interface (GUI) to allow user interaction, input processing, and result display.
* **Threading** – Utilized to manage real-time interactions in the GUI.

3. **Development & Version Control Tools**

* Git & GitHub – Used for version control, collaboration, and maintaining the project repository.
* VS Code / PyCharm – IDEs used for coding, testing, and debugging.

**5. Flow Diagram**

**Flowchart Representation**



**6. Revision Tracking on GitHub**

**Repository Name:**

**kritiKriti-kumari221**

**Bottom of Form**

[**New**](https://github.com/new)

* [**Deadlock-Detection**](https://github.com/Kriti-kumari221/Deadlock-Detection)

**GitHub Link:**

**https://github.com/Kriti-kumari221/Deadlock-Detection**

**7. Conclusion and Future Scope**

**Conclusion:**

The **Automated Deadlock Detection Tool** successfully identifies, monitors, and resolves deadlocks in real-time using a graphical interface. The system efficiently detects circular wait conditions by analyzing **resource allocation graphs (RAGs)** and provides insights into process dependencies. The integration of **Python, Tkinter, and NetworkX** ensures a user-friendly experience while maintaining high computational efficiency. By automating deadlock detection, the tool reduces manual intervention and enhances system reliability.

**Future Scope:**

**1.** Real-Time Deadlock Prevention

* Extend the tool to implement deadlock prevention mechanisms, such as resource ordering or request-time analysis, to avoid deadlocks before they occur.

**2.**Support for Multiple Operating Systems

* Currently optimized for specific platforms, future versions can include support for MacOS and other UNIX-based systems.

**3**.Integration with Cloud Systems

* The tool can be extended to monitor cloud-based and distributed systems where deadlocks occur due to resource contention.

**4**.AI-Based Deadlock Prediction

* Implement machine learning algorithms to predict potential deadlocks based on historical process behavior and resource utilization trends.

**5**.Automated Resolution and Recovery

* Future updates can include self-healing mechanisms, such as process preemption and dynamic resource allocation, to automatically resolve deadlocks without human intervention.

1. **References**

Online Resources

* Python Official Documentation – <https://docs.python.org/3/>
* NetworkX Library – https://networkx.github.io/
* Tkinter Documentation – <https://docs.python.org/3/library/tkinter.html>
* GitHub Documentation – <https://docs.github.com/en>

Project Repository & Development Tools

Kriti-Kumari221, *Automated Deadlock Detection Tool Repository* **https://github.com/Kriti-kumari221/Deadlock-Detection**

* PyCharm IDE – <https://www.jetbrains.com/pycharm/>
* MinGW for Windows Subsystem – <https://www.mingw-w64.org/>

.

**B. Problem Statement: Real-Time Process Monitoring Dashboard.**

**C.** **Solution/Code:**

import numpy as np

import networkx as nx

import matplotlib.pyplot as plt

import tkinter as tk

from tkinter import ttk, messagebox

import matplotlib.patches as mpatches

class DeadlockDetection:

    def \_\_init\_\_(self, processes, resources, allocation, max\_need, available):

        self.processes = processes

        self.resources = resources

        self.allocation = np.array(allocation, dtype=int)

        self.max\_need = np.array(max\_need, dtype=int)

        self.available = np.array(available, dtype=int)

        self.need = self.max\_need - self.allocation

        print ("1st commit")

    def is\_safe\_state(self):

        work = self.available.copy()

        finish = [False] \* len(self.processes)

        safe\_sequence = []

        steps\_info = "\n🔍 \*\*Safe State Calculation Steps:\*\*\n\n"

        while len(safe\_sequence) < len(self.processes):

            allocated = False

            for i in range(len(self.processes)):

                if not finish[i] and np.all(self.need[i] <= work):

                    steps\_info += f"✅ Process P{i} can execute (Need ≤ Available)\n"

                    work += self.allocation[i]

                    finish[i] = True

                    safe\_sequence.append(f"P{i}")

                    allocated = True

                    break

            if not allocated:

                steps\_info += "❌ No process can proceed further, leading to DEADLOCK!\n"

                return False, [], steps\_info

        steps\_info += "\n✅ \*\*Safe Sequence:\*\* " + " ➡ ".join(safe\_sequence)

        return True, safe\_sequence, steps\_info

    def detect\_deadlock(self):

        is\_safe, safe\_sequence, steps\_info = self.is\_safe\_state()

        return not is\_safe, safe\_sequence, steps\_info

    def visualize\_graph(self):

        G = nx.DiGraph()

        # Add Nodes with Labels

        for i in self.processes:

            G.add\_node(f"P{i}", color='#4D9DE0')  # Process Node (Blue)

        for j in self.resources:

            G.add\_node(f"R{j}", color='#E15554')  # Resource Node (Red)

        edge\_colors = {}

        labels = {}

        # Add Edges (Process -> Resource Allocation & Resource -> Process Request)

        for i, p in enumerate(self.processes):

            for j, r in enumerate(self.resources):

                if self.allocation[i][j] > 0:

                    G.add\_edge(f"P{p}", f"R{r}", color='green', width=2.5)

                    labels[(f"P{p}", f"R{r}")] = "Allocated"

                if self.need[i][j] > 0:

                    G.add\_edge(f"R{r}", f"P{p}", color='red', width=2.5)

                    labels[(f"R{r}", f"P{p}")] = "Request"

        # Graph Layout (Circular for better readability)

        pos = nx.circular\_layout(G)

        node\_colors = [G.nodes[n]['color'] for n in G.nodes]

        # Set Figure Size & Background

        plt.figure(figsize=(9, 7), facecolor="#F7F7F7")

        # Draw Graph with Improved Styles

        edges = G.edges(data=True)

        edge\_colors = [d['color'] for \_, \_, d in edges]

        edge\_widths = [d['width'] for \_, \_, d in edges]

        nx.draw(G, pos, with\_labels=True, node\_color=node\_colors, node\_size=2800,

                font\_size=14, font\_weight="bold", edgecolors="black", linewidths=1.8)

        nx.draw\_networkx\_edges(G, pos, edgelist=G.edges(), edge\_color=edge\_colors, width=edge\_widths)

        nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=labels, font\_color='black', font\_size=11)

        # Add an Enhanced Legend

        legend\_p = mpatches.Patch(color='#4D9DE0', label="Process (P)")

        legend\_r = mpatches.Patch(color='#E15554', label="Resource (R)")

        legend\_alloc = mpatches.Patch(color='green', label="Allocated")

        legend\_request = mpatches.Patch(color='red', label="Request")

        plt.legend(handles=[legend\_p, legend\_r, legend\_alloc, legend\_request], loc="upper right", fontsize=11)

        plt.title("🔍 Deadlock Detection Graph", fontsize=16, fontweight="bold", color="#333")

        plt.show()

# GUI Implementation

class DeadlockGUI:

    def \_\_init\_\_(self, root):

        self.root = root

        self.root.title("🔍 Deadlock Detection System")

        self.root.geometry("700x600")

        self.root.configure(bg="#f0f0f0")

        style = ttk.Style()

        style.configure("TButton", font=("Arial", 12), padding=10)

        style.configure("TLabel", font=("Arial", 14), background="#f0f0f0")

        # Header

        ttk.Label(root, text="🔍 Deadlock Detection System", font=("Arial", 16, "bold")).pack(pady=10)

        ttk.Label(root, text="Enter system details below:", font=("Arial", 12)).pack(pady=5)

        # Input Fields

        self.processes\_entry = self.create\_labeled\_entry("Processes (e.g., 0 1 2 3)")

        self.resources\_entry = self.create\_labeled\_entry("Resources (e.g., 0 1)")

        self.allocation\_entry = self.create\_labeled\_entry("Allocation Matrix (comma-separated rows, space-separated values)")

        self.max\_need\_entry = self.create\_labeled\_entry("Max Need Matrix (comma-separated rows, space-separated values)")

        self.available\_entry = self.create\_labeled\_entry("Available Resources (space-separated)")

        # Check Deadlock Button

        ttk.Button(root, text="Check Deadlock", command=self.check\_deadlock, style="TButton").pack(pady=20)

    def create\_labeled\_entry(self, label\_text):

        frame = ttk.Frame(self.root)

        frame.pack(pady=5, fill="x", padx=20)

        ttk.Label(frame, text=label\_text, font=("Arial", 11)).pack(anchor="w")

        entry = ttk.Entry(frame, font=("Arial", 12))

        entry.pack(fill="x", padx=5, pady=3)

        return entry

    def check\_deadlock(self):

        try:

            # Get Inputs

            processes = list(map(int, self.processes\_entry.get().strip().split()))

            resources = list(map(int, self.resources\_entry.get().strip().split()))

            allocation = self.parse\_matrix(self.allocation\_entry.get(), len(processes), len(resources))

            max\_need = self.parse\_matrix(self.max\_need\_entry.get(), len(processes), len(resources))

            available = list(map(int, self.available\_entry.get().strip().split()))

            if len(available) != len(resources):

                messagebox.showerror("Input Error", "Available resources count must match resource count.")

                return

            # Run Deadlock Detection

            detector = DeadlockDetection(processes, resources, allocation, max\_need, available)

            is\_deadlock, safe\_sequence, steps\_info = detector.detect\_deadlock()

            if is\_deadlock:

                messagebox.showerror("❌ Deadlock Detected", f"⚠ The system is in a DEADLOCK state!\n\n{steps\_info}")

            else:

                messagebox.showinfo("✅ Safe State", f"🎉 The system is in a SAFE state!\n\n{steps\_info}")

            detector.visualize\_graph()

        except Exception as e:

            messagebox.showerror("Input Error", f"Invalid input format! Please check your entries.\nError: {str(e)}")

    def parse\_matrix(self, matrix\_str, rows, cols):

        matrix = [list(map(int, row.strip().split())) for row in matrix\_str.strip().split(",")]

        if len(matrix) != rows or any(len(row) != cols for row in matrix):

            raise ValueError("Matrix dimensions do not match input count.")

        return matrix

# Run GUI

root = tk.Tk()

app = DeadlockGUI(root)

root.mainloop()