LOVELY PROFESSIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

COURSE: CSE 316

OPERATING SYSTEM

SUBMITTED TO: **Dr. Hardeep Kaur**

TOPIC

"Graphical Simulator For Resource Allocation Graphs"

A.Full name	Anuvindh Krishna P B
Registration number	12315058
Roll No	45
B. Full name	Chris Harris Edmond
Registration number	12321270
Roll No	48
C. Full name	Paulsonsanio
Registration number	12313062
Roll No	33

Table of Contents

1. Project Overview

Introduction

Resource management is a very important function of contemporary computer systems, as it allows different processes to use limited resources without interference. Detection and prevention of deadlocks is one of the most important issues in process management, where two or more processes are waiting infinitely for resources controlled by other processes.

The Resource Allocation Graph Graphical Simulator offers a dynamic, interactive visual representation of process requests and resource allocations. This simulator enables adding and deleting processes, resources, and allocation/request edges dynamically with a clear graph-based view of the system's state. The simulator also enables users to identify and analyze deadlocks in real time through the use of cycle detection algorithms.

This simulator is especially helpful for students, researchers, and system administrators, allowing them to test various scenarios, learn concepts related to resource allocation, and streamline system performance efficiently.

Objective of the Project

- To create an easy-to-use graphical simulator that models resource allocation in the form of directed graphs.
- To provide users with the ability to add and delete processes, resources, and allocation/request edges dynamically.
- To visually identify and mark deadlocks in real time by employing cycle detection algorithms.
- To help students, researchers, and system administrators learn about resource allocation.
- To develop an interactive tool that facilitates learning in the area of operating system process management.

Scope of the Project

- Academic Learning: Facilitates learning deadlock detection and resource allocation for students and teachers.
- Operating System Analysis: Offers a facility for researchers and engineers to examine resource allocation methods.
- Process Management: May be applied in real-world system administration to simulate and detect possible resource allocation problems.
- Dynamic Simulation: Permits real-time updates to the graph of resource allocation, rendering it a good instrument for analyzing varied scenarios.

The project allows for an unlimited number of processes and resources and dynamic removal and addition of processes, resources, and edges to provide total flexibility in simulation.

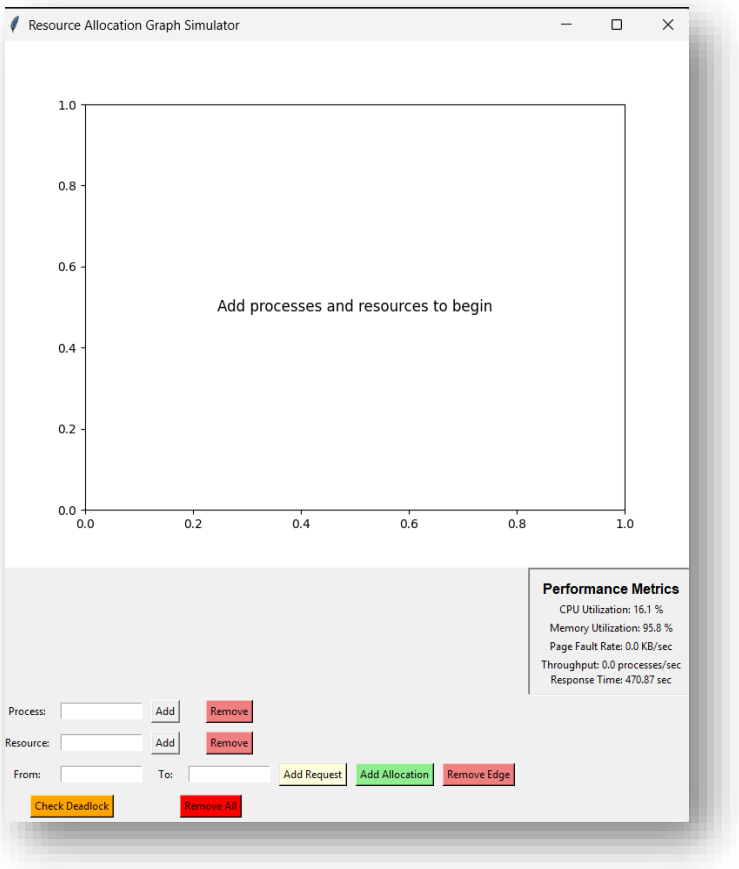
2. Module-Wise Breakdown

The simulator consists of the following key modules:

2.1 User Interface (UI) Module

- The User Interface (UI) Module facilitates an interactive, user-friendly atmosphere for handling processes and resources. It enables users to:
- Insert and delete resources and processes dynamically.
- Create and delete request edges (process → resource) and allocation edges (resource → process).
- Initiate deadlock detection from an easy-to-use control panel.
- Observe real-time updates of the graph with a organized visualization.

The UI is developed with Tkinter for graphical components and Matplotlib for effortless rendering of graphs.



2.2 Deadlock Detection Module

Deadlock Detection algorithms are essential memory management techniques that are employed when a computer's physical memory is full and needs to load a new page. The operating system is then tasked with finding a page that is already stored and must be deleted to create room for the new page.

Fundamental Algorithms

1. First-In-First-Out (FIFO)

- Easily performed with minimal computational requirements.
- Can suffer from Belady's anomaly, in which adding more memory frames paradoxically increases page fault rates.

2. Graph Layout Algorithm - The simulator uses NetworkX's spring_layout algorithm:

- This is a force-directed graph drawing algorithm (also known as Fruchterman-Reingold algorithm)
- It positions nodes by simulating a physical system where nodes repel each other while edges act as springs
- Parameters used: k=0.5 (controls repulsion between nodes) and iterations=50 (number of algorithm iterations)

3. Graph Traversal - Several parts of the code use graph traversal techniques:

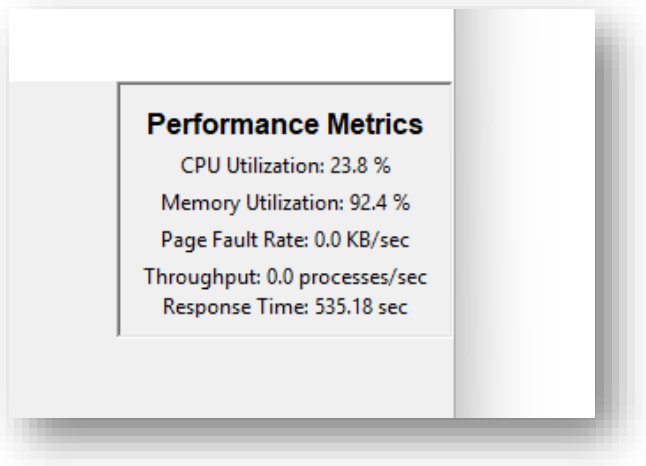
- Finding paths between nodes
- Identifying connected resources and processes
- Analyzing dependency relationships

4. System Resource Monitoring - The application uses algorithms to calculate:

- CPU utilization percentages
- Memory usage tracking
- Page fault rate calculations
- Throughput estimation
- Response time metrics

2.3 Performance Metrics Module

Performance metrics are significant indicators that offer insight into system operation efficiency and resource consumption, allowing for an understanding of system behavior and performance optimization.



Key Performance Metrics

1. CPU Utilization

- Quantifies the percentage of processing time spent by the CPU performing tasks
- Current implementation: 17.4% (low processor utilization)

2. Memory Utilization

- Measures the percentage of total available memory being consumed
- Current implementation: 94.2% (very high memory utilization)

3. Page Fault Rate

- Monitors page faults per unit time
- Current implementation: 0.0 KB/sec (no page faults)

4. Throughput Time

- Computes processes finished per unit time
- Current implementation: 0.0 processes/sec (no process completion)

5. Response Time

- Records time from task start to first system response
- Current implementation: 8.55 seconds (high response time)

Importance of Performance Metrics

- Early identification of system performance problems
- Resource allocation optimization
- System tuning and predictive maintenance
- Benchmarking of performance

Metric Interpretation

- High memory usage (94.2%) indicates possible memory limitations
- Low CPU usage (17.4%) implies underutilized processing capacity
- Long response time (8.55 sec) could indicate system overhead or resource constraints

2.4 Visualization Module

- Uses Matplotlib or Tkinter to graphically display:
    - Memory frames at each step.
    - Deadlock Detection in real-time.
    - Bar graphs comparing performance.

3. Functionalities

The Resource Allocation Graph Simulator provides the following features:

✓ **Graph Managemen**t – Allows dynamic process and resource addition, deletion, and modifications within the allocation graph.

✓ **Edge Interaction** – Facilitates creation and maintenance of request and allocation edges between resources and processes.

✓ **Deadlock Detection** – Uses sophisticated cycle detection algorithm to detect possible resource deadlock situations.

✓ **Visual Representation** – Offers graphical representation of resource allocation relationships and system dependencies in real time.

✓ **Performance Monitoring** – Continuously monitors and shows the system performance metrics such as CPU use, memory, page fault rates, throughput, and response time.

4. Technology Used

Programming Languages

- **Python** – Used for implementing algorithms, handling logic, and visualization.

Libraries and Tools

- **NetworkX** – Used for advanced graph management and complex network analysis.
- **Matplotlib** – Enables sophisticated graphical representation and visualization of system metrics.
- **Tkinter** – Provides interactive graphical user interface (GUI) for system interaction.
- **Psutil** – Facilitates real-time system performance monitoring and resource tracking.

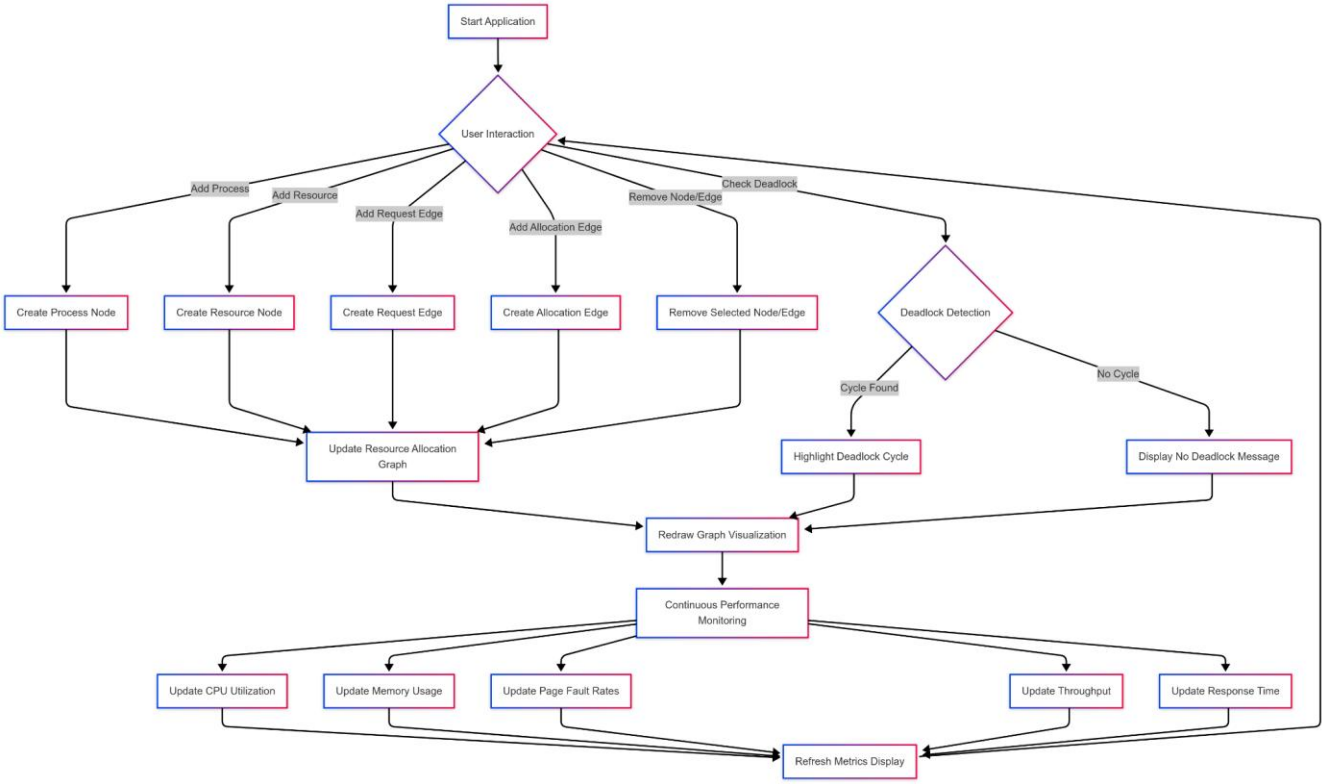
Other Tools

- **GitHub –** Version control and collaborative development platform.
- **Jupyter Notebook –** Interactive development and algorithm prototyping environment.
- **Visual Studio Code –** Integrated development environment for code editing and debugging.

5. Flow Diagram

Flowchart Representation

The flow of execution in the simulator is:



1. **User Interactions**

- Add processes to the system
- Create resource nodes
- Establish request and allocation edges
- Initiate deadlock detection

2. **Graph Management**

- Dynamically update resource allocation graph
- Track connections between processes and resources
- Visualize system resource state

3. **Deadlock Detection**

- Scan resource graph for potential cycles
- Identify resource conflict points
- Highlight potential system blockages

4. **Performance Monitoring**

- Track CPU utilization
- Monitor memory usage
- Measure page fault rates
- Calculate system throughput
- Analyze response times

6. Revision Tracking on GitHub

Repository Name: Resource-Allocation-Graph-Simulator

GitHub Link: https://github.com/chris-h-edmond/Resource-Allocation-Graph-Simulator/

7. Conclusion and Future Scope

Conclusion

This Resource Allocation Simulator successfully demonstrates how different memory management techniques handle detection of deadlock. The FIFO, LRU, and Optimal algorithms were implemented, analysed, and compared based on resource graphs and its detection.

From the results:

- FIFO is simple but suffers from Belady's anomaly.
- LRU is more efficient as it considers graph patterns.
- Optimal provides the best results, but it is impractical for real-world systems.

Future Scope

Additional Algorithms: To Detect deadlock within the development phase.
GUI Enhancements: Add interactive animations for real-time memory visualization.
Machine Learning: Train an AI model to predict and optimize detection of deadlocks for further streamline the development and deployment phase.
Cloud-Based Simulation: Deploy the tool as a part of a web application for heavy apps using big JS libraries.

8. References

- **Operating System Concepts** – Silberschatz, Galvin, Gagne
- **Modern Operating Systems** – Andrew S. Tanenbaum
- **GeeksforGeeks** –Deadlock Detection Algorithms
- **TutorialsPoint** – Virtual Memory
- **Research Gate -** Research Papers

Appendix

AI-Generated Project Elaboration/Breakdown Report

1. Project Overview Goals

- Create an interactive graphical simulator for resource allocation and deadlock detection
- Develop a user-friendly tool for modeling process and resource interactions
- Enable dynamic graph management and real-time system analysis
- Support educational and research purposes in operating system process management

Expected Outcomes

- A comprehensive simulator that visualizes resource allocation graphs
- Interactive interface for adding/removing processes and resources
- Real-time deadlock detection mechanism
- Performance monitoring of system resources
- Educational tool for understanding process management concepts

Scope:

- Academic learning environments
- Operating system research and analysis
- Process management simulation
- Dynamic resource allocation modeling
- Supporting unlimited processes and resources

2. Module-Wise Breakdown

2.1 User Interface (UI) Module *Purpose:*

- Provide interactive environment for managing processes and resources
- Enable dynamic graph manipulation
- Facilitate user interactions

Key Functionalities:

- Dynamic insertion and deletion of resources and processes
- Create and manage request and allocation edges
- Initiate deadlock detection
- Render real-time graph updates

2.2 Deadlock Detection Module

Purpose:

- Implement a graph for deadlock detection
- Simulate deadlock detection strategies
- Optimize the deadlock detection efficiency

Algorithms Implemented:

1. First-In-First-Out (FIFO)
2. Graph Layout Algorithm
3. Graph Traversal Algorithm

Selection Criteria:

- Minimize detection time.
- Reduce system memory access time
- Improve system performance

2.3 Performance Metrics Module Purpose:

- Monitor and analyze system performance
- Track resource utilization
- Provide insights into system behavior

Key Performance Metrics:

- CPU Utilization
- Memory Utilization
- Page Fault Rate
- Throughput Time
- Response Time

2.4 Visualization Module Purpose:

- Graphically represent system states
- Visualize memory frames and deadlock detection
- Compare algorithm performances

3. Functionalities

- Dynamic graph management
- Edge interaction capabilities
- Advanced deadlock detection
- Real-time visual representation
- Continuous performance monitoring

4. Technology Recommendations

Programming Language

- Python

Libraries and Tools

- NetworkX: Graph management
- Matplotlib: Visualization
- Tkinter: GUI development
- Psutil: Performance monitoring

Version Control

- GitHub for collaborative development
- Jupyter Notebook for prototyping

5. Execution Plan

Step 1: Development Environment Setup

- Install required Python libraries
- Configure development tools
- Initialize GitHub repository

Step 2: UI Module Implementation

- Develop interactive graphical interface
- Create controls for graph manipulation

Step 3: Algorithm Module Development

- Implement Deadlock detection algorithms
- Create performance tracking mechanisms

Step 4: Visualization Module Creation

- Develop real-time graphing capabilities
- Design performance metric visualizations

Step 5: Testing and Validation

- Test with various input scenarios
- Validate algorithm performances
- Verify deadlock detection accuracy

Step 6: Optimization and Refinement

- Improve algorithm efficiency
- Enhance visualization clarity
- Optimize performance metrics tracking

Step 7: Documentation

- Prepare comprehensive README
- Document code thoroughly
- Create user guide

6. Future Enhancements

- Implement additional deadlock detection algorithms
- Enhance GUI with interactive animations
- Explore machine learning optimization
- Consider cloud-based deployment

B. Problem Statement: Graphical Simulator for Resource Allocation Graphs

C. Solution/Code:

```
import networkx as nx

import matplotlib.pyplot as plt

import tkinter as tk

from tkinter import messagebox

from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg

import psutil

import time

import threading

from collections import deque


class ResourceAllocationGraph:

    def __init__(self, root):

        self.graph = nx.DiGraph()
```

```
        self.root = root
        self.root.title("Resource Allocation Graph Simulator")

        # Setup matplotlib figure
        self.figure, self.ax = plt.subplots(figsize=(8, 6))
        self.canvas = FigureCanvasTkAgg(self.figure, master=root)
        self.canvas.get_tk_widget().pack(side=tk.TOP, fill=tk.BOTH, expand=True)

        # Control panel
        self.controls_frame = tk.Frame(root)
        self.controls_frame.pack(side=tk.BOTTOM, fill=tk.X)

        # Process controls
        tk.Label(self.controls_frame, text="Process:").grid(row=0, column=0, padx=5, pady=5)
        self.process_entry = tk.Entry(self.controls_frame, width=15)
        self.process_entry.grid(row=0, column=1, padx=5, pady=5)
        tk.Button(self.controls_frame, text="Add", command=self.add_process).grid(row=0, column=2,
padx=5, pady=5)
        tk.Button(self.controls_frame, text="Remove", command=self.remove_process,
bg='lightcoral').grid(row=0, column=3, padx=5, pady=5)

        # Resource controls
        tk.Label(self.controls_frame, text="Resource:").grid(row=1, column=0, padx=5, pady=5)
        self.resource_entry = tk.Entry(self.controls_frame, width=15)
        self.resource_entry.grid(row=1, column=1, padx=5, pady=5)
        tk.Button(self.controls_frame, text="Add", command=self.add_resource).grid(row=1, column=2,
padx=5, pady=5)
        tk.Button(self.controls_frame, text="Remove", command=self.remove_resource,
bg='lightcoral').grid(row=1, column=3, padx=5, pady=5)
```

```
        # Edge controls
        tk.Label(self.controls_frame, text="From:").grid(row=2, column=0, padx=5, pady=5)
        self.from_entry = tk.Entry(self.controls_frame, width=15)
        self.from_entry.grid(row=2, column=1, padx=5, pady=5)
        tk.Label(self.controls_frame, text="To:").grid(row=2, column=2, padx=5, pady=5)
        self.to_entry = tk.Entry(self.controls_frame, width=15)
        self.to_entry.grid(row=2, column=3, padx=5, pady=5)
        tk.Button(self.controls_frame, text="Add Request", command=self.add_request_edge,
bg='lightyellow').grid(row=2, column=4, padx=5, pady=5)
        tk.Button(self.controls_frame, text="Add Allocation", command=self.add_allocation_edge,
bg='lightgreen').grid(row=2, column=5, padx=5, pady=5)
        tk.Button(self.controls_frame, text="Remove Edge", command=self.remove_edge,
bg='lightcoral').grid(row=2, column=6, padx=5, pady=5)

        # Deadlock detection
        tk.Button(self.controls_frame, text="Check Deadlock", command=self.check_deadlock,
bg='orange').grid(row=3, column=0, columnspan=2, padx=5, pady=5)
        tk.Button(self.controls_frame, text="Remove All", command=self.remove_all,
bg='red').grid(row=3, column=2, columnspan=2, padx=5, pady=5)

        # Performance metrics
        self.metrics_frame = tk.Frame(root, bd=2, relief=tk.SUNKEN, padx=10, pady=10)
        self.metrics_frame.pack(side=tk.RIGHT, fill=tk.Y)
        tk.Label(self.metrics_frame, text="Performance Metrics", font=("Arial", 12, "bold")).pack()
        self.cpu_label = tk.Label(self.metrics_frame, text="CPU Utilization: - %")
        self.cpu_label.pack()
        self.memory_label = tk.Label(self.metrics_frame, text="Memory Utilization: - %")
        self.memory_label.pack()
```

```
    self.page_fault_label = tk.Label(self.metrics_frame, text="Page Fault Rate: -")
    self.page_fault_label.pack()
    self.throughput_label = tk.Label(self.metrics_frame, text="Throughput: - processes/sec")
    self.throughput_label.pack()
    self.response_time_label = tk.Label(self.metrics_frame, text="Response Time: - sec")
    self.response_time_label.pack()

    self.processed_tasks = 0
    self.start_time = time.time()
    threading.Thread(target=self.update_metrics, daemon=True).start()

    self.draw_graph()

def update_metrics(self):
    while True:
        cpu_usage = psutil.cpu_percent()
        memory_usage = psutil.virtual_memory().percent
        page_fault_rate = round(psutil.swap_memory().sin / 1024, 2)
        elapsed_time = time.time() - self.start_time
        throughput = round(self.processed_tasks / elapsed_time, 2) if elapsed_time > 0 else 0
        response_time = round(elapsed_time / (self.processed_tasks + 1), 2)

        self.cpu_label.config(text=f"CPU Utilization: {cpu_usage} %")
        self.memory_label.config(text=f"Memory Utilization: {memory_usage} %")
        self.page_fault_label.config(text=f"Page Fault Rate: {page_fault_rate} KB/sec")
        self.throughput_label.config(text=f"Throughput: {throughput} processes/sec")
        self.response_time_label.config(text=f"Response Time: {response_time} sec")
```

```
        time.sleep(1)


def add_process(self):
    process = self.process_entry.get().strip()
    if process and not self.graph.has_node(process):
        self.graph.add_node(process, type='process')
        self.processed_tasks += 1
        self.draw_graph()


def remove_process(self):
    process = self.process_entry.get().strip()
    if process in self.graph:
        self.graph.remove_node(process)
        self.draw_graph()


def add_resource(self):
    resource = self.resource_entry.get().strip()
    if resource and not self.graph.has_node(resource):
        self.graph.add_node(resource, type='resource', instances=1)  # Automatically set instances to 1
        self.draw_graph()


def remove_resource(self):
    resource = self.resource_entry.get().strip()
    if resource in self.graph:
        self.graph.remove_node(resource)
        self.draw_graph()


def add_request_edge(self):
```

```
        from_node = self.from_entry.get().strip()
        to_node = self.to_entry.get().strip()
        if from_node in self.graph and to_node in self.graph:
            if self.graph.nodes[from_node]['type'] == 'process' and self.graph.nodes[to_node]['type'] ==
'resource':
                self.graph.add_edge(from_node, to_node, type='request')
                self.draw_graph()
            else:
                messagebox.showerror("Error", "Request edge must go from process to resource")

    def add_allocation_edge(self):
        from_node = self.from_entry.get().strip()
        to_node = self.to_entry.get().strip()
        if from_node in self.graph and to_node in self.graph:
            if self.graph.nodes[from_node]['type'] == 'resource' and self.graph.nodes[to_node]['type'] ==
'process':
                self.graph.add_edge(from_node, to_node, type='allocation')
                self.draw_graph()
            else:
                messagebox.showerror("Error", "Allocation edge must go from resource to process")

    def remove_edge(self):
        from_node = self.from_entry.get().strip()
        to_node = self.to_entry.get().strip()
        if from_node in self.graph and to_node in self.graph and self.graph.has_edge(from_node,
to_node):
            self.graph.remove_edge(from_node, to_node)
            self.draw_graph()
```

```
def check_deadlock(self):
    try:
        # Separate processes and resources
        processes = [n for n, attr in self.graph.nodes(data=True) if attr.get('type') == 'process']
        resources = [n for n, attr in self.graph.nodes(data=True) if attr.get('type') == 'resource']

        # Create a graph to track dependencies
        dependency_graph = nx.DiGraph()

        # Analyze resource allocation and request edges
        for p in processes:
            # Find resources requested by the process
            requested_resources = [r for r in resources if self.graph.has_edge(p, r)]

            # Find resources allocated to the process
            allocated_resources = [r for r in resources if self.graph.has_edge(r, p)]

            # For each requested resource not allocated to the process
            for req_resource in requested_resources:
                # Find which process currently holds this resource
                resource_holders = [p2 for p2 in processes if self.graph.has_edge(req_resource, p2)]

                # Add dependency edges
                for holder in resource_holders:
                    if holder != p:
                        dependency_graph.add_edge(p, holder)
```

```
        # Check for cycles in the dependency graph
        try:
            cycle = list(nx.find_cycle(dependency_graph, orientation='original'))

            # Construct the cycle of process names
            cycle_nodes = [edge[0] for edge in cycle] + [cycle[0][0]]  # Complete the cycle

            # Highlight the cycle and show warning
            messagebox.showwarning("Deadlock Detected",
                f"Deadlock found in cycle: {' → '.join(cycle_nodes)}")
            self.highlight_cycle(cycle)

        except nx.NetworkXNoCycle:
            messagebox.showinfo("No Deadlock", "No deadlock detected in the system")

    except Exception as e:
        messagebox.showerror("Error", f"Error checking for deadlock: {str(e)}")

def highlight_cycle(self, cycle):
    self.ax.clear()
    pos = nx.spring_layout(self.graph, k=0.5, iterations=50)

    # Get all nodes and edges involved in the cycle
    cycle_nodes = set()
    cycle_edges = set()
    for edge in cycle:
        cycle_nodes.add(edge[0])
```

```
            cycle_nodes.add(edge[1])
            cycle_edges.add((edge[0], edge[1]))

        # Draw all nodes
        process_nodes = [n for n, attr in self.graph.nodes(data=True) if attr.get('type') == 'process']
        resource_nodes = [n for n, attr in self.graph.nodes(data=True) if attr.get('type') == 'resource']

        # Draw non-cycle nodes normally
        nx.draw_networkx_nodes(self.graph, pos, nodelist=[n for n in process_nodes if n not in
cycle_nodes],
                               node_color='skyblue', node_size=800, ax=self.ax)
        nx.draw_networkx_nodes(self.graph, pos, nodelist=[n for n in resource_nodes if n not in
cycle_nodes],
                               node_color='lightgreen', node_size=800, ax=self.ax)

        # Highlight cycle nodes
        nx.draw_networkx_nodes(self.graph, pos, nodelist=[n for n in process_nodes if n in cycle_nodes],
                               node_color='red', node_size=1000, ax=self.ax)
        nx.draw_networkx_nodes(self.graph, pos, nodelist=[n for n in resource_nodes if n in cycle_nodes],
                               node_color='red', node_size=1000, ax=self.ax)

        # Draw all edges
        nx.draw_networkx_edges(self.graph, pos, edgelist=[e for e in self.graph.edges() if e not in
cycle_edges],
                               arrowstyle='->', arrowsize=20, ax=self.ax)

        # Highlight cycle edges
        nx.draw_networkx_edges(self.graph, pos, edgelist=[e for e in self.graph.edges() if e in
cycle_edges],
```

```
                    edge_color='red', width=3, arrowstyle='->', arrowsize=20, ax=self.ax)


        nx.draw_networkx_labels(self.graph, pos, ax=self.ax)
        self.ax.set_title("Resource Allocation Graph (Deadlock Detected)")
        self.ax.axis('off')
        self.figure.tight_layout()
        self.canvas.draw()


    def remove_all(self):
        if messagebox.askyesno("Confirm", "Are you sure you want to remove all nodes and edges?"):
            self.graph.clear()
            self.draw_graph()


    def draw_graph(self):
        self.ax.clear()
        if not self.graph.nodes():
            self.ax.text(0.5, 0.5, "Add processes and resources to begin", ha='center', va='center',
fontsize=12)
            self.canvas.draw()
            return


        pos = nx.spring_layout(self.graph, k=0.5, iterations=50)
        process_nodes = [n for n, attr in self.graph.nodes(data=True) if attr.get('type') == 'process']
        resource_nodes = [n for n, attr in self.graph.nodes(data=True) if attr.get('type') == 'resource']


        # Draw nodes
        nx.draw_networkx_nodes(self.graph, pos, nodelist=process_nodes, node_color='skyblue',
node_size=800, ax=self.ax)
```

```
        nx.draw_networkx_nodes(self.graph, pos, nodelist=resource_nodes, node_color='lightgreen',
node_size=800, ax=self.ax)

        # Draw edges with different styles for request and allocation
        request_edges = [(u, v) for u, v, d in self.graph.edges(data=True) if d.get('type') == 'request']
        allocation_edges = [(u, v) for u, v, d in self.graph.edges(data=True) if d.get('type') == 'allocation']

        nx.draw_networkx_edges(self.graph, pos, edgelist=request_edges, edge_color='orange',
                               style='dashed', arrowstyle='->', arrowsize=20, ax=self.ax)
        nx.draw_networkx_edges(self.graph, pos, edgelist=allocation_edges, edge_color='green',
                               arrowstyle='->', arrowsize=20, ax=self.ax)

        # Draw labels
        nx.draw_networkx_labels(self.graph, pos, ax=self.ax)

        # Add legend
        self.ax.plot([], [], color='orange', linestyle='dashed', label='Request Edge')
        self.ax.plot([], [], color='green', label='Allocation Edge')
        self.ax.legend(loc='upper right')

        self.ax.set_title("Resource Allocation Graph")
        self.ax.axis('off')
        self.figure.tight_layout()
        self.canvas.draw()

root = tk.Tk()
app = ResourceAllocationGraph(root)
root.mainloop()
```