

# PROJECT REPORT CSE316

On

## **“Graphical Simulator for Resource Allocation Graphs”**

Submitted by:-

Suraksha Sharma - 12319926

Raj – 12317144

Ronald William Joseph - 12324466

In partial fulfilment for the requirement of the award of the degree  
of  
“B.Tech. (Computer Science & Engineering)”



SCHOOL OF COMPUTER SCIENCE

**LOVELY PROFESSIONAL UNIVERSITY**

**PHAGWARA, PUNJAB**

## 1. Project Overview

This project is a **Graphical Simulator for Resource Allocation Graphs**, designed to analyze deadlock scenarios interactively. The application provides a visual representation of how resources are allocated to processes and allows users to detect potential deadlocks. The system employs cycle detection algorithms to identify deadlocks and offers a clear graphical representation of the resource allocation graph.

The primary goal of this project is to aid in understanding and managing deadlocks in an operating system environment by simulating real-world resource allocation cases.

## 2. Module-Wise Breakdown

- **main.py:**
  - Serves as the entry point of the application.
  - Initializes the graphical user interface (GUI).
  - Manages event handling and user interactions.
- **ui.py:**
  - Implements the GUI using PyQt6.
  - Provides buttons and input fields for adding/removing processes and resources.
  - Calls the appropriate functions from `graph_manager.py` to update the graph visualization dynamically.
  - Displays the state of the resource allocation graph.
- **graph\_manager.py:**
  - Manages the graph structure using NetworkX.
  - Handles process and resource allocation logic.
  - Implements cycle detection for deadlock identification.
  - Provides methods to add/remove processes and resources, update allocations, and check for cycles in the graph.

### 3. Functionalities

- **Process and Resource Management:**
  - Add processes to the system.
  - Add resources available for allocation.
  - Remove processes and resources as needed.
- **Resource Allocation and Deallocation:**
  - Allocate a resource to a process.
  - Release a resource from a process when it is no longer needed.
- **Deadlock Detection:**
  - Utilize cycle detection algorithms in NetworkX to determine if a deadlock is present.
  - Display an alert when a deadlock is detected.
- **Graph Visualization:**
  - Represent processes and resources as nodes in a graph.
  - Display resource allocation using directed edges.
  - Update the visualization dynamically as processes and resources change.

### 4. Technology Used

#### Programming Languages:

- Python (for application logic and GUI development)

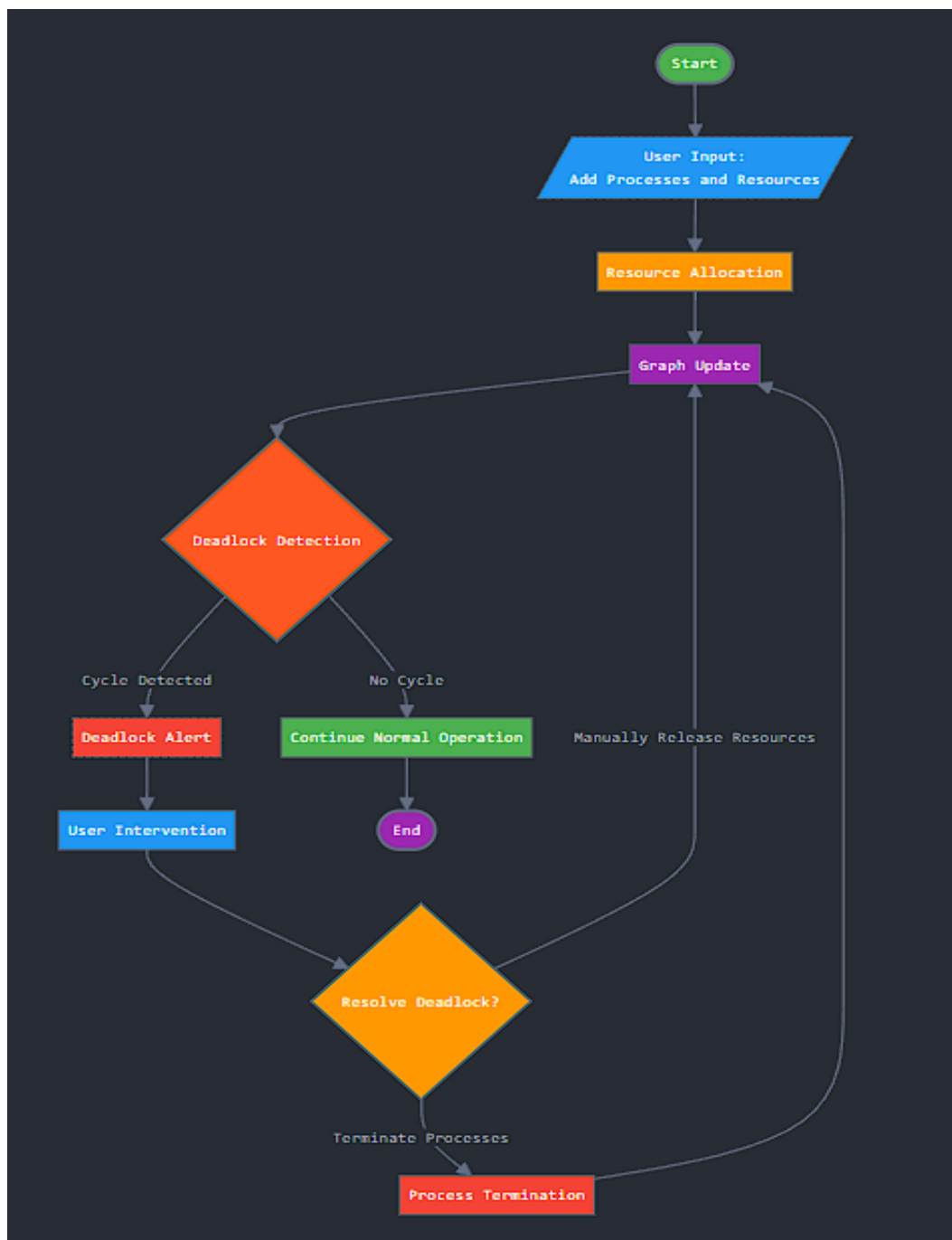
#### Libraries and Tools:

- **PyQt6:** Used for developing the GUI components.
- **NetworkX:** Used for representing and managing resource allocation graphs.
- **Matplotlib:** Used for rendering the graphical representation of the allocation graph.

## Other Tools:

- **GitHub:** Used for version control and collaborative development.
- **PyCharm/VS Code:** Used as the primary development environment.
- **Pip:** For dependency management.

## 5. Flow Diagram



### Process Flow:

1. **User Input:** The user adds processes and resources.
2. **Resource Allocation:** The user assigns resources to processes.
3. **Graph Update:** The system updates the visual graph representation.
4. **Deadlock Detection:** The program checks for cycles in the graph.
5. **Deadlock Alert:** If a cycle is found, a deadlock warning is displayed.
6. **User Intervention:** The user can manually release resources or terminate processes to resolve the deadlock.

## 6. Revision Tracking on GitHub

- **Repository Name:** Graphical-Simulator-for-Resource-Allocation-Graphs
- **GitHub Link:** <https://github.com/Ronald-William/Graphical-Simulator-for-Resource-Allocation-Graphs>
- **Version History:**
  - v1.0: Initial project setup with basic UI and graph logic.
  - v1.1: Added deadlock detection functionality.
  - v1.2: Improved UI for better user experience.
  - v1.3: Enhanced visualization with Matplotlib integration.
  - v1.4: Bug fixes and optimization of cycle detection algorithm.

## 7. Conclusion and Future Scope

This project successfully implements a resource allocation graph simulator with deadlock detection. The system efficiently visualizes how resources are allocated among processes and identifies potential deadlocks using cycle detection algorithms.

### Future Scope:

- **Enhanced Deadlock Resolution:** Implement automated deadlock recovery strategies such as preemptive resource allocation or process termination.
- **Multi-User Support:** Extend the application to support multiple users in a distributed environment.
- **Advanced UI Features:** Improve the GUI with interactive elements like drag-and-drop nodes and real-time graph updates.
- **Performance Optimization:** Optimize the cycle detection algorithm for handling large-scale graphs.

## 8. References

- NetworkX Documentation: <https://networkx.github.io/>
- PyQt6 Documentation: <https://www.riverbankcomputing.com/software/pyqt/>
- Matplotlib Documentation: <https://matplotlib.org/>

## 9. Appendix

### A. AI-Generated Project Elaboration/Breakdown Report

This project is a **Graphical Simulator for Resource Allocation Graphs**, designed to visualize and detect deadlocks dynamically. It consists of three main modules:

1. **Main Module (main.py)** – Initializes the GUI, manages user interactions, and connects UI elements with the logic layer.
2. **UI Module (ui.py)** – Uses PyQt6 to provide user controls for adding/removing processes and resources, updating the graph, and displaying deadlock alerts.
3. **Graph Manager (graph\_manager.py)** – Uses NetworkX to manage the resource allocation graph, implement cycle detection, and identify deadlocks.

### Functionality Flow

1. Users add processes and resources via the UI.
2. Resources are allocated to processes dynamically.
3. The system updates the graph in real time.
4. A cycle detection algorithm checks for deadlocks.
5. If a deadlock is found, an alert is displayed.

### **Key Features**

- **Real-time Graph Updates** – Dynamic visualization using Matplotlib.
- **Efficient Deadlock Detection** – Optimized cycle detection with NetworkX.
- **Interactive Controls** – Users can allocate, deallocate, and modify processes/resources easily.

### **Future Enhancements**

- AI-based deadlock prediction.
- Cloud-based multi-user support.
- Advanced UI with drag-and-drop functionality.

## **B. Problem Statement:** Graphical Simulator for Resource Allocation Graphs

### **C. Solution Code:**

#### **i) “main.py”:**

```
import sys
sys.path.append(".")
from PyQt6.QtWidgets import QApplication
from ui import ResourceAllocationSimulator

if __name__ == "__main__":
    app = QApplication(sys.argv)
```

```
window = ResourceAllocationSimulator()
window.show()
sys.exit(app.exec())
```

## ii) “graph\_manager.py”

```
import networkx as nx
```

```
class GraphManager:
```

```
    def __init__(self):
```

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

```
        self.process_count = 0
```

```
        self.resource_count = 0
```

```
        self.resource_instances = {} # Stores total and available instances
```

```
        self.allocations = {} # {process: {resource: allocated_count}}
```

```
    def add_process(self):
```

```
        process_id = f"P{self.process_count}"
```

```
        self.graph.add_node(process_id)
```

```
        self.allocations[process_id] = {}
```

```
        self.process_count += 1
```

```
        return process_id
```

```
    def add_resource(self, instances):
```

```
        resource_id = f"R{self.resource_count}"
```

```
        self.graph.add_node(resource_id)
```

```

        self.resource_instances[resource_id] = {"total": instances, "available":
instances}

        self.resource_count += 1

        return resource_id

def allocate_resource(self, process, resource):

    if resource not in self.resource_instances or process not in self.allocations:

        return "does not exist"

    if self.resource_instances[resource]["available"] > 0:

        self.resource_instances[resource]["available"] -= 1

        self.allocations[process][resource] =
self.allocations[process].get(resource, 0) + 1

        self.graph.add_edge(resource, process) # Allocation edge

        return "allocated"

    # If no available instances, add a request edge

    self.graph.add_edge(process, resource, style='dashed', color='orange')

    return "not enough instances"

def release_resource(self, resource, process):

    if resource in self.allocations.get(process, {}) and
self.allocations[process][resource] > 0:

        self.resource_instances[resource]["available"] += 1

        self.allocations[process][resource] -= 1

        if self.allocations[process][resource] == 0:

```

```

del self.allocations[process][resource]

self.graph.remove_edge(resource, process) # Remove allocation edge

# Check if any process is waiting for this resource
waiting_processes = [p for p in self.graph.predecessors(resource)
                     if self.graph.edges[p, resource].get('style') == 'dashed']
if waiting_processes:
    next_process = waiting_processes[0] # Pick the first waiting process
    self.graph.remove_edge(next_process, resource) # Remove request
edge
    self.allocate_resource(next_process, resource) # Allocate resource

return True

return False

def remove_resource(self, resource):
    if resource in self.resource_instances:
        del self.resource_instances[resource] # Remove resource record
        self.graph.remove_node(resource) # Remove from the graph
        return True
    return False

def remove_process(self, process):
    if process in self.allocations:
        # Release all allocated resources
        allocated_resources = list(self.allocations[process].keys()) # Copy keys to
avoid mutation issues

```

```

    for resource in allocated_resources:
        self.release_resource(resource, process)

    del self.allocations[process] # Remove process record

    if process in self.graph.nodes:
        self.graph.remove_node(process) # Remove process from graph
        return True

    return False

```

### iii) “ui.py”

```

import sys

from PyQt6.QtWidgets import QLabel
from PyQt6.QtCore import Qt
import networkx as nx
import matplotlib.pyplot as plt

from PyQt6.QtWidgets import (QApplication, QMainWindow, QPushButton,
                              QGridLayout,
                              QWidget, QMessageBox, QInputDialog, QLabel)
from graph_manager import GraphManager

# Updated button style
BUTTON_STYLE = """
    QPushButton {

```

```

        background-color: #a677d9;
        color: white;
        font-size: 16px;
        border-radius: 5px;
        padding: 10px;
    }
    QPushButton:hover {
        background-color: #472b66;
    }
    QPushButton:pressed {
        background-color: #3a294d;
    }
}

```

```

class ResourceAllocationSimulator(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Resource Allocation Graph")
        self.setGeometry(100, 100, 800, 500)
        self.graph_manager = GraphManager()
        self.initUI()

    def initUI(self):
        central_widget = QWidget()
        layout = QGridLayout()

```

```
title = QLabel("Resource Allocation Graph Simulator")

title.setStyleSheet("font-size: 18px; font-weight: bold; text-align: center;
color: white;")
```

```
buttons = [
    ("Add Process", self.add_process),
    ("Add Resource", self.add_resource),
    ("Allocate/Request", self.manage_allocation),
    ("Release Allocation", self.release_resource),
    ("Remove Resource", self.remove_resource),
    ("Remove Process", self.remove_process),
    ("Check Deadlock", self.detect_deadlock)
]
```

```
row, col = 1, 0
for text, handler in buttons:
    btn = QPushButton(text)
    btn.clicked.connect(handler)
    btn.setStyleSheet(BUTTON_STYLE)
    btn.setMinimumHeight(80) # Make buttons larger
    layout.addWidget(btn, row, col)
    col += 1
    if col > 2:
        col = 0
        row += 1
```

```
layout.addWidget(title, 0, 0, 1, 3)
```

```

# Shift buttons slightly upward by reducing bottom row stretch
for i in range(row):
    layout.setRowStretch(i, 2) # Increase stretch factor for upper rows
layout.setRowStretch(row, 0) # Reduce stretch for last row to pull buttons
up

for i in range(3):
    layout.setColumnStretch(i, 1)

central_widget.setStyleSheet("background-color: #2E2E2E;") # Dark
background for better contrast
central_widget.setLayout(layout)
self.setCentralWidget(central_widget)

def add_process(self):
    process_id = self.graph_manager.add_process()
    QMessageBox.information(self, "Process Added", f"Added {process_id}")
    self.show_graph()

def add_resource(self):
    quantity, ok = QInputDialog.getInt(self, "Resource Instances", "Enter
number of instances:", 1, 1, 10, 1)
    if ok:
        resource_id = self.graph_manager.add_resource(quantity)
        QMessageBox.information(self, "Resource Added", f"Added
{resource_id} with {quantity} instances")

```

```
self.show_graph()
```

```
def manage_allocation(self):
```

```
    processes = [n for n in self.graph_manager.graph.nodes if  
n.startswith("P")]
```

```
    resources = [n for n in self.graph_manager.graph.nodes if n.startswith("R")]
```

```
    if not processes or not resources:
```

```
        QMessageBox.warning(self, "Error", "Create at least one process and  
one resource")
```

```
        return
```

```
        process, ok = QDialog.getItem(self, "Select Process", "Process:",  
processes, 0, False)
```

```
        if not ok: return
```

```
        resource, ok = QDialog.getItem(self, "Select Resource", "Resource:",  
resources, 0, False)
```

```
        if not ok: return
```

```
        status = self.graph_manager.allocate_resource(process, resource)
```

```
        if status == "not enough instances":
```

```
            self.graph_manager.graph.add_edge(process, resource,  
style="dashed") # Add waiting edge
```

```
            QMessageBox.information(self, "Request", f"{process} waiting for  
{resource}")
```

```
        else:
```

```
            QMessageBox.information(self, "Allocation", f"{resource} {status} to  
{process}")
```

```
self.show_graph() # Refresh graph after allocation
```

```
def release_resource(self):
```

```
    allocations = [(u, v) for u, v in self.graph_manager.graph.edges if  
u.startswith("R") and v.startswith("P")]
```

```
    if not allocations:
```

```
        QMessageBox.warning(self, "Error", "No resources allocated")
```

```
        return
```

```
    allocation_strs = [f"{u} → {v}" for u, v in allocations]
```

```
    selection, ok = QDialog.getItem(self, "Release Resource", "Select  
allocation:", allocation_strs, 0, False)
```

```
    if not ok: return
```

```
    resource, process = selection.split(" → ")
```

```
    success = self.graph_manager.release_resource(resource, process)
```

```
    if success:
```

```
        QMessageBox.information(self, "Released", f"Released {resource} from  
{process}")
```

```
self.show_graph() # Refresh graph after release
```

```
def remove_resource(self):
```

```
    resources = [n for n in self.graph_manager.graph.nodes if n.startswith("R")]
```

```
    if not resources:
```

```
        QMessageBox.warning(self, "Error", "No resources available to remove")
```

```
        return
```

```
resource, ok = QDialog.getItem(self, "Remove Resource", "Select  
Resource:", resources, 0, False)
```

```
if not ok:
```

```
    return
```

```
success = self.graph_manager.remove_resource(resource)
```

```
if success:
```

```
    QMessageBox.information(self, "Resource Removed", f"{resource} has  
been removed")
```

```
    self.show_graph()
```

```
def remove_process(self):
```

```
    processes = [n for n in self.graph_manager.graph.nodes if  
n.startswith("P")]
```

```
if not processes:
```

```
    QMessageBox.warning(self, "Error", "No processes available to remove")
```

```
    return
```

```
process, ok = QDialog.getItem(self, "Remove Process", "Select  
Process:", processes, 0, False)
```

```
if not ok:
```

```
    return
```

```
success = self.graph_manager.remove_process(process)
```

```
if success:
```

```
    QMessageBox.information(self, "Process Removed", f"{process} has  
been removed")
```

```

        self.show_graph()
    else:
        QMessageBox.warning(self, "Error", f"Failed to remove {process}")

def detect_deadlock(self):
    try:
        cycles = list(nx.simple_cycles(self.graph_manager.graph)) # Find all
cycles
        if cycles:
            QMessageBox.critical(self, "Deadlock Detected!", f"Deadlock cycles:
{cycles}")
        else:
            QMessageBox.information(self, "No Deadlock", "System is safe")
    except nx.NetworkXNoCycle:
        QMessageBox.information(self, "No Deadlock", "System is safe")

def show_graph(self):
    plt.clf()
    color_map = []
    labels = {}
    node_shapes = {}

    resource_instances = self.graph_manager.resource_instances

    for node in self.graph_manager.graph.nodes:
        if node.startswith("R"): # Resource node

```

```

labels[node] = f"{node} ({resource_instances.get(node, 0)})"
color_map.append("#9370DB")
node_shapes[node] = "s"
else: # Process node
    labels[node] = node
    color_map.append("#44b0f2")
    node_shapes[node] = "o"

pos = nx.spring_layout(self.graph_manager.graph)

# Draw nodes with different shapes
for shape in set(node_shapes.values()):
    nx.draw_networkx_nodes(self.graph_manager.graph, pos,
                           nodelist=[n for n in self.graph_manager.graph.nodes if
node_shapes[n] == shape],
                           node_shape=shape,
                           node_color=[color_map[i] for i, n in
enumerate(self.graph_manager.graph.nodes) if node_shapes[n] == shape],
                           node_size=2000)

# Draw normal edges (allocations)
nx.draw_networkx_edges(self.graph_manager.graph, pos,
                       edgelist=[(u, v) for u, v in self.graph_manager.graph.edges if
self.graph_manager.graph.edges[u, v].get('style') != 'dashed'],
                       edge_color="gray", arrowsize=20)

# Draw request edges (dashed orange)

```

```
nx.draw_networkx_edges(self.graph_manager.graph, pos,  
                        edgelist=[(u, v) for u, v in self.graph_manager.graph.edges if  
self.graph_manager.graph.edges[u, v].get('style') == 'dashed'],  
                        edge_color="orange", style='dashed', arrowsize=20)
```

```
nx.draw_networkx_labels(self.graph_manager.graph, pos, labels,  
font_size=10)
```

```
# **Add Legend**
```

```
legend_labels = {  
    "Process": plt.Line2D([0], [0], marker='o', color='w', markersize=10,  
markerfacecolor="#44b0f2"),  
    "Resource": plt.Line2D([0], [0], marker='s', color='w', markersize=10,  
markerfacecolor="#9370DB"),  
    "Allocation Edge": plt.Line2D([0], [0], color="gray", lw=2),  
    "Request Edge": plt.Line2D([0], [0], color="orange", lw=2,  
linestyle="dashed")  
}
```

```
plt.legend(handles=legend_labels.values(), labels=legend_labels.keys(),  
loc="upper right")
```

```
plt.pause(0.1) # Allow real-time updates
```

```
plt.draw()
```

```
plt.show()
```