Automated Deadlock Detection Tool

PROJECT REPORT

by

Satyam Kumar - 12322535

Madhav Jee - 12307965

Shubham Kumar - 12325677

(Section: K23PM)

(Roll No. 08,13,09)

School of Computer Science Engineering

Lovely Professional University

Jalandhar, Punjab

BONAFIDE CERTIFICATE

Certified that this project report "Automated Deadlock Detection Tool" is the Bonafide work of "SATYAM KUMAR, MADHAV JEE, SHUBHAM KUMAR" Who carried out the project work under my supervision.

Dr. Suman Rani

Assistant Professor

ID: 30932

1. Project Overview

A **deadlock** happens when multiple processes are waiting for resources that other processes are holding, creating a cycle where no process can proceed. This can slow down or completely freeze a system, making it inefficient and unreliable.

The **Automated Deadlock Detection Tool** helps to **identify and resolve deadlocks automatically** by analysing how processes and resources interact. It uses **graph-based cycle detection** to check if any processes are stuck in a deadlock. If a deadlock is found, the tool provides suggestions to fix it and even allows the user to **free up resources** to break the deadlock.

The tool also has a **visual representation** of process-resource dependencies, making it easier to understand the issue. This is especially useful for developers working on complex, multi-threaded, or distributed systems.

2. Module-Wise Breakdown

a. Data Processing Module

- This module processes **user input** that defines which process is using or waiting for which resource (e.g., "P1 -> R1" means **Process 1 is waiting for Resource 1**).
- It then builds a **Resource Allocation Graph (RAG)** to represent these dependencies in a structured way.

b. Deadlock Detection Module

- After the graph is created, this module **checks for cycles** in it.
- If a cycle is found, it means that a group of processes are waiting for each other's resources in a circular way—this is a **deadlock**.
- The tool immediately **alerts the user** and highlights the processes involved in the deadlock.

c. Deadlock Resolution Module

- Once a deadlock is found, the tool suggests ways to **resolve it**, such as:
  1. **Manual Fix** – The user decides which process should be stopped or which resource should be freed.

2. **Automatic Fix** – The tool **removes an edge** from the cycle (i.e., it **frees a resource**) to break the deadlock.

d. Visualization Module

- This module creates a **graphical representation** of process-resource dependencies.
- **Deadlock cycles are shown in red**, making them easy to spot.
- The visualization helps users understand how processes are interacting and what's causing the deadlock.

3. Functionalities

- **Monitors Process-Resource Allocations** – Keeps track of how processes and resources are assigned.
- **Detects Deadlocks Automatically** – Identifies deadlocks by checking for cycles in the resource graph.
- **Provides Alerts** – Notifies the user when a deadlock is detected.
- **Visual Representation** – Displays a **graph** of system dependencies.
- **Suggests or Applies Fixes** – Offers manual and automatic options to resolve deadlocks.

4. Technology Used

Programming Language:

Python – The tool is built using Python because of its strong support for graph processing and visualization.

Libraries and Tools:

NetworkX – Used to create and analyse the **Resource Allocation Graph (RAG)**.

Matplotlib – Used to **visualize** the graph and highlight deadlocks.

Other Tools:

GitHub – Used for **version control and collaboration**.

5. Flow Diagram

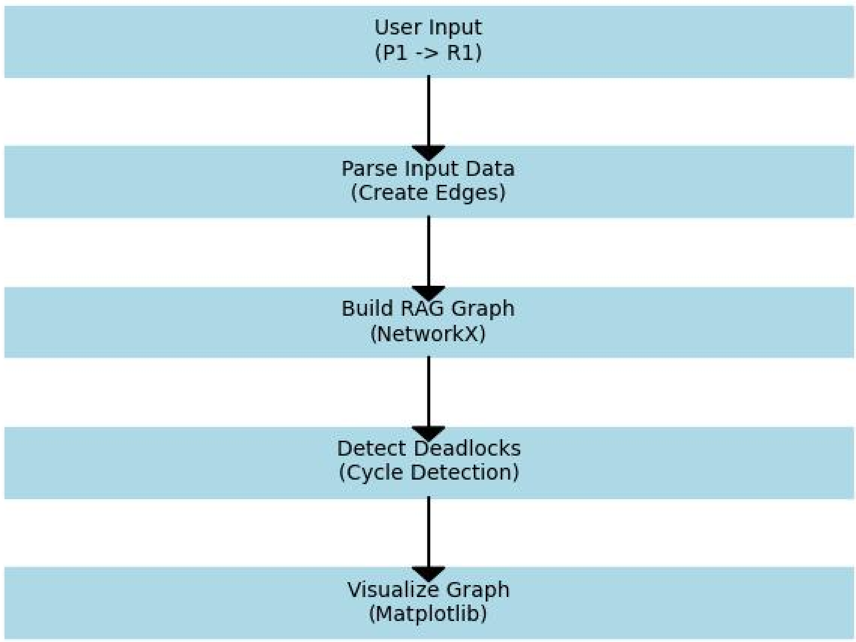


Figure 1: Flow Diagram

6. Revision Tracking on GitHub

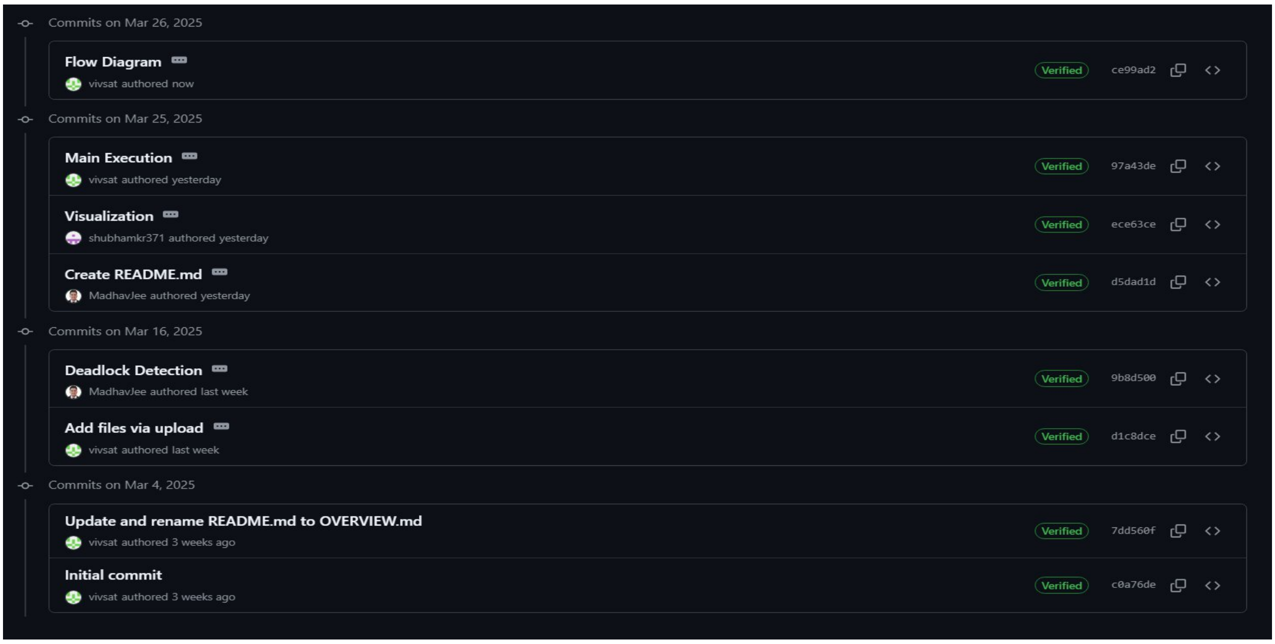
- **Repository Name:** *Automated Deadlock Detection Tool*
- **GitHub Link:**

  https://github.com/vivsat/Automated-Deadlock-Detection-Tool.git

GitHub Workflow:

1. **Branching Strategy:**
   - main branch – Stable version.
   - development branch – Ongoing improvements.
   - feature-* branches – Updates for specific parts of the tool.

2. **Commit Message Format:**
    - Added automatic deadlock detection.
    - Added data processing Module.
    - Added Visualization and Main execution file.
    - Fix some bugs in cyclic graphs.
3. **Versioning:**
    - Major releases: v1.0, v1.1, etc.



7. Conclusion and Future Scope

Conclusion

The **Automated Deadlock Detection Tool** makes it easy to **identify and fix deadlocks** in a system. By using **graph-based cycle detection**, it helps developers quickly spot and resolve deadlocks, improving software performance and stability. The tool also provides **visual graphs** to make debugging easier.

Future Scope

- **Predict Deadlocks Before They Happen** – Using **AI** to anticipate potential deadlocks.

- **Fully Automatic Deadlock Resolution** – Instead of just suggesting, the tool could resolve deadlocks **on its own**.

- **Support for More Systems** – Extend the tool to work with **cloud applications** and other platforms.

- **Integration with Debugging Tools** – Connect with **GDB, Visual Studio, and Eclipse Debuggers** for better debugging.

8. References

1. **Books:**
    - *Operating System Concepts* – Silberschatz, Galvin, Gagne.

2. **Research Papers:**
    - *Deadlock Detection Algorithms: A Comparative Study* – IEEE Transactions.
    - *Graph-Based Deadlock Detection Techniques* – ACM Computing Surveys.

3. **Online Documentation:**
    - Python **NetworkX Library** Documentation.
    - **Matplotlib** Graph Visualization Guide.
    - GitHub **Best Practices**.

9.Code and Program Sample:

```
import networkx as nx
import matplotlib.pyplot as plt

# ======================
# Data Processing Module
# ======================

def parse_input_data(logs):
    #Give input in form of list as P1->R1
    edges = []
    for log in logs:
        parts = log.split("->")
        if len(parts) == 2:
            edges.append((parts[0].strip(), parts[1].strip()))
    return edges


def build_resource_allocation_graph(edges):
    """
    Build a directed graph from the parsed edges.
    """
    graph = nx.DiGraph()
    graph.add_edges_from(edges)
    return graph
```

Figure 3: Data processing Module

```
# ======================
# Deadlock Detection Module
# ======================

def detect_deadlock(graph):

    #Detect deadlock by finding cycles in RAG.

    try:
        cycle = nx.find_cycle(graph, orientation="original")
        return True, cycle
    except nx.NetworkXNoCycle:
        return False, None


def suggest_resolution(cycle):
    # Giving Suggestions
    if cycle:
        resolution = f"Terminate one of the processes in the cycle: {cycle}"
        return resolution
    return "No deadlock detected."


def preempt_resource(graph, cycle):

    # Break the deadlock by preempting a reaource from one process in the cycle.
    if not cycle:
        return graph, "No deadlock to resolve."

    # Select the first edge in the cycle to preempt
    preempt_edge = cycle[0]
    process, resource = preempt_edge[0], preempt_edge[1]

    # Remove the edge from the graph
    graph.remove_edge(process, resource)

    return graph, f"Preempted resource {resource} from process {process}."
```

Figure 4: Deadlock Detection Module

```
# ======================
# Visualization Module
# ======================

def visualize_graph(graph, deadlock_cycle=None):

    #Visualize the RAG with deadlock
    pos = nx.spring_layout(graph)
    plt.figure(figsize=(8, 6))

    # Draw nodes and edges
    nx.draw_networkx_nodes(graph, pos, node_size=2000, node_color="lightblue")
    nx.draw_networkx_edges(graph, pos, edge_color="gray", arrowstyle="->", arrow
    nx.draw_networkx_labels(graph, pos, font_size=12, font_weight="bold")

    # Highlight deadlock cycle
    if deadlock_cycle:
        cycle_edges = [(u, v) for u, v, _ in deadlock_cycle]
        nx.draw_networkx_edges(graph, pos, edgelist=cycle_edges, edge_color="red

    plt.title("Resource Allocation Graph")
    plt.show()


# ======================
# Main Execution
# ======================

def get_user_input():
    #getting input from user

    logs = []
    print("Enter process-resource relationships (e.g., 'P1 -> R1'). Type 'done'
    while True:
        user_input = input("Enter relationship: ").strip()
        if user_input.lower() == "done":
            break
        logs.append(user_input)
    return logs
```

Ln: 1 Col:

Figure 5: Visualization Module

File Edit Shell Debug Options Window Help

```
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun  7 2023, 05:45:37
) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for mor
e information.
>>>
========== RESTART: C:\Users\rajsa\Desktop\Python\Main_Exe
cution_OS.py =========
Enter process-resource relationships (e.g., 'P1 -> R1'). T
ype 'done' to finish.
Enter relationship: p1->r1
Enter relationship: p2->r2
Enter relationship: r1->p2
Enter relationship: r2->p1
Enter relationship: p3->r2
Enter relationship: r3->p3
Enter relationship: done

Deadlock Detected!
Deadlock Cycle: [('p1', 'r1', 'forward'), ('r1', 'p2', 'fo
rward'), ('p2', 'r2', 'forward'), ('r2', 'p1', 'forward')]
Resolution Suggestion: Terminate one of the processes in t
he cycle: [('p1', 'r1', 'forward'), ('r1', 'p2', 'forward'
), ('p2', 'r2', 'forward'), ('r2', 'p1', 'forward')]
```

Figure 6: Taking Input for Allocated and Requested Edge

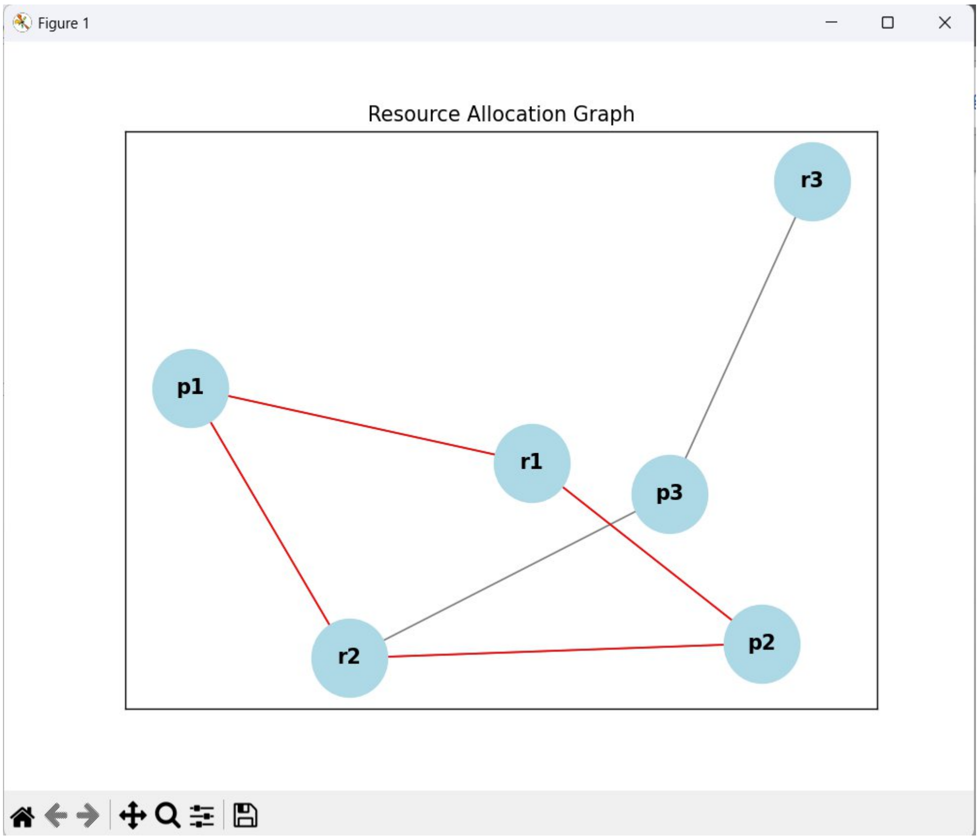


Figure 7: Resource Allocation Graph

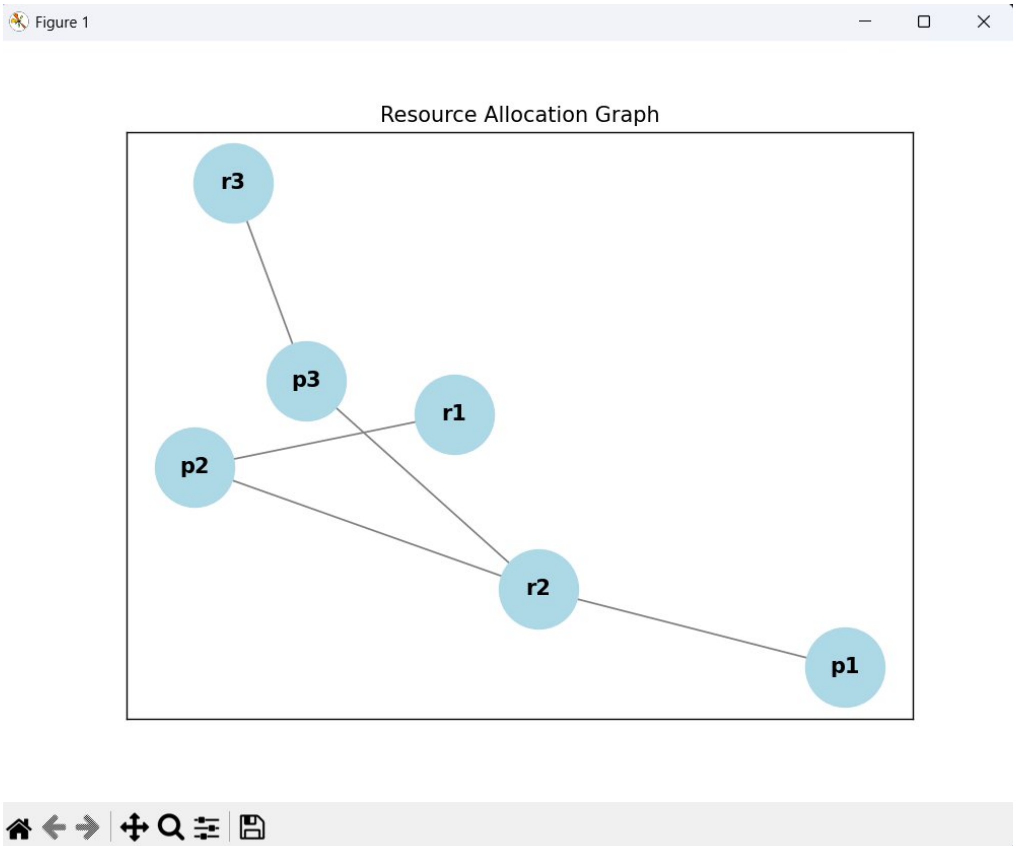


Figure 8: Resource Allocation Graph after Resource Pre-emption