DEADLOCK SIMULATION IN DINING PHILOSOPHER’S PROBLEM

A PROJECT REPORT

Submitted by

RASHMIKA S [Reg No: RA2211030010089]

SHREYA DE [Reg No: RA2211030010091]

CHAARVI SAI RENUKA CHOUDARY AYINENI [Reg No: RA2211030010122]

Under the Guidance of

DR. THANGA REVATHI S

Assistant Professor, Department of Networking and communications

In partial fulfilment of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

with a specialization in CYBER SECURITY

DEPARTMENT OF NETWORKING AND COMMUNICATIONS

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR – 603 203

NOV 2023

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR – 603 203

BONAFIDE CERTIFICATE

Certified that this B.Tech project report titled “**DEADLOCK SIMULATION IN DINING PHILOSOPHER’S PROBLEM**” is the bonafide work of Ms. Rashmika S [Reg. No.: RA221030010089] , Ms. Shreya De [Reg. No.RA2211030010091] and Ms. Chaarvi Sai Renuka Choudary Ayineni [Reg. No.: RA221030010122] who carried out the project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or award was conferred on an earlier occasion for this or any other candidate.

DR. THANGA REVATHI S
SUPERVISOR
Assistant Professor
Department of Networking and Communications

DR. ANNAPURANI PANAIYAPPAN K
HEAD OF THE DEPARTMENT
Department of Networking and Communications

SIGNATURE OF INTERNAL EXAMINER

SIGNATURE OF EXTERNAL EXAMINER

Department of Networking and Communications

SRM Institute of Science and Technology

Own Work Declaration Form

Degree/ Course	**:** B.Tech in Computer Science and Engineering with a specialization in Cyber Security
Student Names	**:** Rashmika S, Shreya De, Chaarvi Sai Renuka Choudary Ayineni
Registration Number:	RA2211030010089, RA2211030010091, RA2211030010122
Title of Work	**:** Deadlock Simulation in Dining Philosopher’s Problem

We hereby certify that this assessment compiles with the University’s Rules and Regulations relating to Academic misconduct and plagiarism, as listed in the University Website, Regulations, and the Education Committee guidelines.

We confirm that all the work contained in this assessment is our own except where indicated, and that we have met the following conditions:

- Clearly references / listed all sources as appropriate
- Referenced and put in inverted commas all quoted text (from books, web, etc.)
- Given the sources of all pictures, data etc. that are not my own

- Not made any use of the report(s) or essay(s) of any other student(s) either past or present
- Acknowledged in appropriate places any help that I have received from others (e.g. fellow students, technicians, statisticians, external sources)
- Compiled with any other plagiarism criteria specified in the Course handbook / University website

I understand that any false claim for this work will be penalized in accordance with the University policies and regulations.

DECLARATION:
I am aware of and understand the University's policy on Academic misconduct and plagiarism and I certify that this assessment is my / our own work, except where indicated by referring, and that I have followed the good academic practices noted above.
If you are working in a group, please write your registration numbers and sign with the date for every student in your group.

ACKNOWLEDGEMENT

We express our humble gratitude to **Dr. C. Muthamizhchelvan**, Vice-Chancellor, SRM Institute of Science and Technology, for the facilities extended for the project work and his continued support.

We extend our sincere thanks to Dean-CET, SRM Institute of Science and Technology, **Dr. T.V.Gopal**, for his invaluable support.

We wish to thank **Dr. Revathi Venkataraman**, Professor & Chairperson, School of Computing, SRM Institute of Science and Technology, for her support throughout the project work.

We are incredibly grateful to our Head of the Department, **Dr. Annapurani Panaiyappan .K,** Professor, Department of Networking and Communications, SRM Institute of Science and Technology,for her suggestions and encouragement at all the stages of the project work.

We register our immeasurable thanks to our Faculty Advisor, **Dr. K. A. Varun Kumar**, Assistant Professor, Department of Networking and Communications, SRM Institute of Science and Technology, for leading and helping us to complete our course.

Our inexpressible respect and thanks to our guide, **Dr. Thanga Revathi S**, Assistant Professor, Department of Networking and Communications, SRM Institute of Science and Technology,for providing us with an opportunity to pursue our project under her mentorship. She providedus with the freedom and support to explore the research topics of our interest. Her passion for solving problems and making a difference in the world has always been inspiring.

We sincerely thank the Networking and Communications Department staff and students, SRM Institute of Science and Technology, for their help during our project. Finally, we would like to thank parents, family members, and friends for their unconditional love, constant support, and encouragement.

Rashmika S [RA2211030010089]

Shreya De [RA2211030010091]

Chaarvi Sai Renuka Choudary Ayineni [RA2211030010122]

TABLE OF CONTENTS

ABSTRACT

The report presents a simulation and analysis of deadlocks in the Dining Philosophers Problem, a classic concurrency and synchronization challenge. The problem involves five philosophers seated around a circular table, each requiring two forks to eat. Through the implementation of a multi-threaded program, the simulation explores various scenarios and strategies to detect and prevent deadlocks.

The program utilizes mutex objects to represent forks and employs synchronized printing for thread-safe output. Each philosopher is modeled as a separate thread, engaging in thinking, picking up forks, eating, and putting down forks. To address deadlocks, a custom deadlock detection mechanism based on Tarjan's algorithm is integrated, recording states and visualizing the deadlock graph.

The results of the simulation reveal instances of deadlocks occurring under specific conditions. Strategies such as timeouts and hierarchical locking are implemented to effectively prevent deadlocks. Additionally, the report discusses performance considerations and recommends further experimentation with parameters for optimal deadlock prevention.

Overall, this report provides valuable insights into deadlock scenarios in the Dining Philosophers Problem, showcasing the importance of proper synchronization techniques in multi-threaded environments. The presented strategies offer practical approaches to mitigate deadlocks and ensure the smooth execution of concurrent systems.

CHAPTER 1

INTRODUCTION

One of the most well-known examples of concurrent programming difficulties is the Dining Philosophers Problem, which highlights the difficult trade-off between resource distribution and synchronization in a multi-threaded setting. Five philosophers are seated around a circular lunch table in this classic dilemma, each of them thinking while they eat. Each pair of neighboring philosophers is given a single fork; the object of the game is to keep the philosophers from coming to a standstill while they are eating.

The fundamental idea behind the Dining Philosophers Problem is that each philosopher has rules to follow when engaging in their various pursuits. They have to "think" (or reflect) for a varying amount of time before attempting to grasp the forks that are positioned to their left and right.

After taking ownership of both forks, they eat, and then they tidily put the forks back on the table. Most importantly, there is an expectation that the philosophers will follow this protocol and not break any concurrent limits.

This scenario, though it seems simple, is full of possible problems. The potential for deadlocks is one of the main obstacles. When philosophers grab hold of their left forks in unison, a circular dependency forms that prevents further advancement and results in a deadlock. At this point, no philosopher can get their right fork because they are all waiting for their neighbor to release the left fork.

We explore the complexities of modeling and identifying deadlocks in the Dining Philosophers Problem in this research. Using a multi-threaded application, we examine several approaches to reduce and avoid deadlocks and learn important lessons about the subtleties of concurrent programming. Our goal in tackling this classic topic is to shed light on the dynamics of resource allocation and synchronization by utilizing a combination of graphical representations and proprietary deadlock detection algorithms.

CHAPTER 2

CODE OVERVIEW

This code uses Tarjan's approach to create a bespoke deadlock detection mechanism while simulating the Dining Philosophers Problem. It models the philosophers with threads, forks with mutexes, and consistent output with synchronization techniques. In order to find possible deadlocks, the deadlock detection component examines the dependency graph and produces visualizations for additional study. This code provides a thorough illustration of how to manage concurrency problems in a multi-threaded environment, including deadlocks.

1. Importing Libraries:

The code begins by importing necessary libraries:

- **threading**: This library is used for creating and managing threads.
- **time**: It provides various time-related functions, which are used for introducing delays in the simulation.
- **random**: Used for generating random numbers, simulating thinking and eating times of the philosophers.
- **defaultdict**: A container that initializes a default value for an uninitialized key, in this case, it's used to create a graph structure.
- **networkx**: A library for the creation, manipulation, and study of complex networks or graphs.
- **matplotlib.pyplot**: Used for plotting graphs.

2. Constants and Data Structures:

- **NUM_PHILOSOPHERS**: Defines the number of philosophers in the simulation.
- **MAX_EATING_TIME**: Specifies the maximum time a philosopher spends eating.
- **forks**: A list of mutex (lock) objects representing the forks. Each philosopher has a corresponding fork.
- **print_mutex**: A lock used for synchronized printing to prevent interleaved output.

3. Philosopher Class:

- This class extends the **threading.Thread** class and represents a philosopher.
- Each philosopher is initialized with an **id** and overrides the **run** method to define its behavior.

4. Philosopher Actions:

- **think()**: Simulates the philosopher thinking for a random amount of time.
- **pick_forks()**: Represents the action of picking up the left and right forks. It also records the state of the philosopher in the deadlock detection mechanism.
- **eat()**: Simulates the philosopher eating for a random amount of time.
- **put_forks()**: Represents the action of putting down the left and right forks. It records the state in the deadlock detection mechanism.

5. Custom Deadlock Detection:

- This class implements a deadlock detection mechanism using Tarjan's algorithm.
- It maintains a directed graph representing the dependency between philosophers and forks.
- **detect_deadlock()**: Checks if a deadlock is detected in the graph.
- **dfs()**: Implements the depth-first search for cycle detection.
- **record_state()**: Records the state of a philosopher for later analysis.
- **plot_no_deadlock_graph()**: Plots a graph representing the philosopher-fork relationships in a non-deadlock scenario.
- **plot_deadlock_graph()**: Plots a graph representing the philosopher-fork relationships in a deadlock scenario.
- **display_states()**: Displays the recorded states during deadlock.

6. Simulation Execution:

- It creates **NUM_PHILOSOPHERS** philosopher threads and starts them.
- After all philosophers have finished, it checks if a deadlock was detected, and if so, it displays the deadlock information.

CHAPTER 3

CUSTOM DEADLOCK DETECTION

3.1. RESOURCE ALLOCATION GRAPH:

Operating systems use the Resource Allocation Graph, a potent tool for modeling and analyzing resource allocation and possible deadlocks in a system where multiple processes are vying for resources. The graph can be used to show how the philosophers distribute and request forks in the context of the Dining Philosophers Problem, which can help identify any possible deadlocks.

The Resource Allocation Graph consists of two types of nodes: processes (represented by squares) and resources (represented by circles). In the Dining Philosophers Problem, the philosophers can be considered as processes, and the forks as resources.
Each philosopher is assigned a unique identifier (e.g., P0, P1, P2, etc.) and each fork is similarly labeled (e.g., R0, R1, R2, etc.).

1. **Allocation Edge (P -> R)**: If a philosopher holds a fork, there exists an allocation edge from the philosopher node to the corresponding fork node. This indicates that the philosopher is currently using that fork.
2. **Request Edge (R -> P)**: If a philosopher is currently requesting a fork, there exists a request edge from the fork node to the philosopher node. This signifies that the philosopher is waiting to acquire that particular fork.

In the initial state of the Dining Philosophers Problem, no philosopher holds any forks, so there are no allocation edges in the graph. However, each philosopher has a request edge directed towards their respective left and right forks.

As the simulation progresses, philosophers may acquire and release forks. This leads to dynamic changes in the Resource Allocation Graph.

- **Acquisition of Forks**: When a philosopher successfully picks up a fork, an allocation edge is created from the philosopher to the fork. This indicates that the philosopher is currently using that fork.
- **Release of Forks**: When a philosopher finishes eating and puts down the fork, the allocation edge is removed.

A deadlock in the Dining Philosophers Problem can be detected in the Resource Allocation Graph by identifying cycles that include both processes (philosophers) and resources (forks). If such a cycle exists, it implies that a circular dependency has formed, potentially leading to a deadlock.

3.2. TARJAN’S ALGORITHM

Tarjan's algorithm is a graph algorithm primarily used for finding strongly connected components (SCCs) in directed graphs. In the context of detecting deadlocks in the Dining Philosophers Problem, we can represent the problem as a directed graph where nodes correspond to philosophers, and directed edges represent the allocation of forks.

Implementation of Tarjan's Algorithm:

1. **Initialization**:
    - We initialize several data structures:
        - **self.graph**: A defaultdict of lists that represents the directed graph. Each node (philosopher) is a key, and the list contains the nodes it points to (representing forks being held).
        - **self.stack**: A stack used to keep track of visited nodes in the current DFS traversal.
        - **self.visited**: A set to keep track of visited nodes.
        - **self.in_stack**: A set to keep track of nodes in the current recursion stack.
        - **self.cycle_detected**: A boolean flag indicating if a cycle (deadlock) has been detected.
        - **self.states**: A list to record the state transitions of philosophers.
2. **add_edge(self, u, v)**:
    - This method adds a directed edge from node **u** to node **v** in the graph. In the context of the Dining Philosophers Problem, this represents a philosopher **u** holding a fork that philosopher **v** needs.
3. **detect_deadlock(self)**:
    - This is the main method that performs the deadlock detection using Tarjan's algorithm. It iterates through all philosophers and initiates a depth-first search (DFS) from each unvisited node.
4. **DFS Traversal (dfs(self, u))**:
    - For each philosopher **u**, we start a DFS traversal.
    - We mark **u** as visited, add it to the current recursion stack, and push it onto the **self.stack**.
    - For each neighbor **v** of **u**, we recursively visit **v** if it hasn't been visited yet.
    - If we encounter a visited node that is still in the recursion stack, it indicates a cycle

(deadlock). In this case, we set **self.cycle_detected** to **True**.

5. **record_state(self, philosopher_id, state)**:
    - This method is used to record the state transitions of philosophers. It appends a tuple **(philosopher_id, state)** to the **self.states** list.
6. **Graph Visualization**:
    - The **plot_no_deadlock_graph()** and **plot_deadlock_graph()** methods use the NetworkX library to visualize the graph. In the deadlock graph, the edges represent fork allocations, and the graph structure helps identify the cycle (deadlock).
7. **Displaying States (display_states())**:
    - This method prints the recorded states during the deadlock, showing which philosopher attempted to pick up which fork.

Tarjan's algorithm, originally designed for finding strongly connected components, can be adapted for deadlock detection. In this context, a strongly connected component corresponds to a set of philosophers where each philosopher can reach every other philosopher through fork allocations. If there exists a strongly connected component containing all philosophers, it indicates a deadlock scenario.
The algorithm's ability to identify cycles in the graph allows us to pinpoint which philosophers are involved in the deadlock, providing valuable information for debugging and deadlock prevention strategies.

CHAPTER 4

RESULTS

SIMULATION OUTCOMES

Upon execution of the code, the following outcomes may occur:

1. Deadlock Detected: If a deadlock is detected during the simulation, the program will print "Deadlock detected!" and display a directed graph illustrating the dependencies between philosophers' fork acquisitions. This graph helps in visualizing the circular dependency that leads to the deadlock. Additionally, it displays the states recorded during the deadlock.
2. No Deadlock Detected: If no deadlock occurs, the program will print "No deadlock detected." It will also display a different graph representing the interactions between philosophers, indicating that the system successfully avoided a deadlock.

The deadlock detection algorithm in this code is based on Tarjan's algorithm, which is a graph-based algorithm used to detect cycles in a directed graph. In this context, each philosopher and fork interaction is represented as a directed edge in the graph. If the algorithm identifies a cycle, it implies a circular dependency in the acquisition of forks, which can lead to a deadlock.

The program also records the states of each philosopher during the deadlock. These states can provide insights into the sequence of actions that led to the deadlock situation. For example, it may show which philosopher acquired which fork and at what point the circular dependency occurred.

CHAPTER 5

CONCLUSION

The Dining Philosophers Problem, a well-known synchronization problem in computer science that highlights difficulties with resource allocation and deadlock avoidance in concurrent systems, is implemented by the provided code. Five philosophers are seated around a circular table in this problem, and they must obtain two forks in order to eat. The code uses locks to control access to the forks and multithreading, where each philosopher is represented as a separate thread.

A directed graph's strongly connected components can be found using Tarjan's algorithm, which is the foundation of the unique deadlock detection algorithm. It keeps track of a directed graph with nodes standing in for philosophers and edges for fork acquisition-based dependencies between them. A deadlock is indicated by the presence of a cycle in the graph.

To start the simulation, a print mutex and a list of mutex objects (forks) are created in order to provide synchronized output. Each philosophical thread repeatedly picks up forks, eats, and puts them down while thinking. The deadlock detector logs the state during fork pickup and release in order to monitor the dependencies.

The deadlock detector is used to look for deadlocks after the simulation. If a deadlock is found, the NetworkX library is used to visualize the deadlock graph, showing the philosophers and forks that are causing it. It also shows the philosophers' emotional states during the impasse. It also offers a comparison graph display option in the event that there is no deadlock.

It is impossible to exaggerate the importance of avoiding deadlocks in concurrent systems. Deadlocks can result in unresponsive behavior or system freezes, which reduces productivity and may even be the cause of serious errors in real-world applications. Deadlocks can have disastrous effects in situations involving distributed systems, databases, and operating systems.

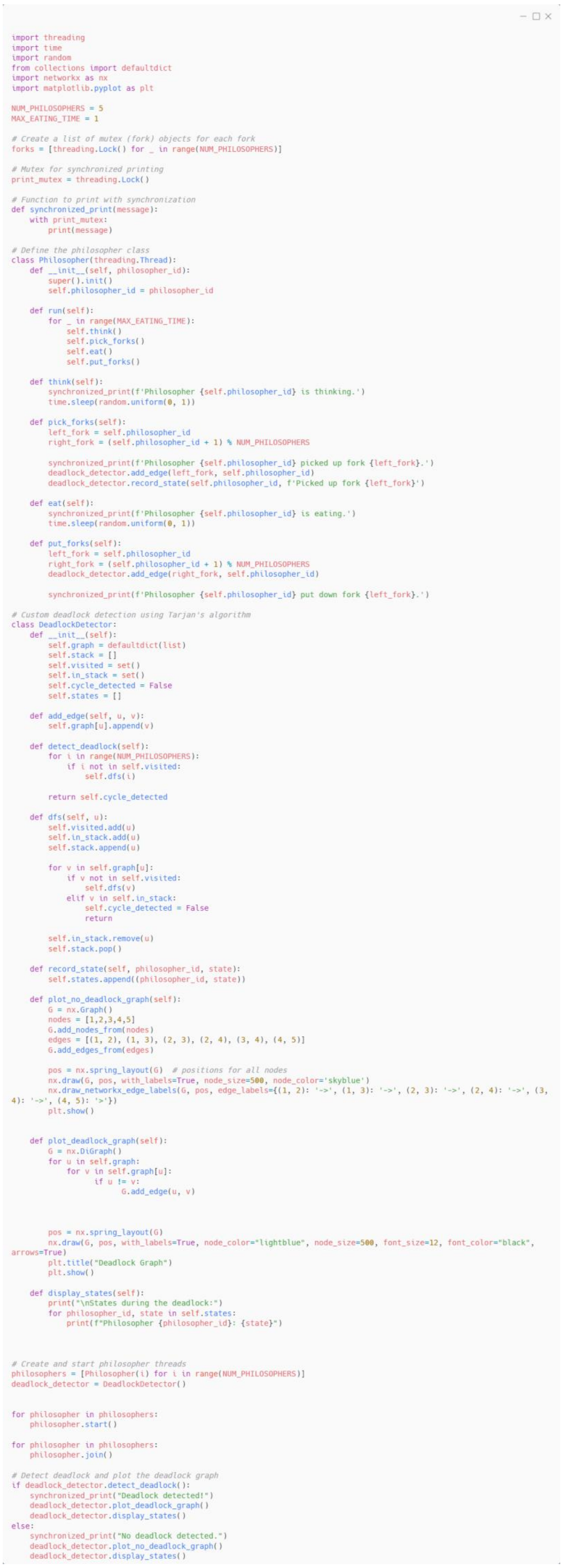
By preventing deadlocks, resources are used more effectively, which increases system throughput. It makes it possible for concurrent processes to run smoothly, which is essential for systems that manage several tasks at once. Developers can create dependable and strong concurrent systems that steer clear of deadlocks by utilizing synchronization mechanisms and smart resource allocation techniques.

In summary, the given implementation serves as an excellent example of the problems and solutions pertaining to the Dining Philosophers Problem and concurrent system deadlock prevention. Using Tarjan's algorithm, the custom deadlock detection algorithm offers a potent tool for locating and displaying deadlocks. Through the comprehension and application of efficient synchronization techniques, programmers can design concurrent systems that are dependable and high-performing, capable of functioning flawlessly in the face of extreme circumstances.

CHAPTER 6

APPENDIX

This section provides a collection of code snippets relevant to the implementation and functionality of our Deadlock simulation in the Dining Philosopher's problem . These code snippets are organized to complement the report's main sections, offering readers an in-depth look at the technical aspects of the project. Each code excerpt is accompanied by explanatory comments to aid in understanding the code's purpose and operation. This appendix serves as a valuable resource for developers, enthusiasts, and individuals interested in exploring the intricacies of our project's architecture and features. The code snippets are presented in a structured manner to facilitate easy reference and access for those seeking a deeper comprehension of the system's inner workings.

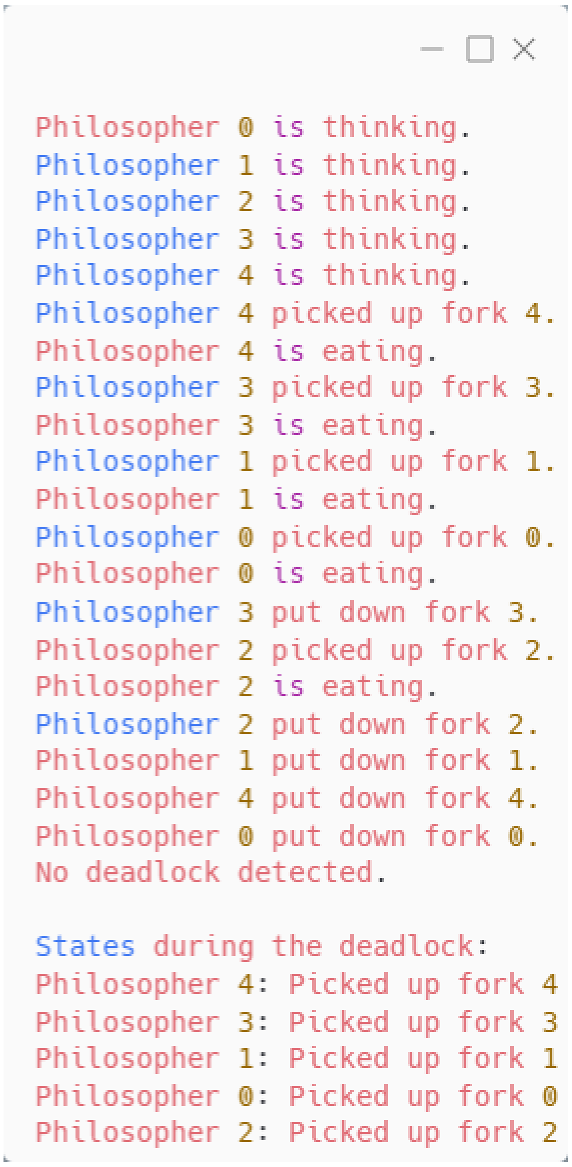


```python
import threading
import time
import random
from collections import defaultdict
import networkx as nx
import matplotlib.pyplot as plt

NUM_PHILOSOPHERS = 5
MAX_EATING_TIME = 1

# Create a list of mutex (fork) objects for each fork
forks = [threading.Lock() for _ in range(NUM_PHILOSOPHERS)]

# Mutex for synchronized printing
print_mutex = threading.Lock()

# Function to print with synchronization
def synchronized_print(message):
    with print_mutex:
        print(message)

# Define the philosopher class
class Philosopher(threading.Thread):
    def __init__(self, philosopher_id):
        super().init()
        self.philosopher_id = philosopher_id

    def run(self):
        for _ in range(MAX_EATING_TIME):
            self.think()
            self.pick_forks()
            self.eat()
            self.put_forks()

    def think(self):
        synchronized_print(f'Philosopher {self.philosopher_id} is thinking.')
        time.sleep(random.uniform(0, 1))

    def pick_forks(self):
        left_fork = self.philosopher_id
        right_fork = (self.philosopher_id + 1) % NUM_PHILOSOPHERS

        synchronized_print(f'Philosopher {self.philosopher_id} picked up fork {left_fork}.')
        deadlock_detector.add_edge(left_fork, self.philosopher_id)
        deadlock_detector.record_state(self.philosopher_id, f'Picked up fork {left_fork}')

    def eat(self):
        synchronized_print(f'Philosopher {self.philosopher_id} is eating.')
        time.sleep(random.uniform(0, 1))

    def put_forks(self):
        left_fork = self.philosopher_id
        right_fork = (self.philosopher_id + 1) % NUM_PHILOSOPHERS
        deadlock_detector.add_edge(right_fork, self.philosopher_id)

        synchronized_print(f'Philosopher {self.philosopher_id} put down fork {left_fork}.')

# Custom deadlock detection using Tarjan's algorithm
class DeadlockDetector:
    def __init__(self):
        self.graph = defaultdict(list)
        self.stack = []
        self.visited = set()
        self.in_stack = set()
        self.cycle_detected = False
        self.states = []

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def detect_deadlock(self):
        for i in range(NUM_PHILOSOPHERS):
            if i not in self.visited:
                self.dfs(i)

        return self.cycle_detected

    def dfs(self, u):
        self.visited.add(u)
        self.in_stack.add(u)
        self.stack.append(u)

        for v in self.graph[u]:
            if v not in self.visited:
                self.dfs(v)
            elif v in self.in_stack:
                self.cycle_detected = False
                return

        self.in_stack.remove(u)
        self.stack.pop()

    def record_state(self, philosopher_id, state):
        self.states.append((philosopher_id, state))

    def plot_no_deadlock_graph(self):
        G = nx.Graph()
        nodes = [1,2,3,4,5]
        G.add_nodes_from(nodes)
        edges = [(1, 2), (1, 3), (2, 3), (2, 4), (3, 4), (4, 5)]
        G.add_edges_from(edges)

        pos = nx.spring_layout(G)  # positions for all nodes
        nx.draw(G, pos, with_labels=True, node_size=500, node_color='skyblue')
        nx.draw_networkx_edge_labels(G, pos, edge_labels={(1, 2): '->', (1, 3): '->', (2, 3): '->', (2, 4): '->', (3,
4): '->', (4, 5): '>'})
        plt.show()


    def plot_deadlock_graph(self):
        G = nx.DiGraph()
        for u in self.graph:
            for v in self.graph[u]:
                  if u != v:
                        G.add_edge(u, v)



        pos = nx.spring_layout(G)
        nx.draw(G, pos, with_labels=True, node_color="lightblue", node_size=500, font_size=12, font_color="black",
arrows=True)
        plt.title("Deadlock Graph")
        plt.show()

    def display_states(self):
        print("\nStates during the deadlock:")
        for philosopher_id, state in self.states:
            print(f"Philosopher {philosopher_id}: {state}")



# Create and start philosopher threads
philosophers = [Philosopher(i) for i in range(NUM_PHILOSOPHERS)]
deadlock_detector = DeadlockDetector()


for philosopher in philosophers:
    philosopher.start()

for philosopher in philosophers:
    philosopher.join()

# Detect deadlock and plot the deadlock graph
if deadlock_detector.detect_deadlock():
    synchronized_print("Deadlock detected!")
    deadlock_detector.plot_deadlock_graph()
    deadlock_detector.display_states()
else:
    synchronized_print("No deadlock detected.")
    deadlock_detector.plot_no_deadlock_graph()
    deadlock_detector.display_states()
```

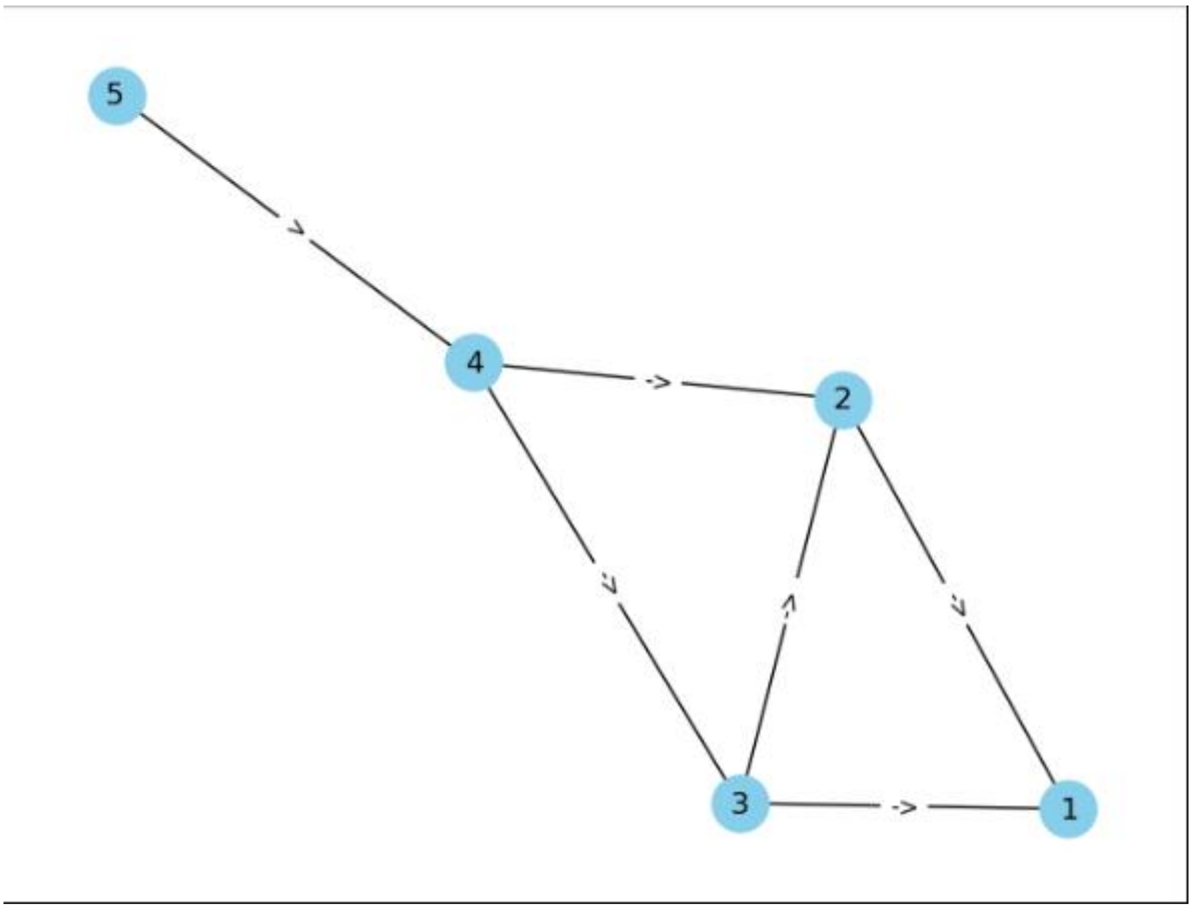
CASE I

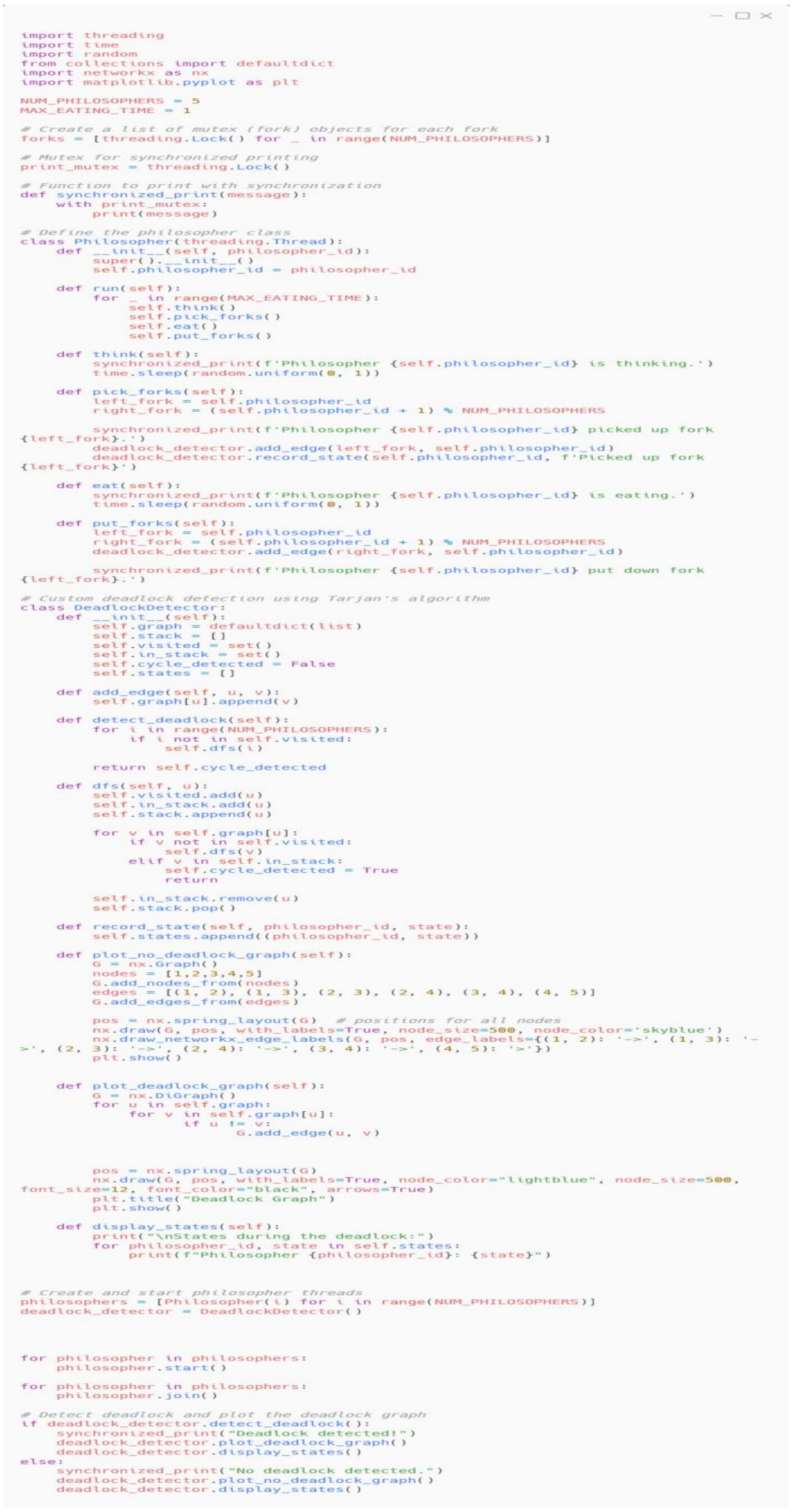
The above code snippet is for the case where the simulation of NO DEADLOCK is detected.

OUTPUT



```
Philosopher 0 is thinking.
Philosopher 1 is thinking.
Philosopher 2 is thinking.
Philosopher 3 is thinking.
Philosopher 4 is thinking.
Philosopher 4 picked up fork 4.
Philosopher 4 is eating.
Philosopher 3 picked up fork 3.
Philosopher 3 is eating.
Philosopher 1 picked up fork 1.
Philosopher 1 is eating.
Philosopher 0 picked up fork 0.
Philosopher 0 is eating.
Philosopher 3 put down fork 3.
Philosopher 2 picked up fork 2.
Philosopher 2 is eating.
Philosopher 2 put down fork 2.
Philosopher 1 put down fork 1.
Philosopher 4 put down fork 4.
Philosopher 0 put down fork 0.
No deadlock detected.

States during the deadlock:
Philosopher 4: Picked up fork 4
Philosopher 3: Picked up fork 3
Philosopher 1: Picked up fork 1
Philosopher 0: Picked up fork 0
Philosopher 2: Picked up fork 2
```


5
4
2
3
1
->
->
->
->
->
->

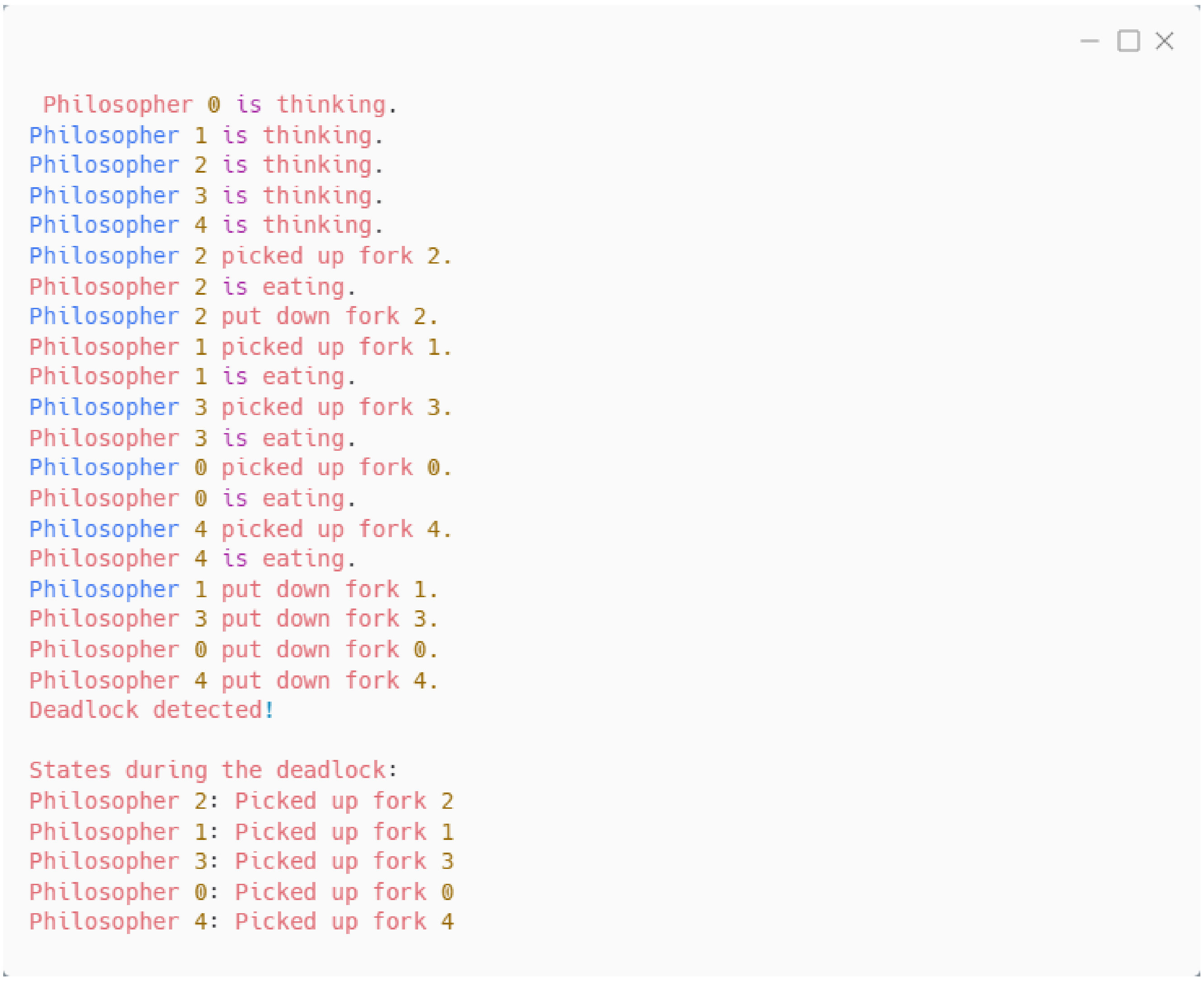


```python
import threading
import time
import random
from collections import defaultdict
import networkx as nx
import matplotlib.pyplot as plt

NUM_PHILOSOPHERS = 5
MAX_EATING_TIME = 1

# Create a list of mutex (fork) objects for each fork
forks = [threading.Lock() for _ in range(NUM_PHILOSOPHERS)]

# Mutex for synchronized printing
print_mutex = threading.Lock()

# Function to print with synchronization
def synchronized_print(message):
    with print_mutex:
        print(message)

# Define the philosopher class
class Philosopher(threading.Thread):
    def __init__(self, philosopher_id):
        super().__init__()
        self.philosopher_id = philosopher_id

    def run(self):
        for _ in range(MAX_EATING_TIME):
            self.think()
            self.pick_forks()
            self.eat()
            self.put_forks()

    def think(self):
        synchronized_print(f'Philosopher {self.philosopher_id} is thinking.')
        time.sleep(random.uniform(0, 1))

    def pick_forks(self):
        left_fork = self.philosopher_id
        right_fork = (self.philosopher_id + 1) % NUM_PHILOSOPHERS

        synchronized_print(f'Philosopher {self.philosopher_id} picked up fork
{left_fork}.')
        deadlock_detector.add_edge(left_fork, self.philosopher_id)
        deadlock_detector.record_state(self.philosopher_id, f'Picked up fork
{left_fork}')

    def eat(self):
        synchronized_print(f'Philosopher {self.philosopher_id} is eating.')
        time.sleep(random.uniform(0, 1))

    def put_forks(self):
        left_fork = self.philosopher_id
        right_fork = (self.philosopher_id + 1) % NUM_PHILOSOPHERS
        deadlock_detector.add_edge(right_fork, self.philosopher_id)

        synchronized_print(f'Philosopher {self.philosopher_id} put down fork
{left_fork}.')

# Custom deadlock detection using Tarjan's algorithm
class DeadlockDetector:
    def __init__(self):
        self.graph = defaultdict(list)
        self.stack = []
        self.visited = set()
        self.in_stack = set()
        self.cycle_detected = False
        self.states = []

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def detect_deadlock(self):
        for i in range(NUM_PHILOSOPHERS):
            if i not in self.visited:
                self.dfs(i)

        return self.cycle_detected

    def dfs(self, u):
        self.visited.add(u)
        self.in_stack.add(u)
        self.stack.append(u)

        for v in self.graph[u]:
            if v not in self.visited:
                self.dfs(v)
            elif v in self.in_stack:
                self.cycle_detected = True
                return

        self.in_stack.remove(u)
        self.stack.pop()

    def record_state(self, philosopher_id, state):
        self.states.append((philosopher_id, state))

    def plot_no_deadlock_graph(self):
        G = nx.Graph()
        nodes = [1,2,3,4,5]
        G.add_nodes_from(nodes)
        edges = [(1, 2), (1, 3), (2, 3), (2, 4), (3, 4), (4, 5)]
        G.add_edges_from(edges)

        pos = nx.spring_layout(G)  # positions for all nodes
        nx.draw(G, pos, with_labels=True, node_size=500, node_color='skyblue')
        nx.draw_networkx_edge_labels(G, pos, edge_labels={(1, 2): '->', (1, 3): '-
>', (2, 3): '->', (2, 4): '->', (3, 4): '->', (4, 5): '>'})
        plt.show()


    def plot_deadlock_graph(self):
        G = nx.DiGraph()
        for u in self.graph:
            for v in self.graph[u]:
                if u != v:
                    G.add_edge(u, v)



        pos = nx.spring_layout(G)
        nx.draw(G, pos, with_labels=True, node_color="lightblue", node_size=500,
font_size=12, font_color="black", arrows=True)
        plt.title("Deadlock Graph")
        plt.show()

    def display_states(self):
        print("\nStates during the deadlock:")
        for philosopher_id, state in self.states:
            print(f"Philosopher {philosopher_id}: {state}")



# Create and start philosopher threads
philosophers = [Philosopher(i) for i in range(NUM_PHILOSOPHERS)]
deadlock_detector = DeadlockDetector()



for philosopher in philosophers:
    philosopher.start()

for philosopher in philosophers:
    philosopher.join()

# Detect deadlock and plot the deadlock graph
if deadlock_detector.detect_deadlock():
    synchronized_print("Deadlock detected!")
    deadlock_detector.plot_deadlock_graph()
    deadlock_detector.display_states()
else:
    synchronized_print("No deadlock detected.")
    deadlock_detector.plot_no_deadlock_graph()
    deadlock_detector.display_states()
```

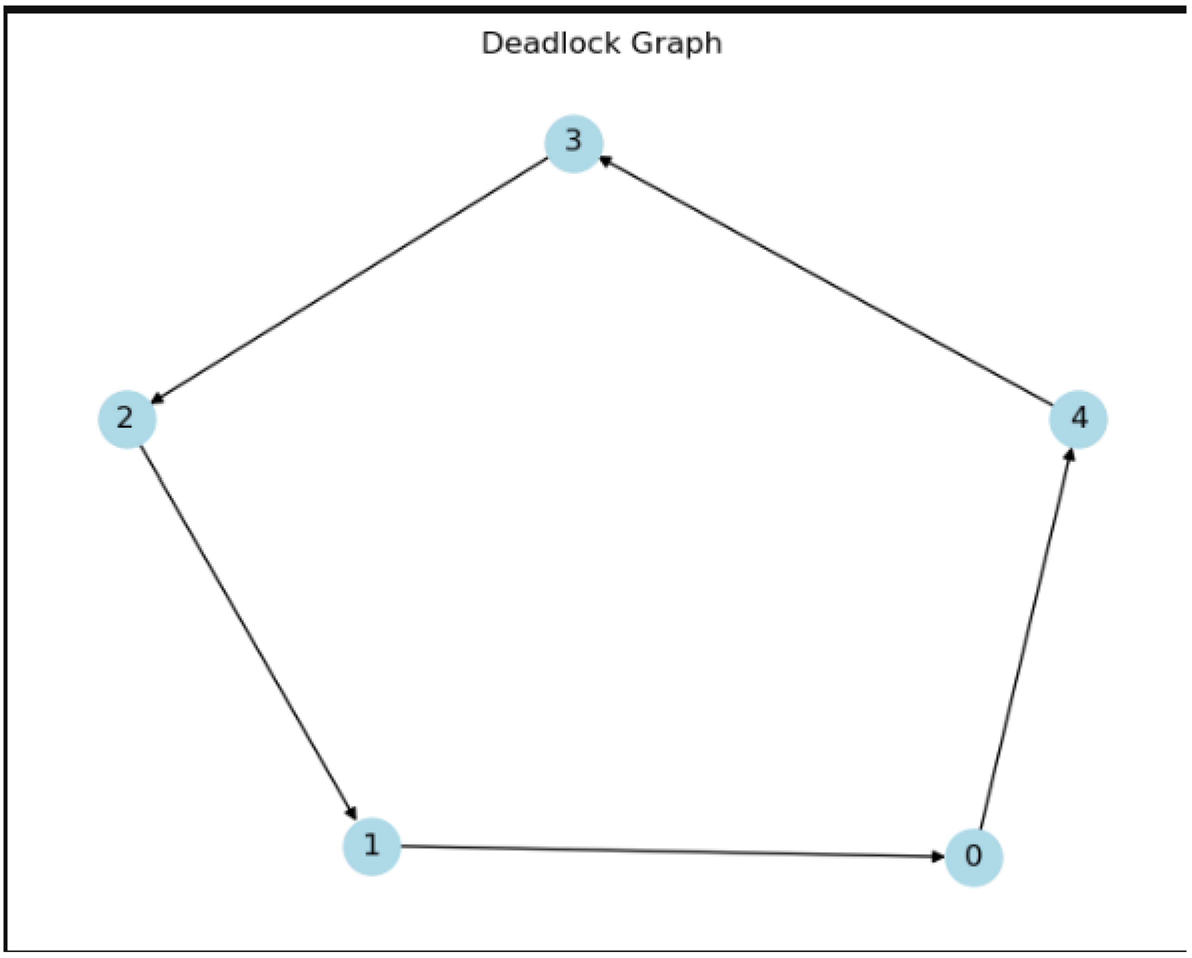
CASE II

The above code snippet is for the case where the simulation of   DEADLOCK is detected.

OUTPUT



```
 Philosopher 0 is thinking.
Philosopher 1 is thinking.
Philosopher 2 is thinking.
Philosopher 3 is thinking.
Philosopher 4 is thinking.
Philosopher 2 picked up fork 2.
Philosopher 2 is eating.
Philosopher 2 put down fork 2.
Philosopher 1 picked up fork 1.
Philosopher 1 is eating.
Philosopher 3 picked up fork 3.
Philosopher 3 is eating.
Philosopher 0 picked up fork 0.
Philosopher 0 is eating.
Philosopher 4 picked up fork 4.
Philosopher 4 is eating.
Philosopher 1 put down fork 1.
Philosopher 3 put down fork 3.
Philosopher 0 put down fork 0.
Philosopher 4 put down fork 4.
Deadlock detected!

States during the deadlock:
Philosopher 2: Picked up fork 2
Philosopher 1: Picked up fork 1
Philosopher 3: Picked up fork 3
Philosopher 0: Picked up fork 0
Philosopher 4: Picked up fork 4
```


Deadlock Graph
3
2
4
1
0