CL 3 Practical 1

Title: Design a distributed application using RPC for remote computation where the client submits an integer value to the server, and the server calculates the factorial and returns the result to the client program.

Software and Hardware Requirements:

- Programming language: Python
- Operating System: Ubuntu

Theory:

Remote Procedure Call (RPC) is a communication protocol that enables a program to request a service from another program located on a different computer in a network. RPC abstracts the complexities of network communication, allowing for seamless interaction between client and server applications in a distributed environment.

In this scenario, the client program initiates an RPC request to the server program, passing an integer value as the input. The server, upon receiving the request, calculates the factorial of the integer and returns the result to the client. This process demonstrates the use of RPC for remote computation, where the client and server communicate over the network without the need for the client to understand the internal workings of the server.

Algorithm:

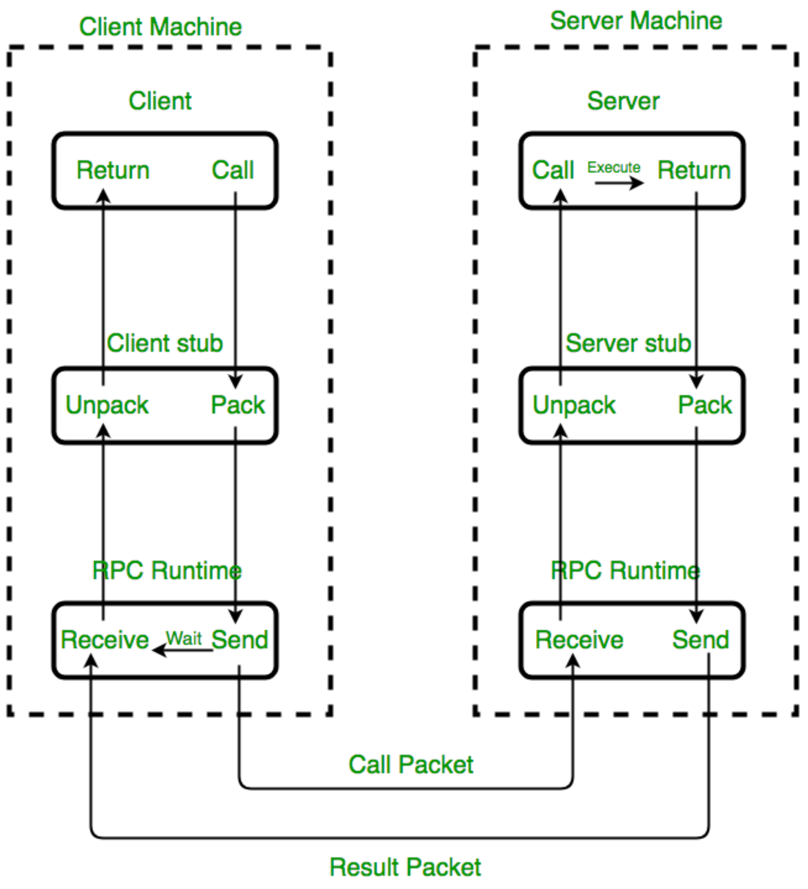
1. **Client Request**: The client program sends an RPC request to the server, passing an integer value as the input for which the factorial needs to be calculated.
2. **Server Receives Request**: The server receives the RPC request from the client, along with the integer value.
3. **Factorial Calculation**: The server calculates the factorial of the integer value using a factorial function. This function iteratively multiplies the integer by all positive integers less than itself down to 1.
4. **Result Generation**: Once the factorial is calculated, the server generates the result, which is the factorial of the input integer.

5. **Sending Result to Client**: The server sends the result (factorial value) back to the client as a response to the RPC request.

6. **Client Receives Result**: The client receives the result from the server and processes it, typically displaying it to the user.

7. **End of RPC**: The RPC process is completed, and the client and server return to their idle states, ready to handle new requests.

Flowchart:



Implementation of RPC mechanism

Conclusion/Analysis: The design and implementation of a distributed application using RPC for remote computation have been successful. RPC provides a convenient way to perform remote procedure calls, enabling the client to interact with the server seamlessly.

CL 3 Practical 2

Title: Design a distributed application using RMI for remote computation where the client submits two strings to the server, and the server returns the concatenation of the given strings.

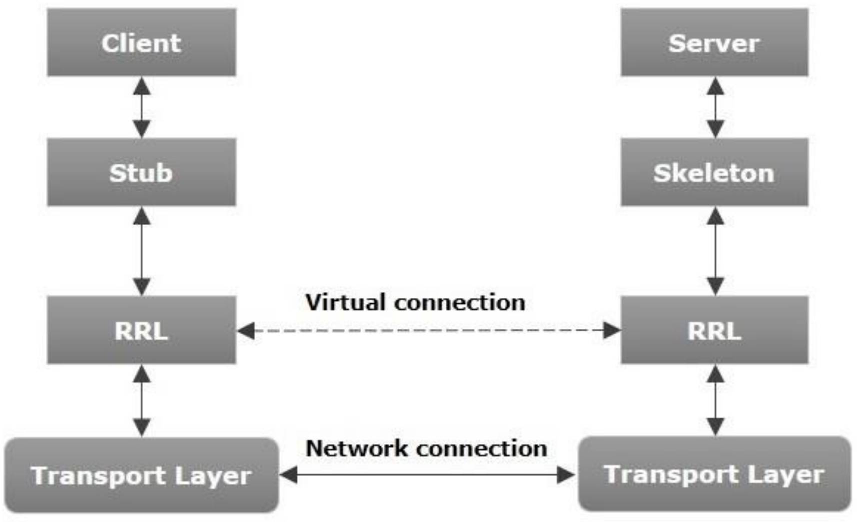
Software Requirements:

- Java Development Kit (JDK) for compiling and running Java code.
- IDE (e.g., IntelliJ IDEA, Eclipse) for writing and debugging code.
- Network connection between client and server machines.

Theory:

Remote Method Invocation (RMI) is a Java API that enables communication between different Java Virtual Machines (JVMs) over a network. It allows a client program to invoke methods on objects located in a remote JVM, as if they were local objects. RMI provides a way for distributed applications to interact and communicate seamlessly, abstracting the complexities of network communication.

In this scenario, the client application communicates with a server application using RMI. The server exposes a remote interface that defines the methods the client can invoke remotely. The client obtains a reference to the remote object from an RMI registry and then invokes methods on that object as if it were a local object. The RMI runtime handles the details of parameter marshalling, network communication, and exception handling, making remote method invocations transparent to the developer.



Algorithm:

1. **Define Remote Interface**:
    - Create a remote interface that extends **java.rmi.Remote** and declares the methods that can be invoked remotely. In this case, the interface should have a method that takes two strings as parameters and returns a concatenated string.
2. **Implement Remote Interface**:
    - Implement the remote interface in a server class. This class provides the actual implementation for the methods declared in the remote interface.
3. **Create Server Application**:
    - In the server application, create an instance of the server class, bind it to an RMI registry, and wait for client requests.
4. **Create Client Application**:
    - In the client application, look up the server object from the RMI registry, obtain a reference to it, and then invoke the methods on the server object as needed.

Conclusion:

RMI provides a powerful mechanism for building distributed applications in Java, allowing for seamless communication between client and server applications. By abstracting the complexities of network communication, RMI simplifies the development of distributed systems and enables developers to focus on the application logic rather than the network infrastructure.

CL3 Practical 4

Title: Simulating requests coming from clients and distributing them among servers using load balancing algorithms.

Software Requirements:

- Python
- Ubuntu

Theory:

Load balancing is a critical aspect of distributed systems, where incoming requests from clients are distributed among multiple servers to optimize resource utilization, maximize throughput, minimize response time, and avoid overloading any single server.

Load balancing algorithms play a crucial role in this process, determining how requests are distributed among servers. Some common load balancing algorithms include:

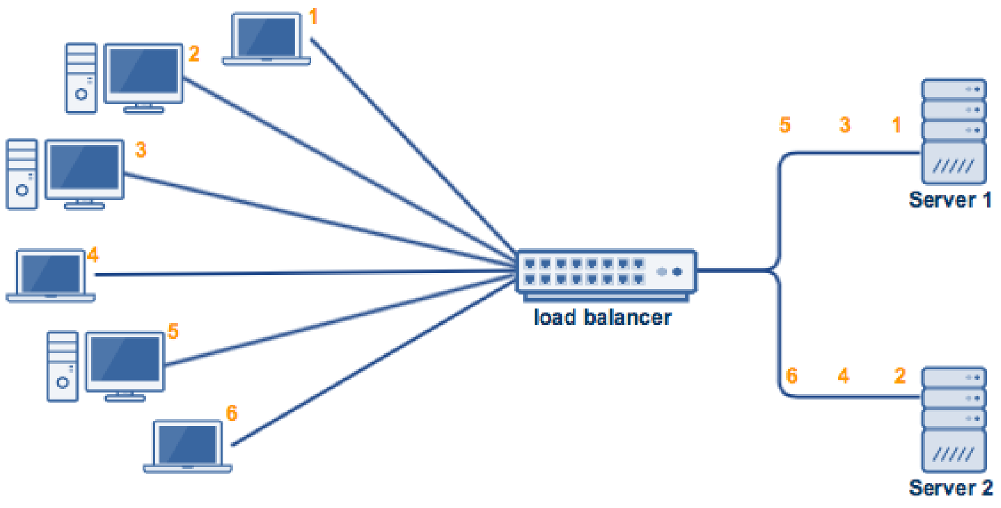
1. **Round Robin**: Requests are distributed to servers in a cyclic manner, where each server is selected in turn.
2. **Least Connections**: Requests are sent to the server with the fewest active connections, aiming to evenly distribute the load.
3. **Weighted Round Robin**: Similar to Round Robin, but servers are assigned different weights based on their capacities, allowing more requests to be sent to higher-capacity servers.
4. **Least Response Time**: Requests are sent to the server with the lowest response time, aiming to reduce overall response time for all requests.

Algorithm:

1. **Initialize Servers**: Create a list of servers and initialize their load and other relevant parameters.
2. **Initialize Load Balancer**: Choose a load balancing algorithm and initialize any required data structures.
3. **Simulate Requests**:

- Generate simulated requests from clients.
- For each request, use the load balancing algorithm to select a server.
- Update the load of the selected server.
- Repeat this process for a specified number of requests or a specified duration.

4. **Measure Performance Metrics**:
    - Calculate and analyze performance metrics such as server load distribution, response time, throughput, etc.
5. **Display Results**: Display the results of the simulation and performance metrics.



Conclusion:

Simulating requests and distributing them among servers using load balancing algorithms is essential for maintaining optimal performance and resource utilization in distributed systems. By simulating different scenarios and analyzing performance metrics, we can gain insights into the effectiveness of various load balancing algorithms and make informed decisions about their use in real-world applications.

CL Practical 5

Title: Optimization of Genetic Algorithm Parameters in Hybrid Genetic Algorithm-Neural Network Modeling: Application to Spray Drying of Coconut Milk

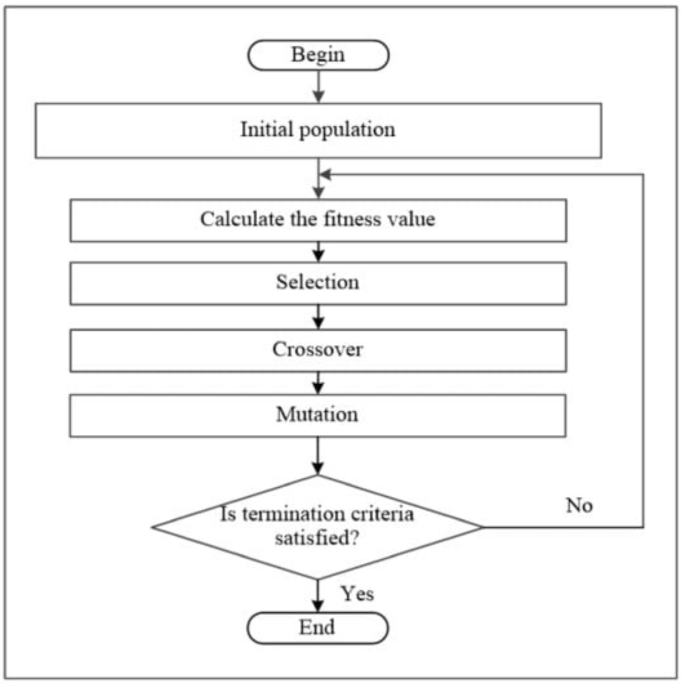
Software Requirements:

- Python
- TensorFlow or other neural network libraries
- Genetic algorithm optimization libraries

Theory:

In hybrid genetic algorithm-neural network modeling, genetic algorithms (GAs) are used to optimize the parameters of a neural network (NN) model. GAs mimic the process of natural selection to search for the optimal set of parameters that minimize a given objective function. In this context, the objective function represents the error between the predicted and actual values of the spray drying process of coconut milk.

The hybrid model combines the strengths of GAs and NNs. GAs explore the search space efficiently, while NNs provide a flexible and powerful function approximator. By optimizing the parameters of the NN using GAs, the hybrid model can improve its predictive accuracy and generalization performance.


Begin
Initial population
Calculate the fitness value
Selection
Crossover
Mutation
Is termination criteria satisfied?
No
Yes
End

Algorithm:

1. **Initialize Population**: Generate an initial population of chromosomes representing the parameters of the NN model.
2. **Evaluate Fitness**: Evaluate the fitness of each chromosome using the NN model and the objective function.
3. **Selection**: Select parents for the next generation based on their fitness, using techniques like roulette wheel selection or tournament selection.
4. **Crossover**: Perform crossover between pairs of parents to create offspring, exchanging genetic information.
5. **Mutation**: Introduce random changes to the offspring's chromosomes to maintain genetic diversity.
6. **Replace**: Replace the old population with the new population of offspring.
7. **Repeat**: Repeat the process for a certain number of generations or until convergence.

Conclusion:

The optimization of genetic algorithm parameters in hybrid genetic algorithm-neural network modeling is a powerful approach for improving the accuracy and efficiency of predictive models. In the context of spray drying of coconut milk, this hybrid model can lead to better process optimization and product quality control.

CL 3 Practical 6

Title: Implementation of Clonal Selection Algorithm Using Python

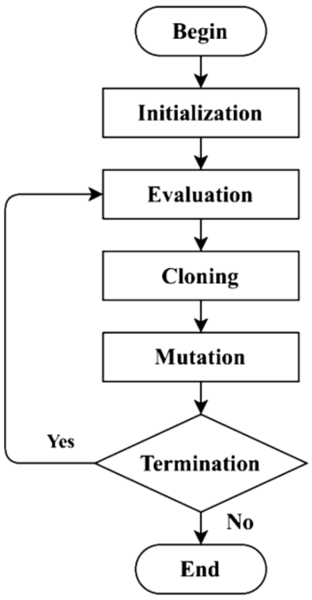
Software Requirements:

- Python
- NumPy
- Matplotlib

Theory:

The Clonal Selection Algorithm (CSA) is a population-based optimization algorithm inspired by the clonal selection process of the immune system. It is commonly used for solving optimization problems, particularly in the field of computational intelligence. The algorithm is based on the idea of simulating the clonal selection process that occurs when the immune system responds to an antigen.

In CSA, a population of antibodies (candidate solutions) is initially generated randomly. During the optimization process, the antibodies with higher affinity (better solutions) are cloned, creating a population of clones. These clones then undergo hypermutation (random changes) to introduce diversity in the population. The antibodies with the highest affinity in the clone population are selected to replace the least fit antibodies in the original population, mimicking the process of natural selection.



Algorithm:

1. **Initialization**:

Generate an initial population of antibodies (candidate solutions) randomly. Each antibody represents a potential solution to the optimization problem.

2. **Affinity Calculation**:

Affinity represents the quality of the solution and is typically defined by the objective function of the optimization problem. Higher affinity indicates a better solution.

3. **Cloning**:

Clone antibodies in proportion to their affinity. Antibodies with higher affinity are cloned more, increasing their representation in the population.

4. **Hypermutation**:

Hypermutation is a process of introducing random changes to the cloned antibodies, simulating the genetic diversity in the immune system.

5. **Selection**:

Select antibodies with the highest affinity from the mutated antibodies. These antibodies are considered the fittest and are selected to replace antibodies in the original population.

6. **Replacement**:

Replace antibodies in the original population with the selected antibodies. This step ensures that the population evolves towards better solutions over iterations.

7. **Termination**:

Repeat the cloning, hypermutation, selection, and replacement steps for a specified number of iterations or until a convergence criterion is met.

Conclusion:

The Clonal Selection Algorithm is a powerful optimization technique that has been successfully applied to various optimization problems. Its ability to mimic the immune system's clonal selection process makes it particularly effective in dynamic optimization problems and multimodal optimization scenarios.

Part B

CL 3 Practical 7

Title: Applying Artificial Immune Pattern Recognition for Structure Damage Classification

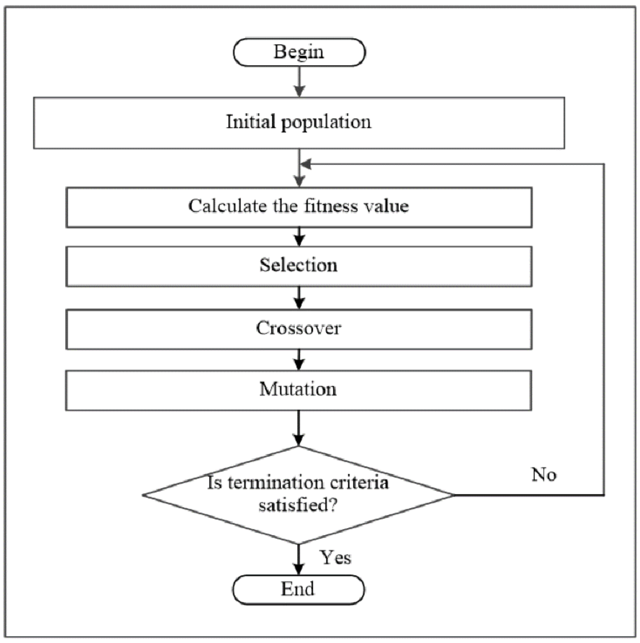
Software Requirements:

- Python (for implementation)
- Scikit-learn (for machine learning)
- Matplotlib (for visualization, optional)

Theory:

Artificial immune systems (AIS) are computational models inspired by the principles and processes of the natural immune system. AIS can be used for pattern recognition tasks, including structure damage classification. The basic idea is to represent data instances (e.g., features describing structural properties) as antigens, and use a set of antibodies to classify them into different classes (e.g., types of damage).

The immune system's ability to distinguish between self and non-self antigens is mimicked by training the AIS on a dataset where instances are labeled with their corresponding classes. During training, the AIS learns to recognize patterns associated with each class, effectively classifying new instances.


Begin
Initial population
Calculate the fitness value
Selection
Crossover
Mutation
Is termination criteria satisfied?
No
Yes
End

Algorithm:

1. **Initialization**: Generate an initial set of antibodies (classification rules) randomly.
2. **Affinity Calculation**:

Calculate the affinity of each antibody to each antigen (data instance) based on their similarity. Affinity represents the likelihood that an antibody will classify an antigen into a specific class.

3. **Clonal Selection**:

Clone antibodies in proportion to their affinity, creating a population of clones.

4. **Hypermutation**:

Introduce random changes (mutations) to the cloned antibodies to increase diversity in the population.

5. **Selection**:

Select antibodies with the highest affinity to form the next generation of antibodies.

6. **Classification**:

Classify new antigens using the selected antibodies as classification rules.

7. **Evaluation**:

Evaluate the performance of the AIS using metrics such as accuracy, precision, recall, and F1-score.

Conclusion:

Artificial immune pattern recognition shows promise for structure damage classification, leveraging the immune system's ability to recognize and respond to different patterns. By implementing AIS, we can develop robust classification models capable of identifying and categorizing various types of damage in structures.

CL 3 Practical 8

Title: Implementing DEAP (Distributed Evolutionary Algorithms) Using Python

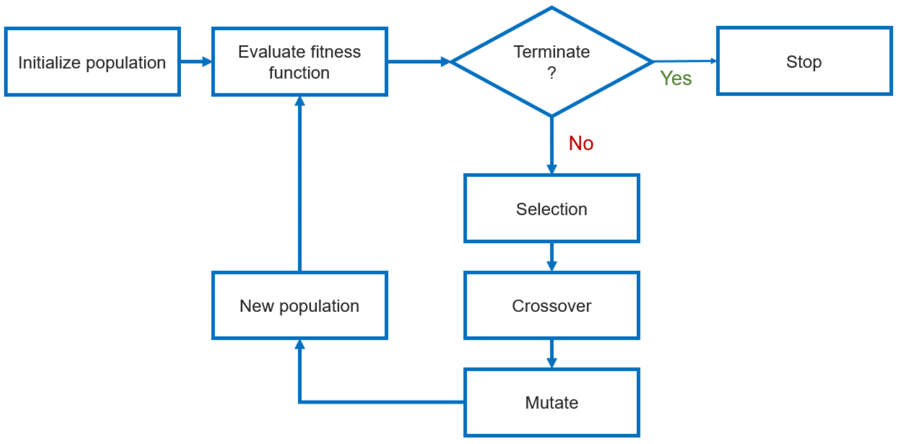
Software Requirements:

- Python
- DEAP library
- Distributed computing framework for parallelizing the evolutionary algorithm.

Theory:

DEAP (Distributed Evolutionary Algorithms in Python) is a framework for building and running evolutionary algorithms. Evolutionary algorithms are optimization algorithms inspired by natural evolution, where candidate solutions evolve over generations to find the optimal solution to a given problem.

In DEAP, a population of candidate solutions (individuals) is evolved using genetic operators such as selection, crossover, and mutation. The fitness of each individual is evaluated based on an objective function, and the fittest individuals are selected to reproduce and create the next generation. DEAP supports parallelization, allowing for the distribution of computation across multiple cores or machines, which can significantly speed up the optimization process.



Algorithm:

1. **Initialization**:

Generate an initial population of individuals randomly.

2. **Evaluation**:

Evaluate the fitness of each individual using an objective function.

3. **Selection**:

Select individuals for reproduction based on their fitness, using selection methods such as tournament selection or roulette wheel selection.

4. **Crossover**:

Perform crossover between pairs of selected individuals to create offspring.

5. **Mutation**:

Introduce random changes (mutations) to the offspring to maintain genetic diversity.

6. **Replacement**:

Replace the least fit individuals in the population with the offspring.

7. **Termination**:

Repeat the selection, crossover, mutation, and replacement steps for a specified number of generations or until a convergence criterion is met.

Conclusion:

DEAP provides a powerful framework for implementing and running distributed evolutionary algorithms in Python. By leveraging parallelization, DEAP can effectively optimize complex problems by distributing the computational workload across multiple processors or machines, leading to faster and more efficient optimization processes.

CL 3 Practical 9

Title: Design and develop a distributed Hotel booking application using Java RMI. A distributed hotel booking system consists of the hotel server and the client machines. The server manages hotel rooms booking information. A customer can invoke the following operations at his machine i) Book the room for the specific guest ii) Cancel the booking of a guest.

Software Requirements:

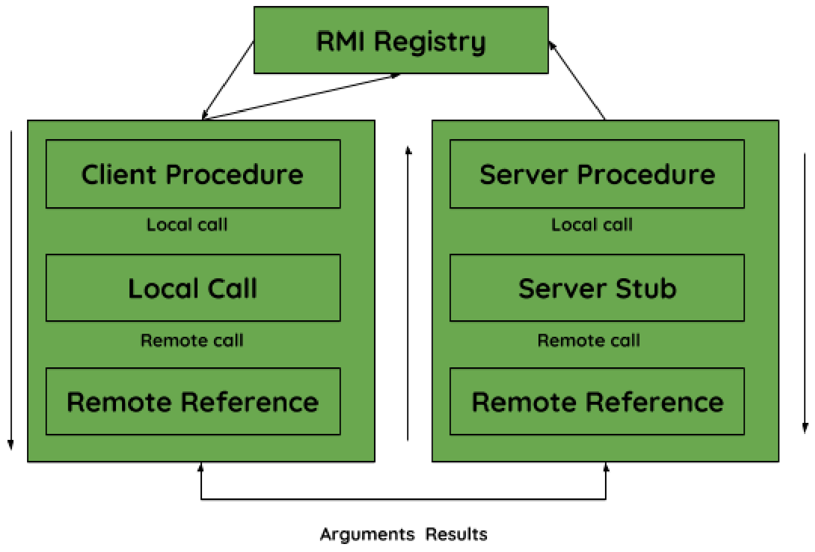
- Java Development Kit (JDK) for compiling and running Java code
- Java RMI (Remote Method Invocation) for implementing distributed communication
- IDE (e.g., IntelliJ IDEA, Eclipse) for writing and debugging code

Theory:

A distributed hotel booking application using Java RMI is designed to facilitate the management of hotel room bookings across multiple client machines. Java RMI (Remote Method Invocation) is a Java API that enables communication between different Java Virtual Machines (JVMs) over a network. In this context, the hotel server acts as the central system responsible for managing hotel room booking information, while the client machines are used by customers to interact with the server.

The application utilizes RMI to allow client machines to invoke methods on objects located in the hotel server's JVM, enabling seamless communication between the client and server components. The server implements a remote interface that declares methods for booking and canceling hotel room bookings, while the client applications interact with the server by invoking these methods.

By using Java RMI, the distributed hotel booking application provides a flexible and efficient solution for managing hotel room bookings. Clients can easily book and cancel room bookings from any location, and the server can manage booking information effectively, ensuring that the hotel operates smoothly and efficiently.



Algorithm:

1. **Define Remote Interface**:

Create a remote interface (**HotelBookingInterface**) that declares methods for booking and canceling hotel rooms.

2. **Implement Remote Interface**:

Implement the remote interface in a server class (**HotelBookingServer**) that provides the actual implementation for the booking and canceling methods.

3. **Create Server Application**:

Create a server application (**HotelBookingServerApplication**) that instantiates the server class, binds it to a registry, and waits for client requests.

4. **Create Client Application**:

Create a client application (**HotelBookingClientApplication**) that looks up the server object from the registry, obtains a reference to it, and then invokes the booking and canceling methods as needed.

Conclusion:

A distributed hotel booking application using Java RMI provides a convenient and efficient way to manage hotel room bookings. By leveraging Java RMI's remote method invocation capabilities, the application allows clients to interact with the server from remote locations, enabling easy booking and cancelation of hotel rooms

CL 3 Practical 10

Title: Implement Ant colony optimization by solving the Traveling salesman problem using python Problem statement- A salesman needs to visit a set of cities exactly once and return to the original city. The task is to find the shortest possible route that the salesman can take to visit all the cities and return to the starting city.

Software Requirements:

- Python
- NumPy
- Matplotlib

Theory:

Ant Colony Optimization (ACO) is a metaheuristic optimization algorithm inspired by the foraging behavior of ants. It is commonly used to solve combinatorial optimization problems such as the Traveling Salesman Problem (TSP). In the TSP, a salesman needs to visit a set of cities exactly once and return to the starting city, finding the shortest possible route.

In ACO, a colony of artificial ants is used to explore the solution space. Each ant constructs a solution by probabilistically selecting the next city to visit based on pheromone trails and heuristic information. Pheromone trails are updated based on the quality of the solutions found by the ants, with stronger trails indicating better paths.

Algorithm:

1. **Initialization**:

Initialize pheromone levels on each edge between cities.

Place ants at random cities.

2. **Ant Tour Construction**:

For each ant, construct a tour by probabilistically selecting the next city based on pheromone levels and heuristic information.

3. **Pheromone Update**:

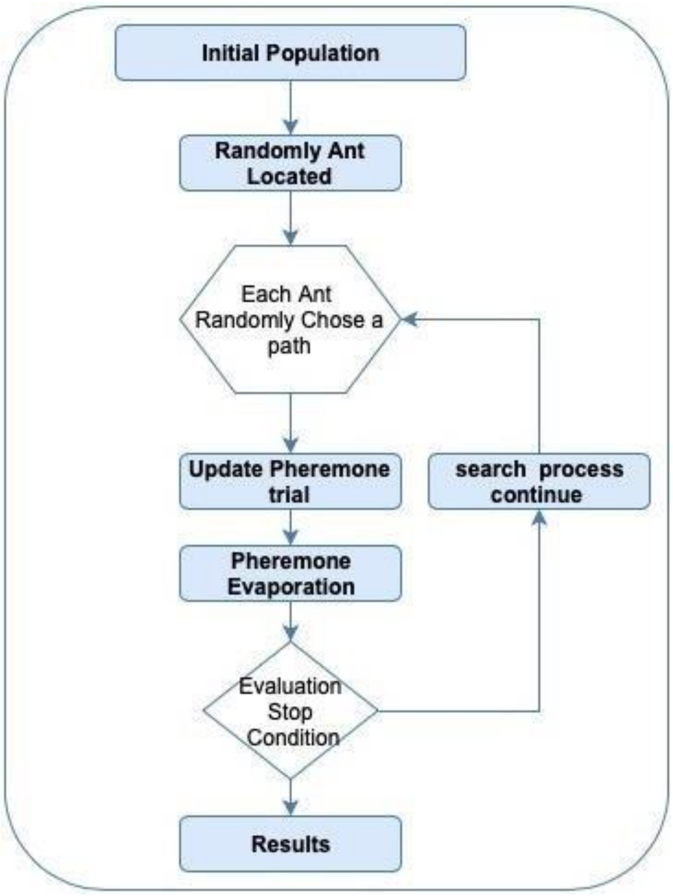
Update pheromone levels on edges based on the quality of the tours constructed by the ants.

4. **Global Update**:

Optionally, perform a global update to evaporate pheromone levels over all edges to prevent stagnation.

5. **Termination**:

Repeat the construction and update steps for a specified number of iterations or until a termination criterion is met.



Conclusion:

Ant Colony Optimization is a powerful algorithm for solving the Traveling Salesman Problem and other combinatorial optimization problems. By simulating the foraging behavior of ants, ACO effectively explores the solution space and finds high-quality solutions.