SJF

Initialize data structures and matrices for cloudlets, VMs, datacenters, commMatrix, and execMatrix.

Load data into commMatrix and execMatrix.

Initialize CloudSim and create datacenters.

Create a datacenter broker.

Create VMs and cloudlets.

Submit VMs and cloudlets to the broker.

Start the CloudSim simulation.

Get the list of received cloudlets (results).

Sort the results by cloudlet length in ascending order (shortest job first) using the SJF algorithm.

Process each job in the sorted results, execute it, and perform additional calculations as needed based on the SJF algorithm.

FCFS

Initialize data structures and matrices for cloudlets, VMs, datacenters, commMatrix, and execMatrix.

Load data into commMatrix and execMatrix.

Initialize CloudSim and create datacenters.

Create a datacenter broker using FCFSDatacenterBroker.

Create VMs and cloudlets using createVM and createCloudlet functions.

Submit VMs and cloudlets to the broker.

Start the CloudSim simulation.

Get the list of received cloudlets (results).

Schedule the received cloudlets using the FCFS scheduling policy:

a. Store the received cloudlets in a list in the order they are received.

b. Set this list as the cloudlets to be executed.

Process each job in the scheduled results and execute it:

a. As the cloudlets are in FCFS order, execute each cloudlet in the list sequentially.

b. Wait for the cloudlet to complete its execution.

c. Check if all cloudlets are executed. If yes, finish the execution.

If there are still remaining cloudlets in the list, continue executing the next cloudlet in FCFS order.

Stop the CloudSim simulation.

Note: The FCFS algorithm is reflected in step 9 where the cloudlets are scheduled and processed in the order they are received without any prioritization or preemption.

QLearning

Initialize data structures, matrices for cloudlets, VMs, data centers, communication matrix (commMatrix), and execution matrix (execMatrix).

Load data into commMatrix and execMatrix.

Initialize Q-table with zeros, learning rate (ALPHA), discount factor (GAMMA), exploration rate (EPSILON), number of training episodes (NUM\_EPISODES), and maximum steps per episode (MAX\_STEPS).

Create CloudSim simulation and data centers.

Create a QLearningDatacenterBroker.

Create VMs and cloudlets.

Submit VMs and cloudlets to the broker.

Start the CloudSim simulation.

Process each event in the broker:

a. If it's a cloudlet submission event:

Update the cloudlet status to INEXEC.

Select an action using the Q-table (exploit or explore).

Update the allocated MIPS for the selected VM.

Update the Q-table with the current state, selected action, and the current time as a reward.

b. If it's a cloudlet return event:

Update the cloudlet status to SUCCESS.

Increment the number of submitted cloudlets.

Determine the next state (if available).

Select an action using the Q-table (exploit or explore).

Update the Q-table with the previous state, selected action, and the current time as a reward.

Stop the CloudSim simulation.

The Q-learning algorithm uses a Q-table to guide the scheduler's decision-making process by selecting actions (VMs) based on the current state (cloudlet) and the learned Q-values. Over time, the Q-table is updated through trial and error to improve the performance of the scheduler in allocating cloudlets to VMs.

A2C

Initialize data structures, matrices for cloudlets, VMs, data centers, communication matrix (commMatrix), and execution matrix (execMatrix).

Load data into commMatrix and execMatrix.

Initialize the actor and critic models as empty HashMaps, learning rates for actor and critic (actorLearningRate and criticLearningRate), and a discount factor (discountFactor).

Create CloudSim simulation and data centers.

Create an A2CDataCenterBroker.

Create VMs and cloudlets.

Submit VMs and cloudlets to the broker.

Start the CloudSim simulation.

Process each event in the broker:

a. If it's a cloudlet return event:

Calculate the reward for the completed cloudlet based on its execution time.

Calculate the state representation based on the VM and cloudlet.

Update the actor and critic models using the A2C algorithm with the current state and reward.

Select an action (VM) using the actor model based on the current state.

Submit the cloudlet for execution on the selected VM.

Stop the CloudSim simulation.

The A2C algorithm uses two models, the actor model and the critic model, to guide the scheduler's decision-making process. The actor model estimates the probability of selecting each VM, and the critic model estimates the value of each state (combination of VM and cloudlet). The models are updated using the TD error (Temporal Difference error), which represents the difference between the estimated reward and the actual reward received after taking an action. Over time, the models are updated to improve the scheduler's performance in allocating cloudlets to VMs.

DoubleQLearning

Initialize data structures, matrices for cloudlets, VMs, data centers, communication matrix (commMatrix), and execution matrix (execMatrix).

Load data into commMatrix and execMatrix.

Initialize the Q-tables (qTable1 and qTable2) as 2D arrays with all values set to 0.

Create CloudSim simulation and data centers.

Create a DoubleQLearningDatacenterBroker.

Initialize parameters such as learning rate (ALPHA), discount factor (GAMMA), exploration rate (EPSILON), number of training episodes (NUM\_EPISODES), and maximum number of steps per episode (MAX\_STEPS).

Implement the Q-Learning algorithm within the DoubleQLearningDatacenterBroker:

a. For each episode (from 1 to NUM\_EPISODES):

Reset the current state to the initial state (e.g., -1).

For each step (from 1 to MAX\_STEPS):

Select an action using the epsilon-greedy policy (with exploration rate EPSILON).

Perform the selected action.

Observe the reward and the next state.

Update the Q-tables using the Q-Learning algorithm.

Set the next state as the current state.

b. After training, the Q-tables will represent the learned Q-values for each state-action pair.

Implement Q-value update in the DoubleQLearningDatacenterBroker:

a. The Q-tables are updated for each action taken in the simulation based on the reward and the next state.

b. The update is done using a combination of the two Q-tables to avoid the overestimation bias in the Q-values.

c. The action is selected based on the epsilon-greedy policy during training.

Process CloudSim events in DoubleQLearningDatacenterBroker:

a. If a cloudlet is submitted, process the submission, update the Q-tables, and select the next action based on the current state.

b. If a cloudlet returns, process the return, update the Q-tables, and select the next action based on the current state.

Start the CloudSim simulation and run the training episodes.

Stop the CloudSim simulation after all training episodes are completed.

The Double Q-Learning Scheduler is now trained and ready to be used for scheduling cloudlets.

The Double Q-Learning algorithm uses two Q-tables to prevent overestimation of Q-values, which can lead to better convergence and more stable learning compared to traditional Q-Learning. The algorithm learns to associate states with actions that yield higher rewards, and this knowledge is used to make better decisions during the simulation.