**Q2 – Server CU Rule**

**Part 1 - planning**

1. The state space:  
   }}  
   The state’s defines the remaining jobs that were not served yet.  
   The start state is   
   Therefore, we have   
   As we can select at each stage s whether some job was finished(not in s) or wasn’t (in s)

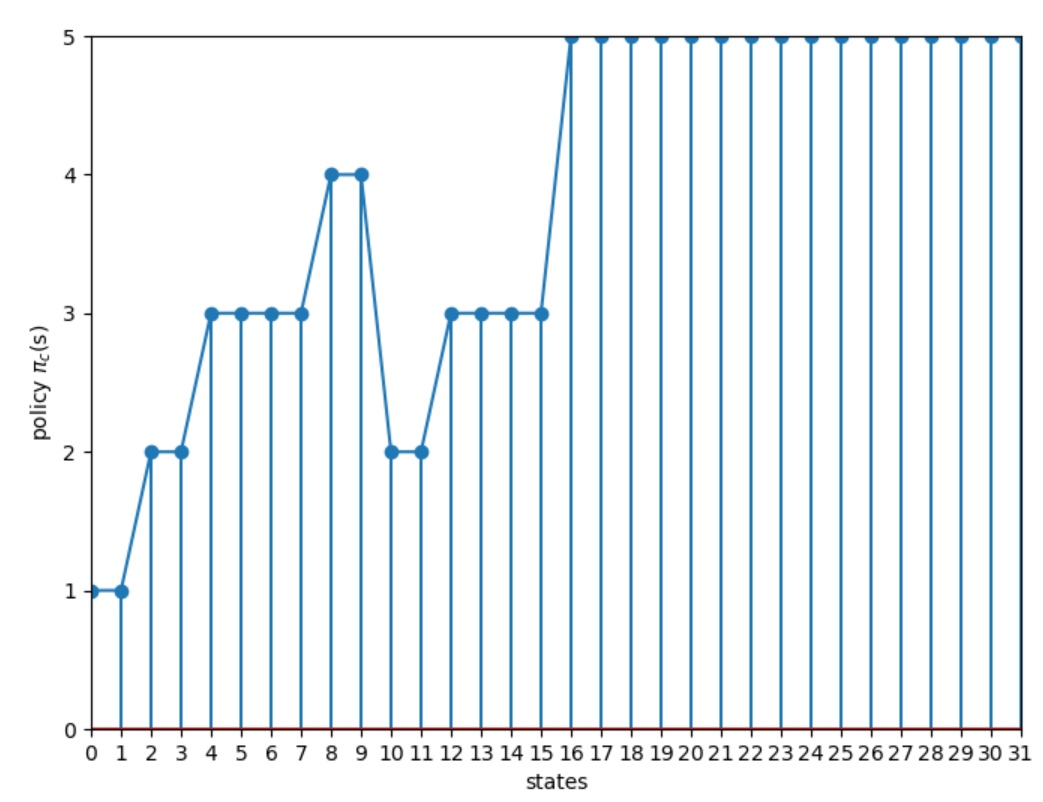
In our case, N=5 so we have 32 different states.

The action space:  
}  
The action index defines the job that the server tries to serve at that time index.  
It cannot be an index that doesn’t belong to the state at that time index.

1. The cost per time step is as follows:  
   We will calculate the value function in python using fixed policy value iteration algorithm.  
   We will use a N bit id per state. In each state, a lit bit defines that the job is unfinished.  
   So in our case the starting state is 31 and the terminal state is 0.

We will use the fact that this is a finite state problem, with a terminal state as absorbing state. Meaning, with probability 1 we will get to the terminal state (with cost 0 defined for it).   
Thus, we can backward recurse in order to compute the value function of the fixed policy.  
The equation for value iteration as we calculated in previous assignment is:

1. Plot of the values of the policy that selects the job with the maximal cost , from the remaining unfinished jobs-



As described in previous section, the representation of state in the plot above is a decimal number matching the binary number where bit if in state job is not yet finished(still in system).

1. For applying the policy iteration algorithm, we need to implement now the policy improvement stage. In our case, as we saw in previous assignment, the bellman equation is as follows (in order to minimize cost):

And the improved policy -

In each iteration in the PI algorithm, we first calculate the Value for policy -

(using FPVI).

We apply the policy improvement above and update our policy to the improved policy found - .

We calculate the Value for policy - (using FPVI).

If we have not improved the value reached by policy and we achieve-

.

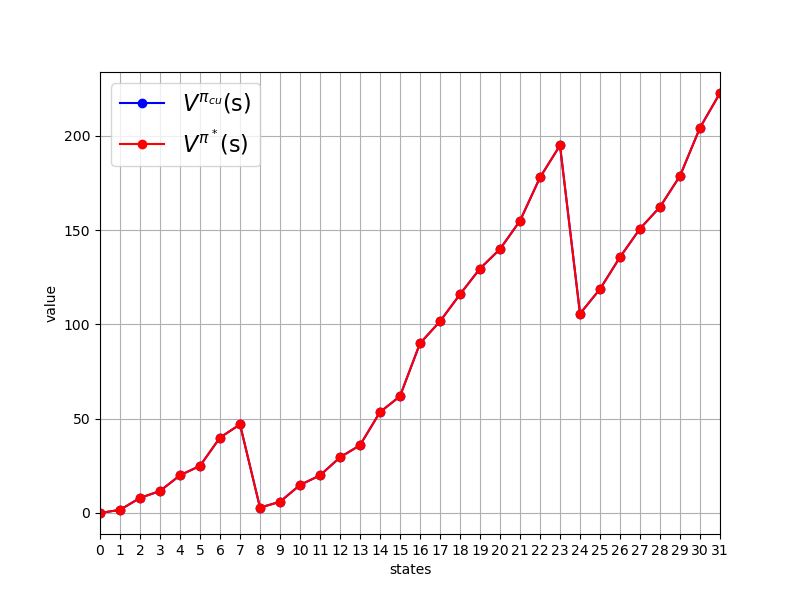
Otherwise, we return the steps for iteration (with policy ).

For initial iteration we set- and got coverage to optimal policy after 2 iterations.  
We actually improved the policy only once (the second time it didn’t improve)

Plot below shows value over iterations of the algorithm (.

A screenshot of a cell phone

Description automatically generated

1. After obtaining ,we get the same value as for the optimal policy. Thus, we can see that the cu rule gives an optimal policy.  
   

Plot of vs. :  
A screenshot of a cell phone

Description automatically generated

For all states as expected of optimal policy.

1. The simulator function implemented (assuming ) –

**Part 2 - learning**

1. The first step size is moving average, as we saw in the lectures. It basically does averaging on all occurrences in which we simulate over some stage. We can see the error remains high as alpha gets low too rapidly, thus the update of the new value is relatively small and the convergence is slow due to it.  
   plot of alpha 1 TD0:  
   The second step size remains constant to all iterations thus the convergence relatively keeps the same pace but the start is slow relative to other step sizes tested.  
   plot of alpha 2 TD0:  
   The third step size starts with ~0.1 alpha and the pace in which the step size getting smaller is less sharp than the first step size, thus we still learn with a relatively high pace even on the long term iterations.  
   plot of alpha 3 TD: