a) State Space: $X = \{1,2,3,4\} \times \{1,2,3,4\}$, that is, $X_t = (j_t, k_t) \in \{1,2,3,4\}^2$, where j_t is the current trailer position and k_t is the current job position.

Control Space: u_t is the next position of the trailer, $U = \{1,2,3,4\}$.

Transition kernel: $P[(j_{t+1}, k_{t+1}) = (j,k) | j_t, k_t, u_t] = \begin{cases} p_{k_t,k}, & if\ j = u_t; \\ 0, & otherwise. \end{cases}$

Step-wise cost:

$$c(j,k,u) = \begin{cases} 0, & if\ k = 1, u = j; \\ 200, & if\ k > 1, u = j = 1; \\ 50, & if\ k = u = j > 1; \\ 100, & if\ k > 1, u = j \notin \{1,k\}; \\ 300, & if\ k = 1, u \neq j; \\ 300 + 200, & if\ k > 1, u = 1, j \neq 1; \\ 300 + 50, & if\ k > 1, u = k, j \neq k; \\ 300 + 100, & if\ k > 1, u \notin \{1,k\}, j \neq u; \end{cases}$$

The optimal value function $v^*: \{1,2,3,4\}^2 \to R$ satisfies the following dynamic programming equation

$v^*(j,k) = \min_{u \in \{1,2,3,4\}} \{c(j,k,u) + \gamma \sum_{l=1}^{4} p_{k_t,l} v^*(u,l)\}$ for all $(j,k) \in X$.

b) Value iteration:

```
% [v_lo,n_it] = dne1_value_iteration_revised (0.95,10000);
function [v_lo,n_it] = dne1_value_iteration_revised (alpha,max_it)
i = 0;
n_it = max_it;
v=[0,0,0,0];
vv=[0,0,0,0];
v_lo=[0,0,0,0];
v_up=[0,0,0,0];
while (i < n_it)
    vv(1) = max(0+alpha*(0.1*v(1)+0.3*v(2)+0.3*v(3)+0.3*v(4)));
    vv(2) = max(-100+alpha*(0.5*v(3)+0.2*v(4)));
    vv(3) = max(-200+alpha*(0.4*v(1)));
    vv(4) = max(-50+alpha*(0.5*v(2)+0.8*v(3)+0.6*v(4)));
    v_lo = vv + min(vv-v)*alpha/(1-alpha);
    v_up = vv + max(vv-v)*alpha/(1-alpha);
    if (isequal(v_lo,v_up))
        n_it=i;
    end
    i=i+1;
    v=v_lo;
end
end
```

c) Policy iteration:

```
In [9]:
```

```
def policy_evaluation(self):
        next_value_table = [[0.00] * self.env.width
                                    for _ in range(self.env.heig
ht)]
        # Bellman Expectation Equation for the every states
```

```
        for state in self.env.get_all_states():
            value = 0.0
            for action in self.env.possible_actions:
                next_state = self.env.state_after_action(state,
action)
                reward = self.env.get_reward(state, action)
                next_value = self.get_value(next_state)
                value += (self.get_policy(state)[action] *
                          (reward + self.discount_factor * next_
value))
            next_value_table[state[0]][state[1]] = round(value,
2)
        self.value_table = next_value_table
```

In [14]:

```
def policy_improvement(self):
        next_policy = self.policy_table
        for state in self.env.get_all_states():
            value = -99999
            max_index = []
            result = [0.0, 0.0, 0.0, 0.0]  # initialize the poli
cy

            # for every actions, calculate
            # [reward + (discount factor) * (next state value fu
nction)]
            for index, action in enumerate(self.env.possible_act
ions):
                next_state = self.env.state_after_action(state,
action)
                reward = self.env.get_reward(state, action)
                next_value = self.get_value(next_state)
                temp = reward + self.discount_factor * next_valu
e
                if temp == value:
                    max_index.append(index)
                elif temp > value:
                    value = temp
                    max_index.clear()
                    max_index.append(index)

            # probability of action
            prob = 1 / len(max_index)

            for index in max_index:
                result[index] = prob

            next_policy[state[0]][state[1]] = result

        self.policy_table = next_policy
```

d) Linear programming:

```
% v = dne1_LP
function v = dne1_LP
clear();
f=[1;1;1;1];

A=[0.95,0.285,0.285,0.285;
    0,0.475,0.76,0.57;
    0,0,0.475,0.19;
    0.38,0,0,0];

b=[0;-50;-100;-200];

v=linprog(f,A,b);

end
```