Single machine scheduling Using Deep Q-Network

명지대학교 산업경영공학과
박 인 범 (inbeom@mju.ac.kr)

Pytorch 설치

- 개인 pc에 설치한 경우
  - Cmd에서 아래 명령어 입력
  - pip install pytorch
- Anaconda에서 설치한 경우
  - Conda 개인 환경(env)에서 아래 명령어 입력
  - conda install pytorch

목차

1. Data set
2. Parameter
3. JobScheduler Class
4. ReplayBuffer Class
5. Qnet Class
6. Train_dqn function
7. Test_dqn function
8. 실행 결과

1. Data set

```
# 입력 데이터
PROCESSING_TIMES = {
    'A': [5, 5],
    'B': [3, 3, 3],
    'C': [7, 7],
    'D': [4],
    'E': [2],
    'F': [6, 6, 6]
}

DUE_DATES = {
    'A': [10, 10],
    'B': [15, 15, 15],
    'C': [25, 25],
    'D': [12],
    'E': [14],
    'F': [21, 21, 21]
}
```

- Job type 별 생산시간과 마감일 정보
- Job type = ['A', 'B' 'C', 'D', 'E', 'F']
- 예시 데이터에서는 'A' 2개, 'B' 3개, 'C' 2개, 'D' 1개, 'E' 1개, 'F' 3개 생산

2. Parameter

```
# 학습 파라미터
N_EPISODES = 7000
GAMMA = 0.99
BATCH_SIZE = 128
BUFFER_LIMIT = 7000
LEARNING_RATE = 0.001

# state, action 정의 파라미터
USE_COUNT_STATE = True
USE_JOB_TYPE_ACTION = True
RULE_LIST = ['SPT', 'LPT', 'EDD', 'LDD']
```

- N_EPISODES : 학습 횟수 설정
- GAMMA : Discount factor 설정
- BATCH_SIZE : 한 번의 학습 스텝에서 사용할 경험(샘플)의 수 설정
- BUFFER_LIMIT : 경험 replay buffer의 최대 크기 설정
- LEARNING_RATE : 인공신경망 학습의 학습률 설정
- USE_COUNT_STATE : 'Count' 상태 정의 사용시 True, 'Progress' 상태 정의 사용시 False
- USE_JOB_TYPE_ACTION : ' Job_type' 행동 정의 사용시 True, 'Rule' 행동 정의 사용시 False
- RULE_LIST : 'Rule' 행동 정의 사용시 사용할 Rule 설정

3. JobScheduler Class

- __init__ Method : 작업 스케줄러의 초기 설정을 수행하는 Method

```
def __init__(self, action_type='job_type', rule_list=None, state_type='count', state_interval=1):
        self.action_type = action_type
        self.rule_list = rule_list if rule_list else RULE_LIST
        self.state_type = state_type
        self.state_interval = state_interval
        self.reset()
```

- **[매개변수]**

  - action_type: 행동을 표현하는 방식 결정 'Job_type'이면 직접 작업을 선택하고, 'Rule'이면 스케줄링 규칙 선택

  - rule_list: 사용할 스케줄링 규칙들의 목록 (기본값은 RULE_LIST)

  - state_type: 상태를 표현하는 방식 결정 ('Count'는 각 job_type별 남은 job의 수, 'Progress'는 전체 진행률)

  - state_interval: 'Progress' 상태 유형에서 사용되는 간격

- **[동작]**

  1. 입력 받은 매개변수들을 객체의 속성으로 저장
  2. reset( ) Method를 호출하여 환경을 초기화

3. JobScheduler Class

- reset Method : 환경을 초기 상태로 되돌리는 Method

```
def reset(self):
        self.processing_times = PROCESSING_TIMES
        self.total_jobs = {job_type: len(times) for job_type, times in self.processing_times.items()}
        self.remaining_jobs = self.total_jobs.copy()
        self.time = 0
        self.tardiness = 0
        self.job_sequence = []

        self.due_dates = DUE_DATES

        self.job_list = []
        for job_type, times in self.processing_times.items():
            for i, time in enumerate(times, 1):
                self.job_list.append((job_type, i, time))

        self.job_list.sort(key=lambda x: (x[2], x[1]))

        return self.get_state()
```

- **[동작]**

1. job별 처리 시간(processing_times) 설정
2. 각 job_type별 총 job 수와 남은 job 수 초기화
3. 현재 시간, 총 지연 시간, job 순서 초기화
4. 각 job의 마감일(due_dates) 설정
5. 모든 job을 job_list에 추가하고 처리 시간에 따라 정렬
6. 초기 상태를 반환

3. JobScheduler Class

- Step Method : 주어진 행동에 따라 환경을 한 단계 진행시키는 Method

```
def step(self, action):
        if self.action_type == 'job_type':
            job_type, job_id = self.get_valid_job_type_action(action)
        else:
            job_type, job_id = self.get_valid_rule_action(action)

        if job_type is None or job_id is None:
            return self.get_state(), 0, True

        job_index = next(i for i, job in enumerate(self.job_list) if job[0] == job_type and job[1] == job_id)

        processing_time = self.job_list[job_index][2]
        due_date = self.due_dates[job_type][job_id - 1]

        self.time += processing_time
        self.job_sequence.append(f"{job_type}")

        self.remaining_jobs[job_type] -= 1

        tardiness = max(0, self.time - due_date)
        self.tardiness += tardiness

        self.job_list.pop(job_index)

        done = len(self.job_list) == 0
        reward = -tardiness

        return self.get_state(), reward, done
```

3. JobScheduler Class

- Step Method : 주어진 행동에 따라 환경을 한 단계 진행시키는 Method

- **[매개변수]**
  - action (선택된 행동)

- **[동작]**

1. action type에 따라 적절한 job_type 또는 rule 선택
2. 선택된 job 처리:
   - 현재 시간을 job의 처리 시간만큼 증가
   - job 순서에 현재 job 추가
   - 남은 job의 수 갱신
3. job의 지연 시간을 계산하고 총 지연 시간에 추가
4. 처리된 job을 job_list에서 제거
5. 모든 job이 완료되었는지 확인
6. 새로운 상태, 보상, 종료 여부 반환

3. JobScheduler Class

- get_valid_job_type_action Method : 유효한 job_type 행동을 반환하는 Method

```
def get_valid_job_type_action(self, action):
        job_types = list(self.processing_times.keys())
        for _ in range(len(job_types)):
            job_type = job_types[action % len(job_types)]
            if self.remaining_jobs[job_type] > 0:
                job_id = self.total_jobs[job_type] - self.remaining_jobs[job_type] + 1
                return job_type, job_id
            action = (action + 1) % len(job_types)
        return None, None
```

- **[매개변수]**
  - action (선택된 행동)

- **[동작]**
  1. 행동에 해당하는 job_type 선택
  2. 선택된 job_type에 남은 job이 있는지 확인
  3. 유효한 job이 있으면 해당 job_type 반환
  4. 유효한 job이 없으면 다음 job_type 확인
  5. 모든 job_type을 확인해도 유효한 job이 없으면 None 반환

3. JobScheduler Class

- get_valid_rule_action Method : 유효한 rule 기반 액션을 반환하는 Method

```
def get_valid_rule_action(self, action):
        for _ in range(len(self.rule_list)):
            rule = self.rule_list[action % len(self.rule_list)]
            job = self.select_job_by_rule(rule)
            if job is not None:
                return job
            action = (action + 1) % len(self.rule_list)
        return None, None
```

- **[매개변수]**
  - action (선택된 행동)

- **[동작]**
  1. 행동에 해당하는 rule 선택
  2. 선택된 rule을 적용하여 job 선택
  3. 유효한 job이 선택되면 해당 job 반환
  4. 유효한 job이 없으면 다음 rule 확인
  5. 모든 rule을 확인해도 유효한 job이 없으면 None 반환

3. JobScheduler Class

- select_job_by_rule Method : 주어진 규칙에 따라 작업을 선택하는 Method

```
def select_job_by_rule(self, rule):
        available_jobs = [job for job in self.job_list if self.remaining_jobs[job[0]] > 0]
        if not available_jobs:
            return None

        if rule == 'SPT':
            return min(available_jobs, key=lambda j: j[2])[0:2]
        elif rule == 'LPT':
            return max(available_jobs, key=lambda j: j[2])[0:2]
        elif rule == 'EDD':
            return min(available_jobs, key=lambda j: self.due_dates[j[0]][j[1]-1])[0:2]
        elif rule == 'LDD':
            return max(available_jobs, key=lambda j: self.due_dates[j[0]][j[1]-1])[0:2]
```

- **[매개변수]**
  - rule (적용할 규칙)

- **[동작]**

1. 현재 가능한 job들에 대한 필터링 실시
2. rule에 따라 job 선택:
   - SPT (Shortest Processing Time): 가장 짧은 처리 시간을 가진 작업 선택
   - LPT (Longest Processing Time): 가장 긴 처리 시간을 가진 작업 선택
   - EDD (Earliest Due Date): 가장 이른 마감일을 가진 작업 선택
   - LDD (Latest Due Date): 가장 늦은 마감일을 가진 작업 선택
3. 선택된 job 반환

3. JobScheduler Class

- get_state Method : 현재 환경의 상태를 반환하는 Method

```
def get_state(self):
        if self.state_type == 'count':
            return list(self.remaining_jobs.values())
        elif self.state_type == 'progress':
            total_jobs = sum(self.total_jobs.values())
            completed_jobs = total_jobs - sum(self.remaining_jobs.values())
            return [completed_jobs / total_jobs]
```

[동작]

1. state_type이 'Count'인 경우:
   - 각 job_type별 남은 job 수를 튜플로 반환
2. state_type이 'Progess'인 경우 :
   - 전체 job 수를 계산
   - 완료된 job 수를 계산
   - (job 진행률 = 완료된 job 수 / 전체 job 수)를 계산하여 리스트로 반환

3. JobScheduler Class

- get_state_size Method : 상태 공간의 크기를 반환하는 Method

```
def get_state_size(self):
        if self.state_type == 'count':
            return len(self.total_jobs)
        elif self.state_type == 'progress':
            return 1
```

- **[동작]**

  1. state_type이 'Count'인 경우:
     - job_type의 수를 반환
  2. state_type이 'Progess'인 경우 :
     - 1 반환

3. JobScheduler Class

- get_valid_actions Method : 유효한 액션들의 목록을 반환하는 Method

```
def get_valid_actions(self):
        if self.action_type == 'job_type':
            return [i for i, job_type in enumerate(self.processing_times.keys()) if self.remaining_jobs[job_type] > 0]
        else:
            return list(range(len(self.rule_list)))
```

- **[동작]**

  1. action_type이 'job_type'인 경우:
     - 모든 job_type을 확인
     - 각 job_type에 대해 아직 남아있는 job이 있는지 확인
     - 남아있는 job에 대해 해당 job_type의 번호를 목록에 추가
     - 아직 할당되지 않은 job들에 대한 목록 반환
  2. action_type이 'rule'인 경우 :
     - 0부터 가능한 모든 rule의 번호를 목록에 추가한 뒤 반환

4. ReplayBuffer Class

- __init__ Method : Replay Buffer를 초기화하는 Method

```
def __init__(self):
        self.buffer = collections.deque(maxlen=BUFFER_LIMIT)
```

- **[동작]**
  1. 최대 크기가 BUFFER_LIMIT인 deque를 생성하여 buffer에 저장

4. ReplayBuffer Class

- Put Method : 새로운 transition을 buffer에 추가하는 Method

```
def put(self, transition):
        self.buffer.append(transition)
```

- **[매개변수]**
  - transition (상태, 행동, 보상, 다음 상태, 종료 여부를 포함하는 튜플)

- **[동작]**
  1. 주어진 transition을 buffer에 추가

4. ReplayBuffer Class

- sample Method : buffer에 무작위로 n개의 샘플을 추출하는 Method

```
def sample(self, n):
        mini_batch = random.sample(self.buffer, n)
        s_lst, a_lst, r_lst, s_prime_lst, done_mask_lst = [], [], [], [], []

        for transition in mini_batch:
            s, a, r, s_prime, done_mask = transition
            s_lst.append(s)
            a_lst.append([a])
            r_lst.append([r])
            s_prime_lst.append(s_prime)
            done_mask_lst.append([done_mask])
        return torch.tensor(s_lst, dtype=torch.float), torch.tensor(a_lst), \
               torch.tensor(r_lst), torch.tensor(s_prime_lst, dtype=torch.float), \
               torch.tensor(done_mask_lst)
```

- **[매개변수]**
  - n (추출할 샘플의 수)

- **[동작]**
  1. buffer에서 n개의 transition을 무작위로 선택
  2. 선택된 transition들을 상태, 행동, 보상, 다음 상태, 종료 여부로 분리
  3. 각 요소를 Pytorch tensor로 변환하여 반환

4. ReplayBuffer Class

- size Method : 현재 buffer에 저장된 transition의 수를 반환하는 Method

```
def size(self):
        return len(self.buffer)
```

[동작]

1. 현재 buffer에 저장된 transition 수를 반환

5. Qnet Class

- __init__ Method : Q-network를 초기화 하는 Method

```
def __init__(self, state_size, action_size):
        super(Qnet, self).__init__()
        self.fc1 = nn.Linear(state_size, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, action_size)
```

- **[매개변수]**
  - state_size (상태 벡터의 크기)
  - action_size (가능한 행동의 수)

- **[동작]**
  1. 3개의 fully connected 레이어를 생성 (input → 128 → 128 → output)

5. Qnet Class

- forward Method : network의 순전파를 수행하는 Method

```
def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

- **[매개변수]**
  - x (입력 상태)

- **[동작]**
  1. 입력을 첫 번째 레이어에 통과 시키고 ReLU 활성화 함수 적용
  2. 두 번째 레이어에 통과시키고 ReLU 활성화 함수 적용
  3. 마지막 레이어에 통과시켜 Q-Value 출력

5. Qnet Class

- sample_action Method : ε-greedy 정책에 따라 행동을 선택하는 Method

```
def sample_action(self, obs, epsilon, valid_actions):
        out = self.forward(obs)
        coin = random.random()
        if coin < epsilon:
            return random.choice(valid_actions)
        else:
            return valid_actions[out[valid_actions].argmax().item()]
```

- **[매개변수]**
  - obs (현재 상태)
  - epsilon (탐험 확률)
  - vaild_actions (현재 상태에서 가능한 행동들의 리스트)

- **[동작]**
  1. ε의 확률로 무작위 행동 선택
  2. (1 - ε)의 확률로 Q-Value가 최대인 행동 선택

6. Train_dqn function

```python
def train_dqn(n_episodes=N_EPISODES, state_type='count', action_type='job_type', rule_list=None):
    env = JobScheduler(action_type=action_type, rule_list=rule_list, state_type=state_type)
    action_size = len(env.rule_list) if action_type == 'rule' else len(env.processing_times)
    state_size = env.get_state_size()

    q = Qnet(state_size, action_size)
    q_target = Qnet(state_size, action_size)
    q_target.load_state_dict(q.state_dict())
    memory = ReplayBuffer()

    optimizer = optim.Adam(q.parameters(), lr=LEARNING_RATE)

    for episode in range(n_episodes):
        epsilon = max(0.01, 0.08 - 0.01*(episode/200))
        state = env.reset()
        done = False
        total_reward = 0

        while not done:
            valid_actions = env.get_valid_actions()
            action = q.sample_action(torch.FloatTensor(state), epsilon, valid_actions)
            next_state, reward, done = env.step(action)
            done_mask = 0.0 if done else 1.0
            memory.put((state, action, reward/100.0, next_state, done_mask))
            state = next_state
            total_reward += reward

            if memory.size() > 2000:
                s, a, r, s_prime, done_mask = memory.sample(BATCH_SIZE)
                q_out = q(s)
                q_a = q_out.gather(1, a)

                valid_actions = [env.get_valid_actions() for _ in range(BATCH_SIZE)]
                max_q_prime = torch.zeros(BATCH_SIZE, 1)
                q_prime = q_target(s_prime)
                for i, actions in enumerate(valid_actions):
                    if actions:
                        max_q_prime[i] = q_prime[i, actions].max().unsqueeze(0)

                target = r + GAMMA * max_q_prime * done_mask
                loss = F.smooth_l1_loss(q_a, target)

                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

        if episode % 500 == 0 and episode != 0:
            q_target.load_state_dict(q.state_dict())
            print(f"Episode: {episode}, Job Sequence: {env.job_sequence}, Total Tardiness: {env.tardiness}, Epsilon: {epsilon:.2f}")

    return env, q
```

6. Train_dqn function

- **[매개변수]**
  - n_episodes (학습할 에피소드 수)
  - state_type (상태 표현 방식)
  - action_type (행동 표현 방식)
  - rule_list (사용할 rule 목록)

- **[동작]**

1. 환경, Q-network, target Q-network, replay buffer 초기화
2. 지정된 에피소드 수만큼 반복:
   a. 환경을 리셋하고 초기 상태 획득
   b. 에피소드가 종료될 때까지 반복:
      - ε-greedy 정책으로 행동 선택
      - 선택한 행동으로 환경을 한 단계 진행
      - 경험을 replay buffer에 저장
      - replay buffer에서 mini batch를 샘플링하여 Q-network 학습
   c. 일정 주기마다 target Q-network 업데이트 및 학습 진행 상황 출력
3. 학습된 환경과 Q-network를 반환

7. Test_dqn function

```
def test_dqn(env, q):
    state = env.reset()
    done = False
    job_sequence = []

    while not done:
        valid_actions = env.get_valid_actions()
        action = q.sample_action(torch.FloatTensor(state), 0, valid_actions)
        next_state, reward, done = env.step(action)
        job_sequence.append(env.job_sequence[-1] if env.job_sequence else None)
        state = next_state

    print("Final Job Sequence:", job_sequence)
    print("Total Tardiness:", env.tardiness)
```

- **[매개변수]**
  - env (학습된 환경)
  - q (학습된 Q-network)

- **[동작]**

1. 환경을 리셋하고 초기 상태 획득
2. 에피소드가 종료될 때까지 반복:
   - 학습된 Q-network를 사용하여 행동 선택
   - 선택한 행동으로 환경을 한 단계 진행
   - 선택한 작업 기록
3. 테스트 결과 출력 (최종 작업 순서, 총 지연 시간)

8. 실행 결과

```
Training with job type actions and 'count' state definition:
Episode: 500, Job Sequence: ['C', 'C', 'F', 'E', 'F', 'F', 'D', 'B', 'B', 'B', 'A', 'A'], Total Tardiness: 230, Epsilon: 0.06
Episode: 1000, Job Sequence: ['B', 'C', 'C', 'E', 'B', 'B', 'F', 'F', 'F', 'A', 'D', 'A'], Total Tardiness: 195, Epsilon: 0.03
Episode: 1500, Job Sequence: ['B', 'E', 'B', 'B', 'F', 'C', 'C', 'F', 'F', 'D', 'A', 'A'], Total Tardiness: 168, Epsilon: 0.01
Episode: 2000, Job Sequence: ['E', 'B', 'B', 'B', 'D', 'C', 'C', 'F', 'F', 'F', 'A', 'A'], Total Tardiness: 156, Epsilon: 0.01
Episode: 2500, Job Sequence: ['E', 'B', 'B', 'B', 'D', 'C', 'C', 'F', 'F', 'F', 'A', 'A'], Total Tardiness: 156, Epsilon: 0.01
Episode: 3000, Job Sequence: ['E', 'B', 'B', 'B', 'D', 'A', 'F', 'F', 'A', 'F', 'C', 'C'], Total Tardiness: 135, Epsilon: 0.01
Episode: 3500, Job Sequence: ['F', 'E', 'D', 'A', 'B', 'B', 'B', 'F', 'F', 'A', 'C', 'C'], Total Tardiness: 149, Epsilon: 0.01
Episode: 4000, Job Sequence: ['E', 'D', 'A', 'B', 'B', 'B', 'A', 'F', 'F', 'F', 'C', 'C'], Total Tardiness: 128, Epsilon: 0.01
Episode: 4500, Job Sequence: ['E', 'B', 'B', 'B', 'D', 'A', 'A', 'F', 'F', 'F', 'C', 'C'], Total Tardiness: 133, Epsilon: 0.01
Episode: 5000, Job Sequence: ['E', 'D', 'A', 'B', 'B', 'B', 'F', 'F', 'F', 'A', 'C', 'C'], Total Tardiness: 131, Epsilon: 0.01
Episode: 5500, Job Sequence: ['E', 'D', 'A', 'F', 'B', 'B', 'B', 'F', 'A', 'C', 'F', 'C'], Total Tardiness: 143, Epsilon: 0.01
Episode: 6000, Job Sequence: ['E', 'D', 'A', 'B', 'A', 'B', 'F', 'F', 'B', 'C', 'F', 'C'], Total Tardiness: 139, Epsilon: 0.01
Episode: 6500, Job Sequence: ['E', 'D', 'A', 'B', 'B', 'B', 'F', 'A', 'F', 'F', 'C', 'C'], Total Tardiness: 129, Epsilon: 0.01
Final Job Sequence: ['E', 'D', 'B', 'B', 'B', 'A', 'F', 'F', 'F', 'C', 'A', 'C']
Total Tardiness: 135

Training with rule-based actions and 'count' state definition:
Episode: 500, Job Sequence: ['C', 'C', 'E', 'B', 'B', 'F', 'A', 'B', 'D', 'A', 'F', 'F'], Total Tardiness: 193, Epsilon: 0.06
Episode: 1000, Job Sequence: ['C', 'E', 'B', 'C', 'B', 'B', 'D', 'F', 'F', 'A', 'A', 'F'], Total Tardiness: 181, Epsilon: 0.03
Episode: 1500, Job Sequence: ['C', 'A', 'E', 'B', 'B', 'B', 'D', 'A', 'F', 'F', 'F', 'C'], Total Tardiness: 155, Epsilon: 0.01
Episode: 2000, Job Sequence: ['A', 'A', 'D', 'E', 'C', 'B', 'B', 'B', 'F', 'F', 'F', 'C'], Total Tardiness: 147, Epsilon: 0.01
Episode: 2500, Job Sequence: ['A', 'A', 'D', 'E', 'B', 'B', 'B', 'F', 'F', 'F', 'C', 'C'], Total Tardiness: 130, Epsilon: 0.01
Episode: 3000, Job Sequence: ['A', 'A', 'E', 'B', 'B', 'B', 'D', 'F', 'F', 'F', 'C', 'C'], Total Tardiness: 127, Epsilon: 0.01
Episode: 3500, Job Sequence: ['A', 'E', 'A', 'B', 'B', 'B', 'D', 'F', 'F', 'F', 'C', 'C'], Total Tardiness: 129, Epsilon: 0.01
Episode: 4000, Job Sequence: ['E', 'A', 'A', 'B', 'B', 'B', 'D', 'F', 'F', 'F', 'C', 'C'], Total Tardiness: 129, Epsilon: 0.01
Episode: 4500, Job Sequence: ['A', 'A', 'D', 'E', 'B', 'B', 'B', 'F', 'F', 'F', 'C', 'C'], Total Tardiness: 130, Epsilon: 0.01
Episode: 5000, Job Sequence: ['A', 'A', 'E', 'B', 'B', 'B', 'D', 'F', 'F', 'F', 'C', 'C'], Total Tardiness: 127, Epsilon: 0.01
Episode: 5500, Job Sequence: ['A', 'E', 'A', 'B', 'B', 'B', 'D', 'F', 'F', 'F', 'C', 'C'], Total Tardiness: 129, Epsilon: 0.01
Episode: 6000, Job Sequence: ['E', 'A', 'A', 'B', 'B', 'B', 'D', 'F', 'F', 'F', 'C', 'C'], Total Tardiness: 129, Epsilon: 0.01
Episode: 6500, Job Sequence: ['A', 'A', 'E', 'B', 'B', 'B', 'D', 'F', 'F', 'F', 'C', 'C'], Total Tardiness: 127, Epsilon: 0.01
Final Job Sequence: ['A', 'A', 'E', 'B', 'B', 'B', 'D', 'F', 'F', 'F', 'C', 'C']
Total Tardiness: 127
```

8. 실행 결과

```
Training with job type actions and 'progress' state definition:
Episode: 500, Job Sequence: ['F', 'F', 'F', 'C', 'C', 'E', 'D', 'A', 'B', 'B', 'B', 'A'], Total Tardiness: 235, Epsilon: 0.06
Episode: 1000, Job Sequence: ['E', 'F', 'C', 'C', 'F', 'F', 'B', 'B', 'B', 'D', 'A', 'A'], Total Tardiness: 219, Epsilon: 0.03
Episode: 1500, Job Sequence: ['E', 'B', 'B', 'C', 'A', 'C', 'F', 'F', 'F', 'B', 'D', 'A'], Total Tardiness: 186, Epsilon: 0.01
Episode: 2000, Job Sequence: ['E', 'F', 'C', 'C', 'F', 'F', 'B', 'B', 'B', 'D', 'A', 'A'], Total Tardiness: 219, Epsilon: 0.01
Episode: 2500, Job Sequence: ['F', 'B', 'C', 'C', 'F', 'F', 'B', 'B', 'E', 'D', 'A', 'A'], Total Tardiness: 224, Epsilon: 0.01
Episode: 3000, Job Sequence: ['C', 'F', 'F', 'C', 'F', 'B', 'E', 'B', 'B', 'D', 'A', 'A'], Total Tardiness: 232, Epsilon: 0.01
Episode: 3500, Job Sequence: ['E', 'F', 'C', 'C', 'F', 'F', 'B', 'B', 'B', 'D', 'A', 'A'], Total Tardiness: 219, Epsilon: 0.01
Episode: 4000, Job Sequence: ['B', 'B', 'C', 'C', 'F', 'F', 'F', 'E', 'B', 'D', 'A', 'A'], Total Tardiness: 211, Epsilon: 0.01
Episode: 4500, Job Sequence: ['F', 'F', 'C', 'C', 'F', 'B', 'B', 'B', 'E', 'A', 'A', 'D'], Total Tardiness: 236, Epsilon: 0.01
Episode: 5000, Job Sequence: ['E', 'F', 'C', 'C', 'F', 'F', 'B', 'B', 'B', 'D', 'A', 'A'], Total Tardiness: 219, Epsilon: 0.01
Episode: 5500, Job Sequence: ['D', 'C', 'C', 'F', 'F', 'F', 'E', 'B', 'B', 'B', 'A', 'A'], Total Tardiness: 227, Epsilon: 0.01
Episode: 6000, Job Sequence: ['B', 'B', 'C', 'C', 'F', 'F', 'F', 'B', 'D', 'A', 'A', 'E'], Total Tardiness: 220, Epsilon: 0.01
Episode: 6500, Job Sequence: ['E', 'B', 'F', 'F', 'C', 'C', 'F', 'B', 'B', 'D', 'A', 'A'], Total Tardiness: 199, Epsilon: 0.01
Final Job Sequence: ['A', 'A', 'C', 'C', 'F', 'F', 'F', 'E', 'B', 'D', 'B', 'B']
Total Tardiness: 227

Training with rule-based actions and 'progress' state definition:
Episode: 500, Job Sequence: ['A', 'E', 'C', 'B', 'B', 'B', 'D', 'A', 'F', 'F', 'F', 'C'], Total Tardiness: 153, Epsilon: 0.06
Episode: 1000, Job Sequence: ['E', 'B', 'C', 'C', 'B', 'B', 'F', 'A', 'D', 'A', 'F', 'F'], Total Tardiness: 182, Epsilon: 0.03
Episode: 1500, Job Sequence: ['A', 'E', 'C', 'C', 'F', 'F', 'B', 'B', 'F', 'B', 'D', 'A'], Total Tardiness: 207, Epsilon: 0.01
Episode: 2000, Job Sequence: ['C', 'C', 'F', 'F', 'F', 'B', 'E', 'B', 'B', 'D', 'A', 'A'], Total Tardiness: 236, Epsilon: 0.01
Episode: 2500, Job Sequence: ['A', 'E', 'C', 'C', 'F', 'F', 'F', 'B', 'B', 'B', 'D', 'A'], Total Tardiness: 213, Epsilon: 0.01
Episode: 3000, Job Sequence: ['C', 'C', 'F', 'F', 'F', 'B', 'B', 'B', 'E', 'D', 'A', 'A'], Total Tardiness: 238, Epsilon: 0.01
Episode: 3500, Job Sequence: ['A', 'E', 'C', 'C', 'F', 'F', 'F', 'B', 'B', 'B', 'D', 'A'], Total Tardiness: 213, Epsilon: 0.01
Episode: 4000, Job Sequence: ['C', 'C', 'F', 'F', 'F', 'B', 'B', 'B', 'E', 'D', 'A', 'A'], Total Tardiness: 238, Epsilon: 0.01
Episode: 4500, Job Sequence: ['A', 'C', 'C', 'E', 'B', 'B', 'F', 'F', 'F', 'B', 'D', 'A'], Total Tardiness: 202, Epsilon: 0.01
Episode: 5000, Job Sequence: ['E', 'C', 'B', 'B', 'C', 'F', 'F', 'F', 'B', 'D', 'A', 'A'], Total Tardiness: 191, Epsilon: 0.01
Episode: 5500, Job Sequence: ['A', 'E', 'B', 'B', 'B', 'D', 'A', 'F', 'C', 'C', 'F', 'F'], Total Tardiness: 133, Epsilon: 0.01
Episode: 6000, Job Sequence: ['C', 'C', 'F', 'F', 'E', 'F', 'B', 'B', 'B', 'D', 'A', 'A'], Total Tardiness: 231, Epsilon: 0.01
Episode: 6500, Job Sequence: ['E', 'B', 'C', 'C', 'B', 'B', 'F', 'F', 'F', 'D', 'A', 'A'], Total Tardiness: 189, Epsilon: 0.01
Final Job Sequence: ['A', 'C', 'C', 'F', 'F', 'F', 'B', 'B', 'B', 'E', 'D', 'A']
Total Tardiness: 235
```