Master's degree in Control System Engineering

Reinforcement Learning
LAB 2

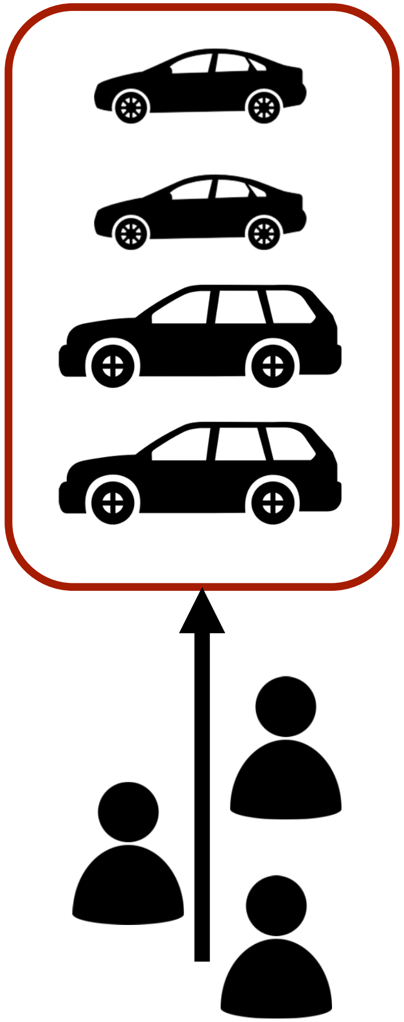
Dynamic Programming
&
car rental problem

Alberto Sinigaglia
(stolen from Niccolò Turcato)
alberto.sinigaglia@phd.unipd.it

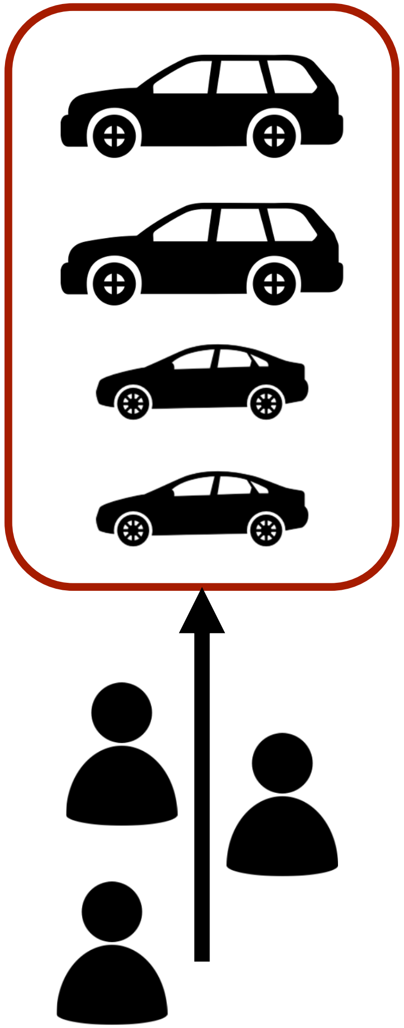
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Jack's Car Rental

2 locations managed by Jack for a nationwide car rental company

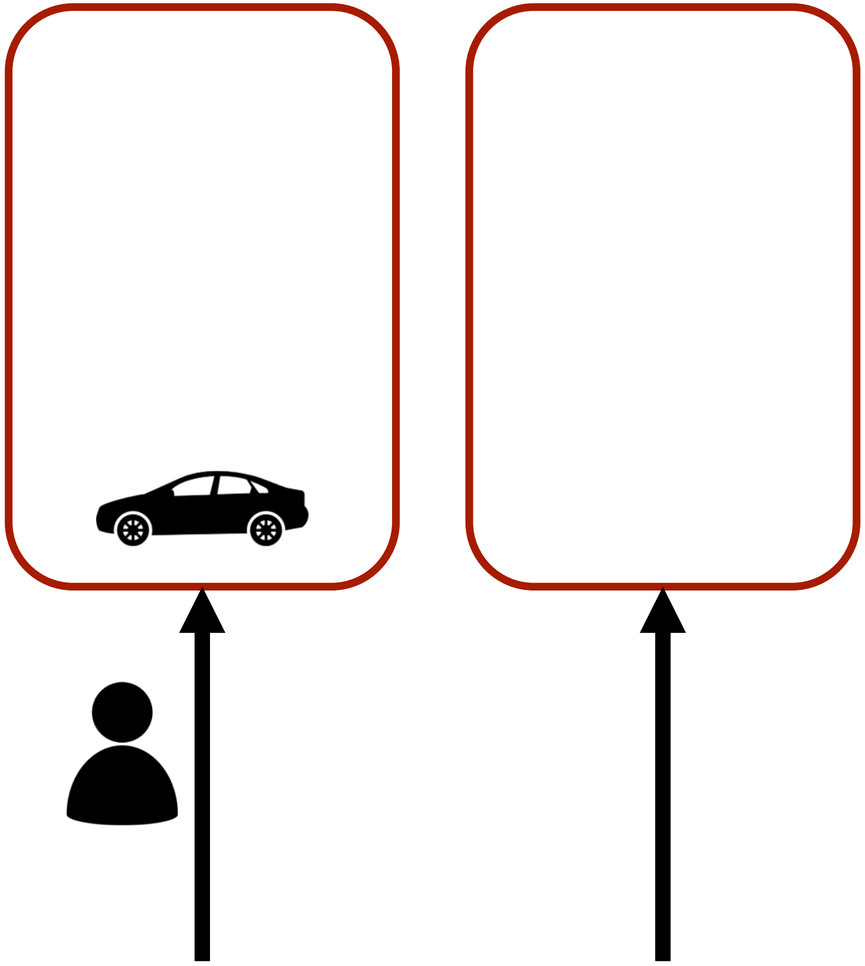


Customers arrive with an unknown distribution

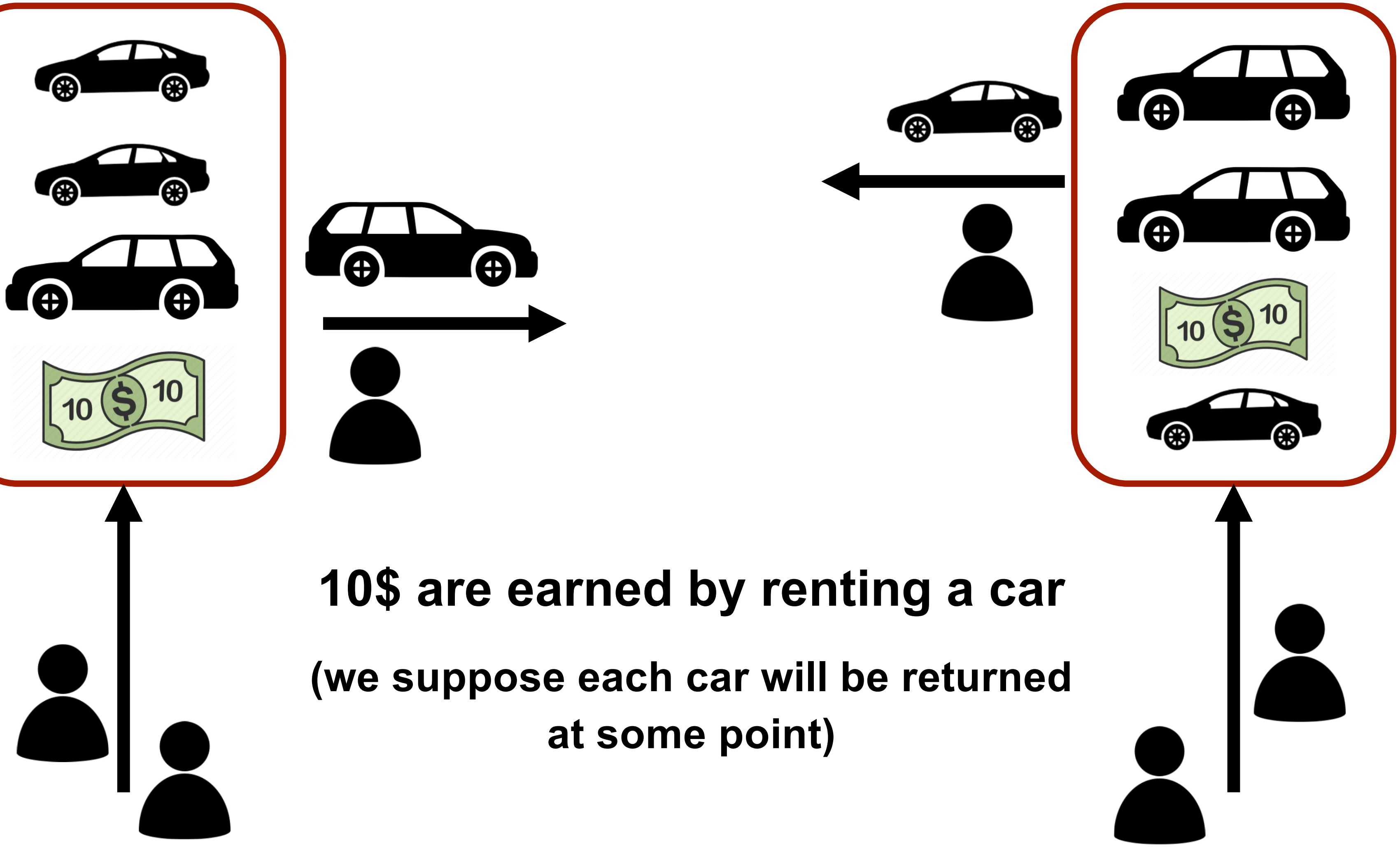


Jack's Car Rental

If there is a car at one location a customer can arrive and rent it.



2 locations managed by Jack for a nationwide car rental company

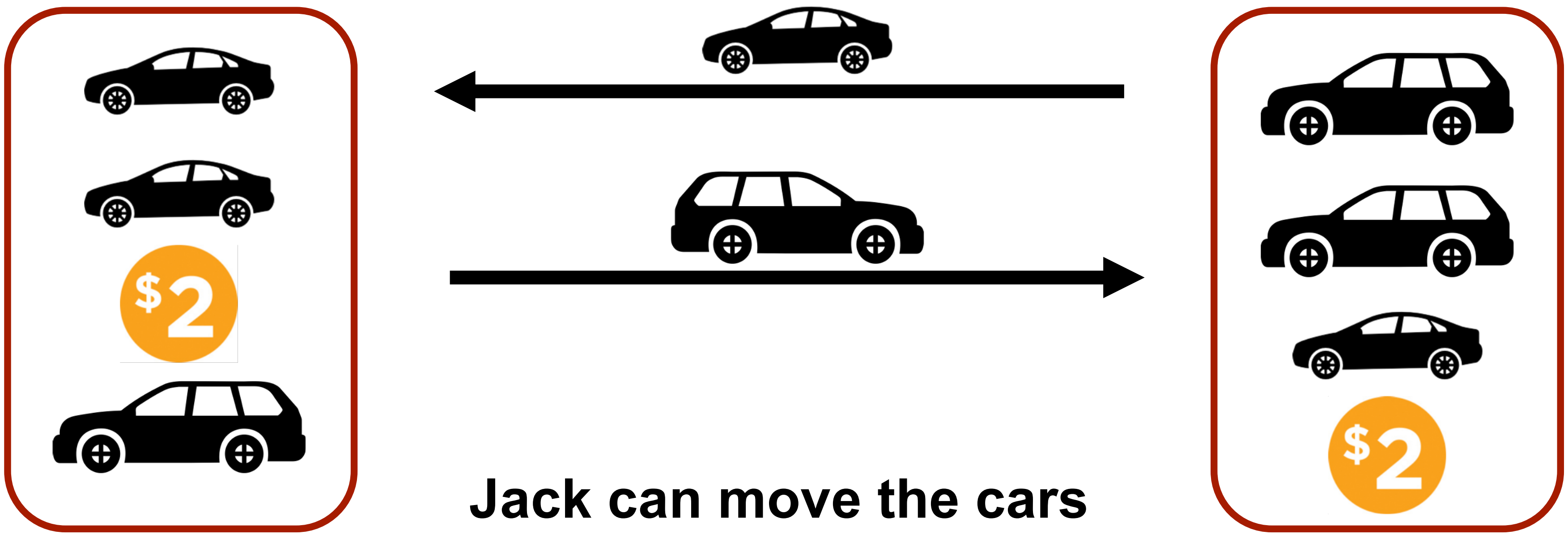


10$ are earned by renting a car

(we suppose each car will be returned at some point)

Jack's Car Rental

2 locations managed by Jack for a nationwide car rental company



Jack can move the cars overnight, paying 2$ for each car

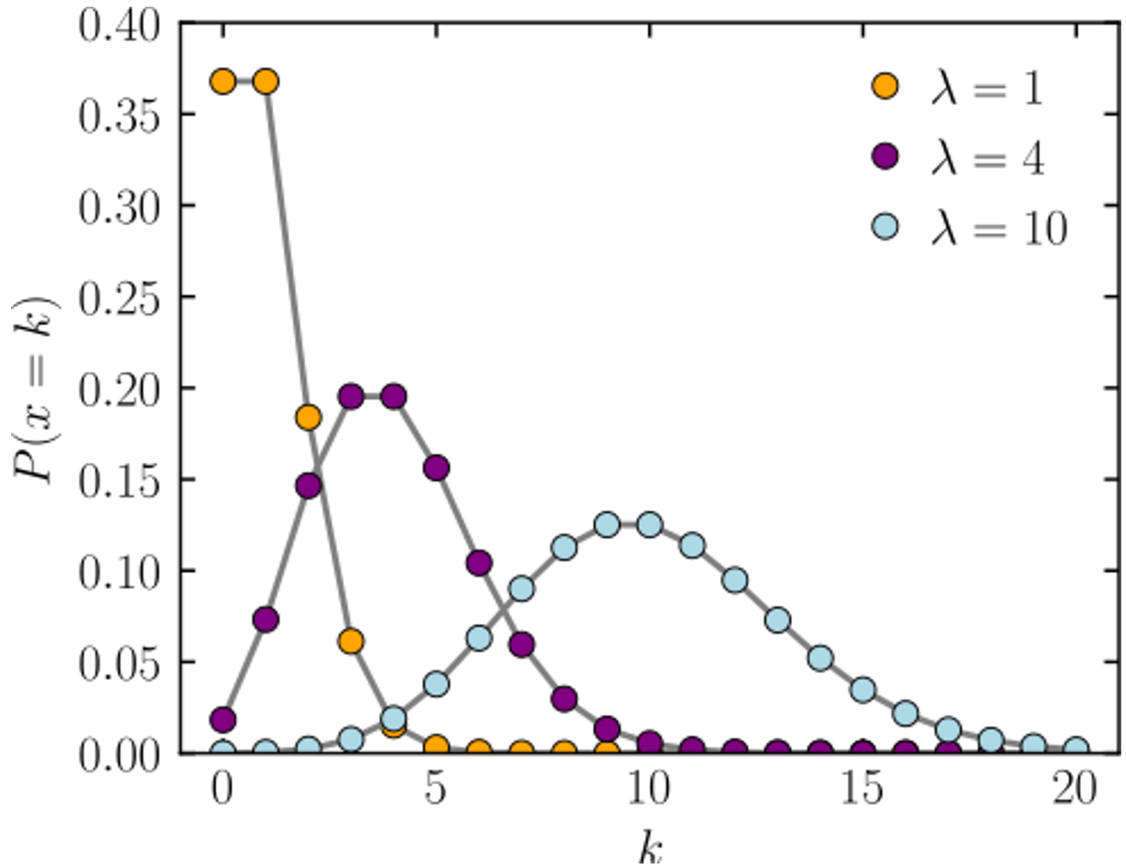
(we'll suppose there's an upper bound to the number of cars he can move)

Jack's Car Rental

Cars requested and returned are modelled either through a Poisson random variable (or through a constant value)

$$\textbf{PMF} = \frac{\lambda^k e^{-\lambda}}{k!}$$

(mean,variance=λ)



Jack's Car Rental

Jack can decide to move cars at the end of the working day!

$$a \in [-Max_M, Max_M]$$

```
actions = np.arange(-MAX_MOVE_OF_CARS, MAX_MOVE_OF_CARS + 1)
```

If a car is available and a customer arrives, he will rent the car!

(no questions asked)

Jack's Car Rental

The state of the environment is the pair of numer:

```
# [# of cars in first location, # of cars in second location]
```

$$s \in \mathbb{R}^2$$

The Value function of the environment is a Matrix!

$$V(s) \in Mat[\mathbb{R}]$$

Code for policy iteration algorithm

```
# Assumption
constant_returned_cars = True

# Initialization of the value-function
value = np.zeros((MAX_CARS + 1, MAX_CARS + 1))

# We start considering the simplest policy: for every possible state, no car is moved
# It makes sense: if is not clear what a meaningful action might be, better not to pay the
cost of moving cars!
policy = np.zeros(value.shape, dtype=np.int)

...
```

Code for policy iteration algorithm

```
...

while True:
    # policy evaluation (in-place)
    while True:
        old_value = value.copy()
        # Sweep through all states following the same policy
        for i in range(MAX_CARS + 1):
            for j in range(MAX_CARS + 1):
                new_state_value = expected_return([i, j], policy[i, j], value)
                # in-place update!
                value[i, j] = new_state_value
        max_value_change = abs(old_value - value).max()

        if max_value_change < eps_val:
            break

    ...
```

Code for policy iteration algorithm

```
...

# policy improvement
policy_stable = True
for i in range(MAX_CARS + 1):
    for j in range(MAX_CARS + 1):
        old_action = policy[i, j]
        action_returns = []
        for action in actions:
            # if it is a 'legal' action,
            if (0 <= action <= i) or (-j <= action <= 0):
                action_returns.append(expected_return([i, j], action, value))
            else:
                action_returns.append(-np.inf)
        # Substitution with greedy policy at all states
        new_action = actions[np.argmax(action_returns)]
        policy[i, j] = new_action
        if policy_stable and old_action != new_action:
            policy_stable = False

# If stable instead
if policy_stable:
    break
```