Let us define with clear mathematical notation the problem here.

The state space consists of tuples of coordinates on the grid where the coordinates correspond to a white square on the grid, and are within the grid limits.

$$\mathbb{S} = \{(x, y) | x, y \in \mathbb{N}, x, y \in \texttt{grid limits}, (x, y) \texttt{ correspond to a white space}\}$$

In the case of the example shown in the subject, we should have: $0 \leq x, y \leq 7$, and for instance $(0, 0)$ is in the state space because it is associated to a white square but $(0, 1)$ is not because there is a black square there.

All the states are non terminating, except for the goal square at the bottom right corner.

The action space is:

$$\mathbb{A} = \{\texttt{Up,Down,Left,Right}\}$$

Yet depending on the position, as you cannot move through blocks, only one, two or three of the actions may be available. If you're in the terminating state, then there are no actions to take.

For instance if we refer to the grid maze picture in the subject, at position $(0, 0)$ only the action $\{\texttt{Down}\}$ is available.

Let us now write the state transition probability function.

We will imagine a position (x, y) such that the four actions $\{\texttt{Up,Down,Left,Right}\}$ are available. Note once again that in some cases, depending on your position (x, y), you may not be able to do one of the actions in the action set.

As a given state is defined by the tuple of coordinates (x, y), we get:

$$P((x, y), \texttt{Right}, (x + 1, y)) = 1$$

In this case, if in the new state (x', y') if $x' \neq x + 1$ or $y' \neq y$, then $\mathbb{P}((x, y), \texttt{Right}, (x', y')) = 0$

Similarly:

$$P((x, y), \texttt{Left}, (x - 1, y)) = 1$$

In this case, if in the new state (x', y') if $x' \neq x - 1$ or $y' \neq y$, then $\mathbb{P}((x, y), \texttt{Left}, (x', y')) = 0$. This is true provided that (x, y) is a state where the action $\{\texttt{Left}\}$ is available. If it is not the case, then it makes no sense to write the state transition probability function from state (x, y) with action $\{\texttt{Left}\}$.

Also:

$$P((x, y), \texttt{Up}, (x, y - 1)) = 1$$

In this case, if in the new state (x', y') if $x' \neq x$ or $y' \neq y - 1$, then $\mathbb{P}((x, y), \texttt{Up}, (x', y')) = 0$

Finally:

$$P((x, y), \texttt{Down}, (x, y + 1)) = 1$$

In this case, if in the new state (x', y') if $x' \neq x$ or $y' \neq y + 1$, then $\mathbb{P}((x, y), \texttt{Down}, (x', y')) = 0$

Note that if you are in a state (x, y), the only states that are at reach (provided that they are not blocks) are the states: $(x+1, y), (x-1, y), (x, y+1), (x, y-1)$

Let us now write two formulations of the reward transition function R_T.

In the first formulation, we will consider a reward only for transitioning into the goal state. In this setting we let a discount factor $\gamma < 1$. This gives us:

$\forall (s, a)$: $R_T(s, a, s') = 0$ if $s' = (x', y')$ is not the terminating goal state.

And $R_T(s, a, s') = 1$ if s' is the terminating state (and if the action a can lead us to transition from s to s').

In the example, the only situation in which the reward can be 1 is: $R_T((6,7), \texttt{Down}, (7,7)) = 1$.

In the second formulation, we are going to give a reward for each transition with a discount factor of $\gamma = 1$. Here the reward will be negative and equal to -1 if we transition to a non terminating state and we will give a big reward of 100 for arriving in the goal state.

$\forall (s, a)$: $R_T(s, a, s') = -1$ if $s' = (x', y')$ is not the terminating state.

And $R_T(s, a, s') = 100$ if s' is the terminating state (and obviously if the action a can lead us to transition from s to s').

To model the MDP and solve the Optimal Value Function and Optimal Policy, I used some libraries and classes from the git repo we cloned at the beginning of the course. My Python code is the following:

```python
from dataclasses import dataclass
from typing import  Tuple,Mapping
from rl.distribution import Constant
from rl.markov_decision_process import (FiniteMarkovDecisionProcess,
                                         StateActionMapping,
                                         ActionMapping,FinitePolicy)
from rl.dynamic_programming import (value_iteration_result,value_iteration,
                                    greedy_policy_from_vf)
from grid_maze import maze_grid,SPACE,BLOCK,GOAL
from rl.markov_process import StateReward
import time

@dataclass(frozen=True)
class Coordinate:
    #This is our state class that we call Coordinate
    x: int
    y: int

@dataclass(frozen=True)
class Action:
    #This is our action class
    action: str

#We are going to model both formulations using instances
#of FiniteMarkovDecisionProcess
class MazeModel1(FiniteMarkovDecisionProcess[Coordinate,Action]):
    #In this first model, we only give reward upon arrival and gamma<1
    def __init__(
        self,
        grid:dict):

        super().__init__(self.get_maze_mapping(grid))

    def get_maze_mapping(self,grid) -> StateActionMapping[Coordinate, Action]:
        mapping: StateActionMapping[Coordinate, Action] = {}
        #To define the StateActionMapping, we need to parse the grid
        #And see which actions are available from each space depending
        #On the surrounding blocks and spaces
        for i in grid.keys():
            x,y = i
            if grid[i] == GOAL:
                mapping[Coordinate(x,y)] = None
            elif grid[i] == SPACE:
                #If we are not in the goal, then we should take actions
                submapping: ActionMapping[Action,StateReward[Coordinate]] = {}

                if grid.get((x+1,y)) == SPACE:
                    submapping[Action('RIGHT')] = Constant((Coordinate(x+1,y),0))
                elif grid.get((x+1,y)) == GOAL:
                    submapping[Action('RIGHT')] = Constant((Coordinate(x+1,y),1))

                if grid.get((x-1,y)) == SPACE:
                    submapping[Action('LEFT')] = Constant((Coordinate(x-1,y),0))
                elif grid.get((x-1,y)) == GOAL:
                    submapping[Action('LEFT')] = Constant((Coordinate(x-1,y),1))

                if grid.get((x,y+1)) == SPACE:
                    submapping[Action('DOWN')] = Constant((Coordinate(x,y+1),0))
                elif grid.get((x,y+1)) == GOAL:
                    submapping[Action('DOWN')] = Constant((Coordinate(x,y+1),1))

                if grid.get((x,y-1)) == SPACE:
                    submapping[Action('UP')] = Constant((Coordinate(x,y-1),0))
                elif grid.get((x,y-1)) == GOAL:
                    submapping[Action('UP')] = Constant((Coordinate(x,y-1),1))
                mapping[Coordinate(x,y)] = submapping
        return mapping

class MazeModel2(FiniteMarkovDecisionProcess[Coordinate,Action]):
    #In this second formulation, we give rewards (positive or negative)
    #at each step and gamma = 1
    def __init__(
        self,
        grid:dict):
        super().__init__(self.get_maze_mapping(grid))

    def get_maze_mapping(self,grid) -> StateActionMapping[Coordinate, Action]:
        mapping: StateActionMapping[Coordinate, Action] = {}
        for i in grid.keys():
            x,y = i
            if grid[i] == GOAL:
                mapping[Coordinate(x,y)] = None
            elif grid[i] == SPACE:
                submapping: ActionMapping[Action,StateReward[Coordinate]] = {}

                if grid.get((x+1,y)) == SPACE:
                    submapping[Action('RIGHT')] = Constant((Coordinate(x+1,y),-1))
                elif grid.get((x+1,y)) == GOAL:
                    submapping[Action('RIGHT')] = Constant((Coordinate(x+1,y),100))

                if grid.get((x-1,y)) == SPACE:
                    submapping[Action('LEFT')] = Constant((Coordinate(x-1,y),-1))
                elif grid.get((x-1,y)) == GOAL:
                    submapping[Action('LEFT')] = Constant((Coordinate(x-1,y),100))

                if grid.get((x,y+1)) == SPACE:
                    submapping[Action('DOWN')] = Constant((Coordinate(x,y+1),-1))
                elif grid.get((x,y+1)) == GOAL:
                    submapping[Action('DOWN')] = Constant((Coordinate(x,y+1),100))

                if grid.get((x,y-1)) == SPACE:
                    submapping[Action('UP')] = Constant((Coordinate(x,y-1),-1))
                elif grid.get((x,y-1)) == GOAL:
                    submapping[Action('UP')] = Constant((Coordinate(x,y-1),100))
                mapping[Coordinate(x,y)] = submapping
        return mapping

 #Below we have made a helper function to find the Optimal Value Function
 #We're using the value_iteration method of rl.dynamic_programming
 TOLERANCE = 1e-5
 def solution(model: FiniteMarkovDecisionProcess[Coordinate,Action],
                  gamma: float
                  )->Tuple[int,Mapping[Coordinate, float],
                           FinitePolicy[Coordinate,Action]]:
        count = 0
        v = value_iteration(model, gamma)
        a = next(v, None)
        while True:
            if a is None:
                break
            b = next(v,None)
            if max(abs(a[s] - b[s]) for s in a) < TOLERANCE:
                break
            a = b
            count+=1

        opt_policy :FinitePolicy[Coordinate,Action] = greedy_policy_from_vf(
            model,
            b,
            gamma
        )
        return count,b,opt_policy

if __name__ == '__main__':
    model1 = MazeModel1(maze_grid)
    model2 = MazeModel2(maze_grid)
    start = time.time()
    count1,opt_vf1,opt_pol1 = solution(model1,0.8)
    print(f"Method␣1␣took␣{time.time()-start}␣to␣converge")
    start = time.time()
    count2,opt_vf2,opt_pol2 = solution(model2,1)
    print(f"Method␣2␣took␣{time.time()-start}␣to␣converge")
    print(f"Solution␣1␣took␣{count1}␣iterations␣to␣converge")
    print(f"Solution␣2␣took␣{count2}␣iterations␣to␣converge")
    print(opt_pol1)
    print(opt_pol2)

    #We also propose another solution where we don't track
    #the number of iterations to converge
    #This solution is lighter in code
    #We're using the built-in function of rl.dynamic_programming
    #value_iteration_result here
    start = time.time()
    opt_vf1,opt_pol1 = value_iteration_result(model1,0.8)
    print(f"Method␣1␣took␣{time.time()-start}␣to␣converge")
    start = time.time()
    opt_vf2,opt_pol2 = value_iteration_result(model2,1)
    print(f"Method␣2␣took␣{time.time()-start}␣to␣converge")
```

The output we got (when not printing the optimal policy) is:

```
Method 1 took 0.007191181182861328 to converge
Method 2 took 0.0073931217193603516 to converge
Solution 1 took 16 iterations to converge
Solution 2 took 16 iterations to converge
#In the method where we don't track the number of solutions
Method 1 took 0.007776021957397461 to converge
Method 2 took 0.007478237152099609 to converge
```

The optimal policy we got in the first method, where we only give a reward at the end of the maze, with $\gamma < 1$ (we chose $\gamma = 0.8$ here) is:

```
For State Coordinate(x=0, y=0):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=0, y=2):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=0, y=3):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=0, y=4):
  Do Action Action(action='UP') with Probability 1.000
For State Coordinate(x=0, y=5):
  Do Action Action(action='UP') with Probability 1.000
For State Coordinate(x=0, y=6):
  Do Action Action(action='UP') with Probability 1.000
For State Coordinate(x=0, y=7):
  Do Action Action(action='UP') with Probability 1.000
For State Coordinate(x=1, y=0):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=1, y=3):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=2, y=0):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=2, y=2):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=2, y=3):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=2, y=4):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=2, y=5):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=2, y=7):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=3, y=0):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=3, y=1):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=3, y=2):
  Do Action Action(action='LEFT') with Probability 1.000
For State Coordinate(x=3, y=5):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=3, y=7):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=4, y=0):
  Do Action Action(action='LEFT') with Probability 1.000
For State Coordinate(x=4, y=2):
  Do Action Action(action='LEFT') with Probability 1.000
For State Coordinate(x=4, y=4):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=4, y=5):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=4, y=6):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=4, y=7):
  Do Action Action(action='UP') with Probability 1.000
For State Coordinate(x=5, y=2):
  Do Action Action(action='LEFT') with Probability 1.000
For State Coordinate(x=5, y=4):
  Do Action Action(action='LEFT') with Probability 1.000
For State Coordinate(x=5, y=6):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=6, y=0):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=6, y=4):
  Do Action Action(action='LEFT') with Probability 1.000
For State Coordinate(x=6, y=6):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=6, y=7):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=7, y=0):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=7, y=1):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=7, y=2):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=7, y=3):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=7, y=4):
  Do Action Action(action='LEFT') with Probability 1.000
```

In the second method where we give a negative reward at each state except at the goal state where the reward is big and where $\gamma = 1$, we get:

```
For State Coordinate(x=0, y=0):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=0, y=2):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=0, y=3):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=0, y=4):
  Do Action Action(action='UP') with Probability 1.000
For State Coordinate(x=0, y=5):
  Do Action Action(action='UP') with Probability 1.000
For State Coordinate(x=0, y=6):
  Do Action Action(action='UP') with Probability 1.000
For State Coordinate(x=0, y=7):
  Do Action Action(action='UP') with Probability 1.000
For State Coordinate(x=1, y=0):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=1, y=3):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=2, y=0):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=2, y=2):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=2, y=3):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=2, y=4):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=2, y=5):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=2, y=7):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=3, y=0):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=3, y=1):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=3, y=2):
  Do Action Action(action='LEFT') with Probability 1.000
For State Coordinate(x=3, y=5):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=3, y=7):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=4, y=0):
  Do Action Action(action='LEFT') with Probability 1.000
For State Coordinate(x=4, y=2):
  Do Action Action(action='LEFT') with Probability 1.000
For State Coordinate(x=4, y=4):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=4, y=5):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=4, y=6):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=4, y=7):
  Do Action Action(action='UP') with Probability 1.000
For State Coordinate(x=5, y=2):
  Do Action Action(action='LEFT') with Probability 1.000
For State Coordinate(x=5, y=4):
  Do Action Action(action='LEFT') with Probability 1.000
For State Coordinate(x=5, y=6):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=6, y=0):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=6, y=4):
  Do Action Action(action='LEFT') with Probability 1.000
For State Coordinate(x=6, y=6):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=6, y=7):
  Do Action Action(action='RIGHT') with Probability 1.000
For State Coordinate(x=7, y=0):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=7, y=1):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=7, y=2):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=7, y=3):
  Do Action Action(action='DOWN') with Probability 1.000
For State Coordinate(x=7, y=4):
  Do Action Action(action='LEFT') with Probability 1.000
```

We notice that the optimal policies we derived for both our implementations match. Besides, we observe that both methods took the same number of iterations (16) to converge as well as a similar amount of time.

Also note that given the different rewards involved in the two implementations, it is normal that we obtained different value functions. The scale of the value functions we obtained differ sensibly from one implementation to the other; this is mainly because in one case we give in the end state a reward of 1, while in the other case we chose to give a reward of 100.

Let us find conditions on the features matrix ϕ such that we can exactly represent the true value function.

We wish to approximate the value function of the given MRP with a linear function based on m features that are specified as a $n \times m$ matrix ϕ. This means that we want to approximate v with v' such that:

$v' = \phi X$ where X is a $m \times 1$ column vector. It's a weight vector that we could typically learn from the data by minimizing the cross entropy loss function for data at our disposal.

The MSE of linear predictions using the feature matrix would be without any regularization:

$$||v - v'||^2 = ||v - \phi X||^2$$

In a Markov Reward Process, the Bellman equation gives us that, for all $s \in N$, and thus in this case with no terminal states, for all $s \in S$:

$$V(s) = R(s) + \gamma \times \sum_{s' \in S} P(s, s') \times V(s')$$

In vector notation, we have:

$$\mathbf{v} = \mathbf{R} + \gamma \mathbf{P} \times \mathbf{v}$$

This means that:

$$(\mathbf{I_n} - \gamma \mathbf{P})\mathbf{v} = \mathbf{R}$$

where $\mathbf{I_n}$ is a $n \times n$ identity matrix.

If we impose that the matrix $(\mathbf{I_n} - \gamma \mathbf{P})$ is invertible (we'll need $det(\mathbf{I_n} - \gamma \mathbf{P}) \neq 0$ for that), then we can have:

$$\mathbf{v} = (\mathbf{I_n} - \gamma \mathbf{P})^{-1} \mathbf{R}$$

This corresponds to a situation in which we can exactly represent the true value function using a $n \times n$ matrix which we will call a feature matrix equal to $\phi = (\mathbf{I_n} - \gamma \mathbf{P})^{-1}$. By multiplying this feature matrix with the reward function, we can exactly get the true value function, which is what we wanted in the beginning.

And, we get back to the MSE notation, we have that, if we use R as weights, according to what we derived with the Bellman Equation:

$$||v - \phi X||^2 = ||v - (\mathbf{I_n} - \gamma \mathbf{P})^{-1} \mathbf{R}||^2 = 0$$

Note that we need the feature matrix to be a $n \times n$ matrix, which needs that we must have $m = n$ for the feature matrix.

In conclusion, the conditions to be able to represent exactly the true value function with a feature matrix ϕ is that the matrix $(\mathbf{I_n} - \gamma \mathbf{P})$ should be invertible. This is equivalent to saying that we should have $det(\mathbf{I_n} - \gamma \mathbf{P}) \neq 0$.

Let us model this as a Finite MDP with proper mathematical notation.

The state space S consists of all the possible wages: $S = \{1, 2, ..., W\}$.

The actions consist of the tuple (l, s). Each day we need to choose the number of hours l spent on learning to get better at our current job as well as the number of hours s spent on searching for another job. From this, we can directly get the number of hours actually working on our current job: $H - l - s$.

Hence: $A = \{(l, s) | l \in \mathbb{Z}_+; s \in \mathbb{Z}_+; 0 \leq l + s \leq H\}$

Let us now specify the reward transition function in this case.

The expression of the reward transition function is quite straightforward here. Indeed, $\forall (l, s)$:

$$R_T(w, (l, s), w') = w(H - l - s)$$

where w is the start state and w' is the next state (states are wages here). The reward we get at a given step is purely deterministic. it only depends on the actions we took during that step and it is independent from the next state we'll reach.

Let us now specify the state transitions.

If you are in a state in which you receive a wage w, and you take action (l, s), then, you'll receive a proposal of hourly wage of $min(w + x, W)$ by your employer and with probability $\beta s/H$, you'll receive a proposal for an hourly wage of $min(w + 1, W)$ by another employer.

With probability at least $1 - \beta s/H$, you can be in a situation in which you don' t receive another job offer, you arrive in a state $min(w + x, W)$, where x is a Poisson random variable with mean $\alpha.l$ for a given action.

Mathematically speaking: $x \sim \texttt{Poisson}(\alpha l)$

To write the probability transition function, we'll have to distinguish cases based on the value of w with respect to W since the next state distribution involves some minimums.

First note that we cannot transition to a state with an inferior wage: $\forall (l, s)$, if $w' < w$

$$P(w, (l, s), w') = 0$$

Yet the salary can stay the same.

If $w \neq W$, then, $\forall (l, s)$:

$$P(w, (l, s), w) = (1 - \frac{\beta s}{H})\mathbb{P}(x = 0) = (1 - \frac{\beta s}{H})e^{-\alpha l}$$

This happens in cases where we don't receive any other job offer and our employer does not increase the salary $(x = 0)$.

In the case where $w = W$, then:

$\forall (l, s)$: $P(W, (l, s), W) = 1$. Your wage cannot increase or decrease from this point.

There are also some situations in which the fact that you receive another job offer will not matter. This happens in cases where x is high enough.

If $x = k$ and if $W > w + k > w + 1$, hence if $W - w > k > 1$, then,

$$\forall (l, s) : P(w, (l, s), w + k) = \mathbb{P}(x = k) = e^{-\alpha l} \frac{(\alpha l)^k}{k!}$$

Indeed, regardless of the offer you'll receive from another employer, if your employer proposes you an important salary, then you'll accept it anyway.

If the offer is high enough, we may reach state W directly from the employer offer. If $w \leq W - 2$:

$$\forall (l, s) : P(w, (l, s), W) = \mathbb{P}(x \geq W - w) = 1 - \mathbb{P}(x < W - w) = 1 - \sum_{i=0}^{W-w-1} e^{-\alpha l} \frac{(\alpha l)^i}{i!}$$

Once again, in this case, the fact that you receive another job offer or no does not matter: you'll accept your employer's offer anyway.

If $w \leq W - 2$ again, we can be in a situation where we accept the new employer's offer with some probability. In this case the next state is $w + 1$. The probability for this is:

$$\forall (l, s) : P(w, (l, s), w + 1) = \frac{\beta s}{H} \times \mathbb{P}(x \leq 1) + (1 - \frac{\beta s}{H})\mathbb{P}(x = 1)$$

To receive the wage immediately superior, you must either receive an offer by another employer and this offer must be superior to the offer from your employer or you can be in a situation where you do not receive an offer and your employer must make you the offer $w + 1$.

If $w = W - 1$, to get to state W, then either you receive an offer from your employer that is greater than one or an offer from another employer:

$$\forall (l, s) : P(W - 1, (l, s), W) = \frac{\beta s}{H} + (1 - \frac{\beta s}{H})\mathbb{P}(x \geq 1) = \frac{\beta s}{H} + (1 - \frac{\beta s}{H})(1 - \mathbb{P}(x = 0))$$

Let us now implement this MDP in Python. We used, like in problem 1, the code in the git repo that we forked at the start of the course.

We thus implemented the MDP using an instance of `FiniteMarkovDecisionProcess`.

To solve for the Optimal Value Function and Optimal Policy, we used the method `value_iteration_result` in the `rl/dynamic_programming.py` file.

We used in our implementation the parameters suggested by the subject: $H = 10, W = 30, \alpha = 0.08, \beta = 0.82, \gamma = 0.95$

My code for this has been:

```
from dataclasses import dataclass
from typing import  List
from rl.distribution import Constant,Categorical
from rl.markov_decision_process import (FiniteMarkovDecisionProcess,
                                        StateActionMapping,
                                        ActionMapping)
from rl.dynamic_programming import (value_iteration_result)
from rl.markov_process import StateReward
from scipy.stats import poisson

@dataclass(frozen=True)
class State:
    #This is our state class that we call State.
    #A state is represented by a wage
    wage:int


@dataclass(frozen=True)
class Action:
    #An action is defined by a Tuple(l,s)
    l: int
    s: int

class Problem3(FiniteMarkovDecisionProcess[State,Action]):
    def __init__(
        self,
        H:int,
        W:int,
        alpha: float,
        beta: float):

        self.H:int = H
        self.W:int = W
        self.alpha:float = alpha
        self.beta:float = beta

        super().__init__(self.get_mapping())
    def get_mapping(self) -> StateActionMapping[State, Action]:
        #We need to define the StateActionMapping for this Finite MDP
        mapping: StateActionMapping[State, Action] = {}
        list_actions:List[Action] = []
        #We start by defining all the available actions
        for i in range(self.H+1):
            range_j = self.H-i
            for j in range(range_j+1):
                list_actions.append(Action(i,j))
        self.list_actions:List[Action] = list_actions
        list_states:List[State] = []
        #Then we define all the possible states
        for i in range(1,self.W+1):
            list_states.append(State(i))
        self.list_states:List[State] = list_states
        for state in list_states:
            submapping:ActionMapping[Action,StateReward[State]] = {}
            for action in list_actions:
                s:int = action.s
                l:int = action.l
                reward:float = state.wage*(self.H-l-s)
                pois_mean:float = self.alpha*l
                proba_offer:float = self.beta*s/self.H
                if state.wage == self.W:
                    #If you're in state W, you stay in state W with constant
                    #Probability. The reward only depends on the action you
                    #you have chosen
                    submapping[action] = Constant((state,reward))
                elif state.wage == self.W-1:
                    #If you're in state W-1, you can either stay in your state
                    #or land in state W
                    submapping[action] = Categorical({
                        (state,
                         reward):
                            poisson.pmf(0,pois_mean)*(1-proba_offer),
                         (State(self.W),
                          reward):proba_offer+(1-proba_offer)*\
                             (1-poisson.pmf(0,pois_mean))
                        })
                else:
                    #If you're in any other state, you can land to any state
                    #Between your current state and W with probabilities
                    #as described before
                    dic_distrib = {}
                    dic_distrib[
                        (state,
                        reward)] = poisson.pmf(0,pois_mean)*(1-proba_offer)
                    dic_distrib[
                        (State(state.wage+1),
                         reward)] = proba_offer*poisson.cdf(1,pois_mean)\
                                +(1-proba_offer)*poisson.pmf(1,pois_mean)
                    for k in range(2,self.W-state.wage):
                        dic_distrib[
                        (State(state.wage+k),
                         reward)] = poisson.pmf(k,pois_mean)
                    dic_distrib[
                        (State(self.W),
                         reward)] = 1-poisson.cdf(self.W-state.wage-1,pois_mean)
                    submapping[action] = Categorical(dic_distrib)
            mapping[state] = submapping
        return mapping


if __name__ == '__main__':
    H = 10
    W = 30
    alpha = 0.08
    beta = 0.82
    gamma = 0.95
    print("Defining␣the␣model")
    model =  Problem3(H,W,alpha,beta)
    print("Value␣iteration␣algorithm")
    print("We're␣using␣a␣built-in␣function␣in␣the␣dynamic_programming␣file")
    opt_val, opt_pol = value_iteration_result(model,gamma)
    print(opt_pol)
```

This gave us the following Optimal Policy:

```
For State State(wage=1):
  Do Action Action(l=10, s=0) with Probability 1.000
For State State(wage=2):
  Do Action Action(l=10, s=0) with Probability 1.000
For State State(wage=3):
  Do Action Action(l=10, s=0) with Probability 1.000
For State State(wage=4):
  Do Action Action(l=10, s=0) with Probability 1.000
For State State(wage=5):
  Do Action Action(l=10, s=0) with Probability 1.000
For State State(wage=6):
  Do Action Action(l=10, s=0) with Probability 1.000
For State State(wage=7):
  Do Action Action(l=10, s=0) with Probability 1.000
For State State(wage=8):
  Do Action Action(l=10, s=0) with Probability 1.000
For State State(wage=9):
  Do Action Action(l=10, s=0) with Probability 1.000
For State State(wage=10):
  Do Action Action(l=10, s=0) with Probability 1.000
For State State(wage=11):
  Do Action Action(l=10, s=0) with Probability 1.000
For State State(wage=12):
  Do Action Action(l=10, s=0) with Probability 1.000
For State State(wage=13):
  Do Action Action(l=10, s=0) with Probability 1.000
For State State(wage=14):
  Do Action Action(l=0, s=10) with Probability 1.000
For State State(wage=15):
  Do Action Action(l=0, s=10) with Probability 1.000
For State State(wage=16):
  Do Action Action(l=0, s=0) with Probability 1.000
For State State(wage=17):
  Do Action Action(l=0, s=0) with Probability 1.000
For State State(wage=18):
  Do Action Action(l=0, s=0) with Probability 1.000
For State State(wage=19):
  Do Action Action(l=0, s=0) with Probability 1.000
For State State(wage=20):
  Do Action Action(l=0, s=0) with Probability 1.000
For State State(wage=21):
  Do Action Action(l=0, s=0) with Probability 1.000
For State State(wage=22):
  Do Action Action(l=0, s=0) with Probability 1.000
For State State(wage=23):
  Do Action Action(l=0, s=0) with Probability 1.000
For State State(wage=24):
  Do Action Action(l=0, s=0) with Probability 1.000
For State State(wage=25):
  Do Action Action(l=0, s=0) with Probability 1.000
For State State(wage=26):
  Do Action Action(l=0, s=0) with Probability 1.000
For State State(wage=27):
  Do Action Action(l=0, s=0) with Probability 1.000
For State State(wage=28):
  Do Action Action(l=0, s=0) with Probability 1.000
For State State(wage=29):
  Do Action Action(l=0, s=0) with Probability 1.000
For State State(wage=30):
  Do Action Action(l=0, s=0) with Probability 1.000
```

The Optimal Policy is quite intuitive. When the wage is really low, we're better off learning to get better at our current job, even if it implies not gaining any reward.

At some point when the salary is quite important (wage $= 14$ or wage $= 15$ here), it's also a good idea to spend time searching for another job that can pay better, even if it implies not gaining any reward during some days.

When the salary begins to be important (wage ≥ 16), then it's not worth anymore to waste working hours during which we could earn rewards and try to get better at our current job or search for another job. Our salary is already high enough. The efforts spent to potentially get higher rewards later are not worth the loss of immediate reward they incur.