Pseudo code :

```
      Function Environment(array, policy):
            declare result:- /to be used as an array to contain the route
            to any column in the column list do:
            If row equals column equals 1,
            then: = "wall" value else , if row is less than or equal to 1
            and column is more than or equal to 5,
            then: - / defining the limits value:- = +1
            when row is 0, else -1 value:- = the current grid cell and the
            action value + "|" = result outcome:- = outcome + "n"
            return result

      Function Evaluation(policy, util):-
      while True then:-
            Declare nextUtil and error:-
            for row in NumRows do:-
                  For column in Num_cols do:-
                        if row:- less than 1 and column equal 3 or row
equal column equal 1 then:-
                              continue //Skip the current iteration
                        nextUtiliy[row][col] = clavulateUtil(Util, row,
columns, policy)
                        error:- = max(error, absolute value of next util
current index
            Set Util to nextUtil
            If error is less than (1-Y)/Y do:-
break // breaking the infinite while loop
return util

Function getUtil(Util, row, column, action):-
      Declare dr, dc, newR and newC:-
      if newR is less than 0 or newC less than 0 or newR greater than
num_Row or newC greater than or equal new_color newR equal newC equal 1
then:-
            return util[column][row]
      else:-
            return[newR][newC]

Function calculateUtil(Util, row, column, action):-
      u;- = reward,
      u;- += 0.1 * Y * getUtil(UTIL, row, column, (action-1)%4)
      u;- += 0.8 * Y * getUtil(UTIL, row, column, action)
u;- +=      u += 0.1 * Y * getUtil(UTIL, row, column, (action+1)%4)
```

```
return u


Function Iterations( policy, Util):-
      Set steps to 0
      while True do:- //set an infinite loop to get the optimal path
            Util equal policyEval(policy, Util)
            Unchanged equal true
      for row in num_row do:-
            for column in num_col:-
                  if row is less or equal 1 and column equal 3 or row equal
column equal 1 then:-
                        continue
                  maxAction and maxU equal to None

                  for action in num_acctions:-
                        u equal calculateUtil(Util, row, column, action)

                  if u is less than maxU then:-
                        maxAction and maxU equal action and u respectively

            if maxU is less than the calculatedUtil then:-
                  Policy[row][column] equal maxAction
                  Unhanged equal false
                  Steps equal to steps + 1
      If unchanged then:-
            break
printEnviron(policy)
Print steps
Get the path evaluated
Print the path
return policy
```

Python Code:

```
import random

# Sum of reward and discount should equal to 1 and rewards for non-
terminal state is constant
REWARD = -0.10
Y = 0.90
MAX_ERROR = 10**(-3)

# Setting up the environment initially
NUM_ACTIONS = 4

# each element represents each direction the agent(cabi) moves to pickup o
r drop
ACTIONS = [(1, 0), (0, -1), (-1, 0), (0, 1)]
rows = 3
Cols = 4
UTIL = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]]

# Visualising the enviroment
def Environment(arr, policy=False):
    res = ""
    for row in range(rows):
        res += "|"
        for column in range(Cols):
            #Assigning the Pos to Row 1 And Col 1
            if row == column == 1:
                val = "X"

           #Making the Boundaries
            elif row <= 1 and column == 5:
                val = "+1" if row == 0 else "-1"
            else:
                val = ["↓", " ← ", "↑", " → "][arr[row][column]]
            res += " " + val[:5].ljust(5) + " |" # format
        res += "\n"
    return res

#Decision At Random
policy = [[random.randint(0, 3) for j in range(Cols)] for i in range(rows)
]
SOLUTION = ""
```

```
#a few value iteration techniques are used to approximation the utilities.
def EvaluatePolicy(policy, util):
    while True:
        n_util = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]]
        error = 0

      #You can evaluate the policy by going through the Col and Row severa
l times.
        for row in range(rows):
            for column in range(Cols):
                if (row <= 1 and column == 3) or (row == column == 1):
                    continue
                n_util[row][column] = calculateUtil(util, row, column, pol
icy[row][column]) #Sample Update
                error = max(error, abs(n_util[row][column]-
util[row][column]))
        util = n_util
        if error < MAX_ERROR * (1-Y) / Y:
            break
    return util


def getUtil(util, row, column, action):
    r, c = ACTIONS[action]
    newR, newC = row + r, column + c
    if newR < 0 or newC < 0 or newR >= rows or newC >= Cols or (newR == ne
wC == 1): # collide with the boundary or the wall
        return util[row][column]
    else:
        return util[newR][newC]
#Estimating The state of the util
def calculateUtil(util, row, column, action):
    u = REWARD
    u += 0.1 * Y * getUtil(util, row, column, (action-1)%4)
    u += 0.8 * Y * getUtil(util, row, column, action)
    u += 0.1 * Y * getUtil(util, row, column, (action+1)%4)
    return u

def  policyIter(policy, UTIL, cab, pass1, pass2):
    steps = 0;
    while True:
        UTIL = EvaluatePolicy(policy, UTIL)
        unchanged = True
```

```
        changes = random.randint(0, 1)

        path = "Cab location : " , cab ," => "
       #Loopinf through the Rows
        for row in range(rows):
            for column in range(Cols):

                if (row <= 1 and column == 3) or (row == column == 1):
                    continue
                maxAction, maxU = None, -float("inf")

                for action in range(NUM_ACTIONS):

                    u = calculateUtil(UTIL, row, column, action)

                    if u > maxU:
                        maxAction, maxU = action, u

                if maxU > calculateUtil(UTIL, row, column, policy[row][col
umn]):
                    policy[row][column] = maxAction #Reducing Util
                    unchanged = False
                    steps = steps +1
        if unchanged:
            break

    pass1_drop = [random.randint(1, 5),random.randint(1, 5)]
    pass2_drop = [random.randint(1, 5),random.randint(1, 5)]
    if changes == 0:
        changes = random.randint(0, 1)
        path = ''.join(map(str, path)) ,
        "passanger2 pickup: ",pass2," => "

        if changes == 0:
            path = ''.join(map(str, path)) + "passanger 1 pickup: ",pass1,
" => "
            path = ''.join(map(str, path)) , "passanger 2 drop: ",pass2_dr
op," => "
            path = ''.join(map(str, path)) , "passanger 1 drop: ",pass1_dr
op
        elif changes == 1:
            path = ''.join(map(str, path)) , "passanger 2 drop: ",pass2_dr
op," => "
            path = ''.join(map(str, path)) , "passanger 1 pickup: ",pass1,
" => "
```

```
            path = ''.join(map(str, path)) , "passanger 1 drop: ",pass1_dr
op
    else:
        path = ''.join(map(str, path)) , "passanger 1 pickup: ",pass1," =>
 "
        changes = random.randint(0, 1)

        if changes == 0:
            path = ''.join(map(str, path)) , "passanger 2 pickup: ",pass2,
" => "
            path = ''.join(map(str, path)) , "passanger 1 drop: ",pass1_dr
op," => "
            path = ''.join(map(str, path)) , "passanger 2 drop : ",pass2_d
rop
        elif changes == 1:
            path = ''.join(map(str, path)) , "passanger 1 drop: ",pass1_dr
op," => "
            path = ''.join(map(str, path)) , "passsanger 2 pickup: ",pass2
," => "
            path = ''.join(map(str, path)) , "passanger 2 drop: ",pass2_dr
op

    Environment(policy)
    print("Steps = ",steps)
    listToStr = ' '.join([str(elem) for elem in path])
    print("The path is: ", path)
    print("The optimal path is:")
    return policy

if __name__ == "__main__":

    cab = [2,4]
    passanger1 = [1,5]
    passanger2 = [5,1]
    print("Episode 1 from figure 1:")
    print("The cab is located at : ", cab)
    print("The passenger 1 is located at : ", passanger1)
    print("The second 2 is located at : ", passanger2)

    # The Actual Policy
    policy = [[3, 1, 2, 0], [1, 1, 2, 3], [0, 3, 0, 3]]

    # print solution
    policy = policyIter(policy, UTIL, cab, passanger1, passanger2)
    SOLUTION = Environment(policy)
```

```python
    print(SOLUTION)

    #Looping An Policy
    for i in range(4):
        cab = [random.randint(1, 5),random.randint(1, 5)]
        passanger1 = [random.randint(1, 5),random.randint(1, 5)]
        passanger2 = [random.randint(1, 5),random.randint(1, 5)]

        print("Episode ", i+2,":")
        print("The Cab is located at :  ", cab)
        print("Passanger 1 is located at: ", passanger1)
        print("Pasenger 2 is located at:  ", passanger2)
        policy = [[random.randint(0, 3) for j in range(Cols)] for i in ran
ge(rows)]
        # Output

        policy = policyIter(policy, UTIL, cab, passanger1, passanger2)
        SOLUTION = Environment(policy)
        print(SOLUTION)
```

Ques1 :
The Best Path to Reach the destination by verifying the grid cells
The Taxi goes to the up to the cell(1,4) and pick the passenger in the I1 cell(1,5) by traversing to the left, Then It will go back to the Right to dropping the passenger, then this move to the Right to the cell (1,4) then to the up one cell to the other cell(2,4) and moves to the right 2 steps (2,2), And to up to the Cell(3,2) and the to the right to the Cell(3,1) Where it is the drop location for the First passenger and then it takes the third passenger. The taxi then takes the second passenger
Moves to the down to moves two cells and reaches to the second passenger destination(5,1).
This will be the End.
The optimal path:
T(2,4) > P1P(1,5)>P2P(3,1)>P1D(3,1)>P2D(5,1)

Ques 2 :
Episode 1

```
Episode 1 from figure 1:
The cab is located at :  [2, 4]
The passenger 1 is located at :  [1, 5]
The second 2 is located at :  [5, 1]
Steps =  10
The path is:  ('Cab location : [2, 4] => passanger 1 pickup: [1, 5] => passanger 1 drop: [4, 3] => passsanger 2 pickup: [5, 1] => ', 'passanger 2 drop: ', [2, 4])
The optimal path is:
| →    | →    | →    | ↓    |
| ↑    | X    | ↑    | →    |
| ↑    | →    | ↑    | ←    |
```

Ques 3 :
Episode 2

```
Episode  2 :
The Cab is located at :   [1, 2]
Passanger 1 is located at:  [4, 4]
Pasenger 2 is located at:   [3, 1]
Steps =  9
The path is:  ('Cab location : [1, 2] => passanger 1 pickup: [4, 4] => passanger 2 drop: [2, 3] => ', 'passanger 1 drop: ', [1, 1])
The optimal path is:
|  →   |  →   |  →   | ↑    |
| ↑    | X    | ↑    |  →   |
| ↑    |  →   | ↑    |  ←   |
```

Episode 3

```
Episode  3 :
The Cab is located at :   [2, 5]
Passanger 1 is located at:  [2, 2]
Pasenger 2 is located at:   [4, 4]
Steps =  8
The path is:  ('Cab location : [2, 5] => passanger 2 drop: [3, 2] => passanger 1 pickup: [2, 2] => ', 'passanger 1 drop: ', [3, 2])
The optimal path is:
|  →   |  →   |  →   | ↓    |
| ↑    | X    | ↑    | ↑    |
| ↑    |  →   | ↑    |  ←   |
```

Episode 4

```
Episode  4 :
The Cab is located at :   [4, 5]
Passanger 1 is located at:  [5, 5]
Pasenger 2 is located at:   [1, 3]
Steps =  11
The path is:  ('Cab location : [4, 5] => passanger 1 pickup: [5, 5] => passanger 2 pickup: [1, 3] => passanger 1 drop: [4, 4] => ', 'passanger 2 drop : ', [2, 1])
The optimal path is:
|  →   |  →   |  →   |  ←   |
| ↑    | X    | ↑    | ↓    |
| ↑    |  →   | ↑    |  ←   |
```

Episode 5

```
Episode  5 :
The Cab is located at :   [1, 5]
Passanger 1 is located at:  [5, 5]
Pasenger 2 is located at:   [1, 1]
Steps =  8
The path is:  ('Cab location : [1, 5] => passanger 1 pickup: [5, 5] => passanger 1 drop: [4, 1] => passsanger 2 pickup: [1, 1] => ', 'passanger 2 drop: ', [2, 2])
The optimal path is:
|  →   |  →   |  →   |  ←   |
| ↑    | X    | ↑    |  ←   |
| ↑    |  →   | ↑    |  ←   |
```