**Virtual Initio Programming: WS32 & Ex32 Sample Answers**

**NOTE:** Exercises have become sufficiently complex that quite wide variability in answers can be expected.

**WS32**

**Question 1:** It moves forward until it detects a line with its left line sensor.

**Question 2:** It moves forward until it reaches the square then it prints line\_left Goal Achieved!

**Exercise:**

import bdi.initioagent as cognitive

import time

class EnterAgent(cognitive.InitioAgent):

 def \_\_init\_\_(self):

 super().\_\_init\_\_()

 only\_line\_goal = self.AND(self.G('line\_left'), self.NOT(self.G('obstacle\_right')))

 goal\_and\_obstacle = self.AND(self.G('line\_left'), self.b\_obstacle\_centre)

 goal\_and\_no\_wall = self.AND(self.G('line\_left'), self.NOT(self.B('obstacle\_right')))

 goal\_and\_wall = self.AND(self.G('line\_left'), self.AND(self.B('obstacle\_right'), self.NOT(self.b\_obstacle\_centre)))

 self.add\_condition\_rule(goal\_and\_obstacle, self.spin\_left\_rule)

 self.add\_condition\_rule(self.G('obstacle\_right'), self.forward\_rule)

 self.add\_condition\_rule(goal\_and\_no\_wall, self.spin\_right\_rule)

 self.add\_condition\_rule(goal\_and\_wall, self.forward\_rule)

 self.add\_goal('line\_left')

 self.add\_goal('obstacle\_right')

 def b\_obstacle\_centre(self):

 if (self.beliefbase['distance'] < 30):

 return True

 return False

 def forward\_rule(self):

 self.robot.forward(10)

 time.sleep(5)

 self.robot.stop()

 def spin\_left\_rule(self):

 self.robot.spinLeft(10)

 time.sleep(1)

 self.robot.stop()

 def spin\_right\_rule(self):

 self.robot.spinRight(10)

 time.sleep(1)

**Ex32**

**Exercise 1:**

import bdi.initioagent as cognitive

import random, time

class MachineLearner(cognitive.InitioAgent):

 def \_\_init\_\_(self, learning\_rate):

 super().\_\_init\_\_()

 self.learning\_rate = learning\_rate

 self.epsilon = 1

 self.actions = ['forward', 'backward', 'left', 'right']

 self.dictionary = self.action\_reward(self.actions, 1)

 self.rewards = {(1, 1):1, (1, 0):2, (0, 1):0, (0, 0):1}

 self.add\_goal('learn\_oval')

 self.add\_condition\_rule(self.G('learn\_oval'), self.learn)

 self.add\_condition\_rule(self.B('learn\_oval'), self.end\_learning)

 def action\_reward(self, action\_list, default):

 action\_rewards = {}

 for i in range(0, 2):

 for j in range(0, 2):

 for k in (action\_list):

 action\_rewards[((i, j), k)] = default

 return action\_rewards

 def end\_learning(self):

 self.robot.stop()

 print(self.dictionary)

 self.done()

 def get\_max(self, state):

 max\_reward = 0

 action = 'forward'

 for act in self.actions:

 if (self.dictionary[(state, act)] > max\_reward):

 action = act

 max\_reward = self.dictionary[(state, act)]

 return action

 def action\_execute(self, action):

 if (action == "forward"):

 self.robot.forward(10)

 elif (action == "backward"):

 self.robot.reverse(10)

 elif (action == "left"):

 self.robot.spinLeft(10)

 elif (action == "right"):

 self.robot.spinRight(10)

 else:

 self.robot.stop()

 def learn(self):

 explore = random.random();

 state = (self.robot.irLeftLine(), self.robot.irRightLine())

 if (explore < self.epsilon):

 print("Exploring")

 action = random.choice(self.actions)

 else:

 action = self.get\_max(state)

 self.action\_execute(action)

 time.sleep(3)

 reward = self.rewards[(self.robot.irLeftLine(), self.robot.irRightLine())]

 if (reward == 2):

 self.epsilon = self.epsilon - 0.05

 print("updating epsilon: " + str(self.epsilon))

 if (self.epsilon < 0):

 self.add\_belief('learn\_oval')

 self.dictionary[(state, action)] = self.dictionary[(state, action)] + ( reward - self.dictionary[(state, action)]) \* self.learning\_rate

**Exercise 2:**

import bdi.initioagent as cognitive

import random, time

class MachineLearner(cognitive.InitioAgent):

 def \_\_init\_\_(self, learning\_rate):

 super().\_\_init\_\_()

 self.learning\_rate = learning\_rate

 self.epsilon = 1

 self.actions = ['forward', 'backward', 'left', 'right']

 self.dictionary = self.action\_reward(self.actions, 1)

 self.rewards = {(1, 1):1, (1, 0):2, (0, 1):0, (0, 0):1}

 self.add\_goal('follow\_oval')

 self.add\_condition\_rule(self.G('learn\_oval'), self.learn)

 self.add\_condition\_rule(self.AND(self.G('follow\_oval'), self.NOT(self.B('learn\_oval'))), self.first\_learn)

 self.add\_condition\_rule(self.AND(self.G('follow\_oval'), self.B('learn\_oval')), self.execute\_best)

 def action\_reward(self, action\_list, default):

 action\_rewards = {}

 for i in range(0, 2):

 for j in range(0, 2):

 for k in (action\_list):

 action\_rewards[((i, j), k)] = default

 return action\_rewards

 def end\_learning(self):

 self.robot.stop()

 print(self.dictionary)

 self.done()

 def get\_max(self, state):

 max\_reward = 0

 action = 'forward'

 for act in self.actions:

 if (self.dictionary[(state, act)] > max\_reward):

 action = act

 max\_reward = self.dictionary[(state, act)]

 return action

 def action\_execute(self, action):

 if (action == "forward"):

 self.robot.forward(10)

 elif (action == "backward"):

 self.robot.reverse(10)

 elif (action == "left"):

 self.robot.spinLeft(10)

 elif (action == "right"):

 self.robot.spinRight(10)

 else:

 self.robot.stop()

 def first\_learn(self):

 self.add\_goal('learn\_oval')

 def execute\_best(self):

 state = (self.robot.irLeftLine(), self.robot.irRightLine())

 best = self.get\_max(state)

 self.action\_execute(best)

 def learn(self):

 explore = random.random();

 state = (self.robot.irLeftLine(), self.robot.irRightLine())

 if (explore < self.epsilon):

 print("Exploring")

 action = random.choice(self.actions)

 else:

 action = self.get\_max(state)

 self.action\_execute(action)

 time.sleep(3)

 reward = self.rewards[(self.robot.irLeftLine(), self.robot.irRightLine())]

 if (reward == 2):

 self.epsilon = self.epsilon - 0.05

 print("updating epsilon: " + str(self.epsilon))

 if (self.epsilon < 0):

 self.add\_belief('learn\_oval')

 self.dictionary[(state, action)] = self.dictionary[(state, action)] + ( reward - self.dictionary[(state, action)]) \* self.learning\_rate

**Exercise 3:**

import bdi.initioagent as cognitive

import random, time

class ZigZag(cognitive.InitioAgent):

 def \_\_init\_\_(self):

 super().\_\_init\_\_()

 only\_line\_goal = self.AND(self.OR(self.G('line\_left'), self.G('clear')), self.NOT(self.G('obstacle\_right')))

 lgoal\_and\_obstacle = self.AND(self.G('line\_left'), self.b\_obstacle\_centre)

 lgoal\_and\_no\_wall = self.AND(self.G('line\_left'), self.NOT(self.B('obstacle\_right')))

 cgoal\_and\_obstacle = self.AND(self.G('clear'), self.b\_obstacle\_centre)

 cgoal\_and\_no\_wall = self.AND(self.G('clear'), self.NOT(self.B('obstacle\_left')))

 lgoal\_and\_wall = self.AND(self.G('line\_left'), self.AND(self.B('obstacle\_right'), self.NOT(self.b\_obstacle\_centre)))

 cgoal\_and\_wall = self.AND(self.G('clear'), self.AND(self.B('obstacle\_left'), self.NOT(self.b\_obstacle\_centre)))

 self.add\_condition\_rule(self.G('end'), self.end)

 self.add\_condition\_rule(self.AND(self.B('line\_left'), self.B('clear')), self.remove\_clear)

 self.add\_condition\_rule(self.AND(self.NOT(self.G('end')), self.AND(self.NOT(self.G('line\_left')), self.NOT(self.G('clear')))), self.ask)

 self.add\_condition\_rule(self.NOT(self.OR(self.B('line\_left'), self.B('clear'))), self.add\_clear)

 self.add\_condition\_rule(self.AND(self.G('obstacle\_right'), self.b\_obstacle\_centre), self.spin\_left\_rule)

 self.add\_condition\_rule(self.G('obstacle\_right'), self.forward\_rule)

 self.add\_condition\_rule(lgoal\_and\_obstacle, self.spin\_left\_rule)

 self.add\_condition\_rule(lgoal\_and\_no\_wall, self.forward\_then\_spin\_right\_rule)

 self.add\_condition\_rule(cgoal\_and\_obstacle, self.spin\_right\_rule)

 self.add\_condition\_rule(cgoal\_and\_no\_wall, self.forward\_then\_spin\_left\_rule)

 self.add\_condition\_rule(lgoal\_and\_wall, self.forward\_rule)

 self.add\_condition\_rule(cgoal\_and\_wall, self.forward\_rule)

 def end(self):

 self.add\_belief('end')

 self.done()

 def ask(self):

 self.drop\_belief('end')

 goal = input('What would you like me to do? [B]find black,[C]find clear,[E]end')

 if (goal == "B"):

 self.add\_goal('obstacle\_right')

 self.add\_goal('line\_left')

 elif (goal == "C"):

 self.add\_goal('obstacle\_right')

 self.add\_goal('clear')

 else:

 self.add\_goal('end')

 def remove\_clear(self):

 print("dropping clear")

 self.drop\_belief('clear')

 def add\_clear(self):

 self.add\_belief('clear')

 def forward\_rule(self):

 self.robot.forward(10)

 time.sleep(5)

 self.robot.stop()

 def spin\_left\_rule(self):

 self.robot.spinLeft(10)

 time.sleep(1)

 self.robot.stop()

 def spin\_right\_rule(self):

 self.robot.spinRight(10)

 time.sleep(1)

 self.robot.stop()

 def forward\_then\_spin\_left\_rule(self):

 self.robot.forward(10)

 time.sleep(1)

 self.robot.spinLeft(10)

 time.sleep(2)

 self.robot.stop()

 def forward\_then\_spin\_right\_rule(self):

 self.robot.forward(10)

 time.sleep(1)

 self.robot.spinRight(10)

 time.sleep(2)

 self.robot.stop()

 def b\_obstacle\_centre(self):

 if (self.beliefbase['distance'] < 30):

 return True

 return False



 University of Liverpool, 2020

This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-nc-sa/4.0/).