مسئله ی دنیای جارو برقی با عامل پیشرفته

در این مسئله ما قصد داریم که یک جاروبرقی با عامل پیشرفته ایجاد کنیم که با استفاده از سه ماژول (پیمانه) محیط یا Environs ، عامل یا Agency و جاروبرقی یا Vacuum کار می کند. بدین ترتیب که ابتدا ماژول محیط ، محیط مورد پیمایش توسط عامل را تعریف و ایجاد می کند که یک ماتریس یا آرایه با ابعاد 3\*3 یا m\*n می باشد که بعضی از خانه های آن آلوده هستند و با مقدار 1 مشخص شده اند و سایر خانه های آن تمیز هستند که با مقدار 0 مشخّص شده اند. این محیط شامل یک خانه ی آغازین و یک خانه ی مقصد و همچنین جهت پیمایش خانه ها می باشد. سپس ماژول عامل ، ادراکات لازم را از محیط دریافت می کند که شامل جهت حرکت و جا به جایی بین خانه های آرایه و بررسی رسیدن عامل به مجاورت دیوار و آلودگی خانه ی جاری و بررسی رسیدن به خانه ی هدف می شود. در انتها ماژول جاروبرقی با به ارث بردن این ادراکات و اهداف به عنوان عملگر نهایی وارد محیط شده و اقدام به ایجاد تغییرات در محیط و تمیز کردن خانه های آلوده ی آن می نماید.

در ادامه ما به بررسی این سه ماژول می پردازیم:

1. ماژول محیط یا Environs :

ابتدا کتابخانه ی numpy را وارد می کنیم:

import numpy as np

پس از آن اوّلین کلاس این بخش را که کلاس وضعیت است ، تعریف می کنیم که در آن یک تابع آغازگر تعریف می کنیم که آرایه ای با ابعاد x , y با خانه های صفر را در نظر می گیرد:

class State():  
 def \_\_init\_\_(self,x,y):  
 self.grid = np.zeros((x,y))

بعد کلاس حداقل و حداکثر را تعریف می کنیم که کلاس وضعیت را به ارث می برد و شامل یک تابع آغازگر می باشد که در آن از تابع آغاز گر به ارث برده شده از کلاس وضعیّت استفاده می شود و ابعاد آرایه و خانه های آلوده را لحاظ می کند که در آن به صورت پیش فرض ابعاد را 2\*2 و محل آلوده را خانه ی (0 , 1) لحاظ می کند:

class MiniMax2(State):  
 def \_\_init\_\_(self, dirt\_locs):  
 x, y = 2,2  
 State.\_\_init\_\_(self,x,y)  
 if dirt\_locs is not None:  
 for loc in dirt\_locs:  
 self.grid[loc[0],loc[1]] = 1

سپس کلاس محدوده ی تصادفی را تعریف می کنیم که بازهم کلاس وضعیّت را به ارث برده و از تابع آغازگر آن استفاده می نماید. این کلاس با کلاس حداقل و حداکثر تقریبا معادل است و در فراخوانی نهایی یکی از این دو کلاس می تواند مورد فراخوانی قرار بگیرد و محیط ما را ایجاد نماید:

class LimitedRandom(State):  
 def \_\_init\_\_(self):  
 x, y = np.random.randint(8,16), np.random.randint(8,16)  
 State.\_\_init\_\_(self,x,y)  
 for a in range(x):  
 for b in range(y):  
 if 100\*np.random.random() < 5:  
 self.grid[a,b] = 1

در اینجا ما تابع باز کردن بسته بندی را تعریف می کنیم که وظیفه ی آن تبدیل اعداد ورودی به لیست ها می باشد:

def unpack(lst, x, \*args):  
 nulst =[]  
 x = x if type(x)==list else [x]  
 if not lst:  
 nulst = [[x\_0] for x\_0 in x]  
 else:  
 nulst = [l+[x\_0] for l in lst for x\_0 in x]  
 if args:  
 return unpack(nulst, \*args)  
 else:  
 return nulst

سپس ما تابع چند جهانی را تعریف می کنیم که شامل ورودی های وضعیت و آرگومان ها می باشد:

def multiverse(state, \*args):  
 verse = []  
 lst = unpack([], \*args)  
 for variant in lst:  
 verse.append(state(\*variant))  
 return verse

در انتها ما تابع بسته های حداقل و حداکثر را تعریف می کنیم که این بسته ها به عنوان ورودی های کلاس با نام مشابه و درصورت برابری ابعاد با ابعاد تعریف شده در کلاس می توانند فراخوانی شوند:

def MiniMax2Package():  
 locs = [None,  
 [(0,0)],  
 [(0,1)],  
 [(1,0)],  
 [(1,1)],  
 [(0,0),(0,0)],  
 [(0,0),(0,1)],  
 [(0,0),(1,0)],  
 [(0,0),(1,1)],  
 [(0,1),(0,0)],  
 [(0,1),(0,1)],  
 [(0,1),(1,0)],  
 [(0,1),(1,1)],  
 [(1,0),(0,0)],  
 [(1,0),(0,1)],  
 [(1,0),(1,0)],  
 [(1,0),(1,1)],  
 [(1,1),(0,0)],  
 [(1,1),(0,1)],  
 [(1,1),(1,0)],  
 [(1,1),(1,1)]]  
 return multiverse(MiniMax2,locs)

2. ماژول عامل یا Agency :

ابتدا کتابخانه ی numpy را وارد می کنیم:

import numpy as np

بعد کلاس عامل را تعریف می کنیم:

class Agent():  
 *'''Template for Agent class. Uncallable'''*

در کلاس عامل ابتدا تابع آغازگر را تعریف می کنیم که در آن متغیرهای ورودی موقعیّت شروع و موقعیّت مقصد و جهت حرکت را تعریف می کنیم و این متغیّرها را به متغیر self این تابع نسبت می دهیم که با این کار این متغیرها به صفات این کلاس تبدیل می شوند:

def \_\_init\_\_(self, start\_loc, home\_loc, start\_face):  
 self.bump = False  
 self.dirty = False  
 self.home = False  
 self.loc = start\_loc  
 self.home\_loc =home\_loc  
 self.bearing = self.\_\_bearing\_\_(start\_face)  
 self.percept\_sequence = []  
 self.action = 'Power Up'

سپس تابع جهت را تعریف می کنیم که هر یک از حروف N ، S ، E و W را به یک مقدار مختصات تبدیل می کند:

def \_\_bearing\_\_(self, face):  
 *'''Converts an alphanumeric bearing into a coordinate'''* return (np.array([-1,0]) if face=='N'  
 else np.array([1,0]) if face=='S'  
 else np.array([0,1]) if face=='E'  
 else np.array([0,-1]))

سپس تابع دریافت ادراک را تعریف می کنیم که مشخص می کند کدام یک از حالت های مورد انتظار محیط اتفاق افتاده است:

def get\_percept(self, state):  
 *'''  
 Sets conditions for when each of three onboard detectors should be*

*set to 1  
 '''* self.bump = False  
 self.dirty = False  
 self.home = False  
 if (min(self.loc + self.bearing) < 0  
 or min(state.shape - self.loc - self.bearing) == 0  
 or state[self.loc[0] + self.bearing[0],  
 self.loc[1] + self.bearing[1]] == 2):  
 self.bump = True  
 if state[self.loc[0], self.loc[1]] == 1:  
 self.dirty = True  
 # determine if vacuum is in home loc  
 if (self.loc-self.home\_loc).sum() == 0:  
 self.home = True  
 print('Bump :' , self.bump , '| Dirty :' , self.dirty , '| Home :' , self.home)  
 self.percept\_sequence.append((self.bump, self.dirty, self.home))

سپس کلاس جدول بی اهمیّت همراه عامل را تعریف می کنیم که کلاس عامل را به ارث می برد و عملیات عامل را در یک جدول ذخیره می کند:

class TrivialTableLookupAgent(Agent):  
 *''' An Agent class that stores instructions in a table'''*

در این کلاس تابع برنامه را تعریف می کنیم که در آن ادراکات دریافت شده از آخرین خانه ی محیط بررسی می شود و با توجّه به آن اقدام لازم و مورد نظر انجام می شود:

def program(self):  
 if self.percept\_sequence[-1][1]==True:  
 self.action = 'Suck'  
 elif self.percept\_sequence[-1][2]==True and (len(self.percept\_sequence)>2):  
 self.action = 'Power Down'  
 elif (len(self.percept\_sequence)  
 - sum([percept[1] for percept in self.percept\_sequence])  
 )% 2 == 1:  
 self.action = 'Move'  
 else:  
 self.action = 'Right Turn'  
 print('Action :' , self.action)

سپس کلاس عامل واکنشی پایه را تعریف می کنیم که از ویژگی های کلاس عامل را به ارث می برد و وظیفه ی آن پاسخ دادن به محیط مجاور می باشد:

class BasicReflexAgent(Agent):  
 *''' An Agent class that responds to its immediate environment'''*

در این کلاس ما تابع برنامه را تعریف می کنیم که وظیفه ی آن بررسی وضعیت فعلی خانه ی جاری و انجام اقدام مناسب برای آن است:

def program(self):  
 if self.dirty==True:  
 self.action = 'Suck'  
 elif self.bump==True:  
 do = 100\*np.random.random()  
 if do < 49:  
 self.action = 'Right Turn'  
 elif do < 98:  
 self.action = 'Left Turn'  
 else:  
 self.action = 'Power Down'  
 else:  
 do = 100\*np.random.random()  
 if do < 50:  
 self.action = 'Move'  
 elif do < 74:  
 self.action = 'Right Turn'  
 elif do < 98:  
 self.action = 'Right Turn'  
 else:  
 self.action = 'Power Down'  
 print('Action :' , self.action)

در این مرحله ما کلاس عامل واکنشی وابسته به وضعیت داخلی اتاق خالی را تعریف می کنیم که ویژگی های کلاس عامل را به ارث می برد و وظیفه ی آن پاسخ دادن به محیط مجاور می باشد:

class EmptyRoomInternalStateReflexAgent(Agent):  
 *''' An Agent class that responds to its immediate environment'''*

در این کلاس ما ابتدا تابع آغازگر را تعریف می کنیم که شامل متغیّرهای ورودی موقعیت آغازین ، موقعیت پایانی و جهت حرکت می باشد و با استفاده از ویژگی های کلاس عامل ، خانه های آلوده ی محیط را مشخّص می کند:

def \_\_init\_\_(self, start\_loc, home\_loc, start\_face):  
 Agent.\_\_init\_\_(self, start\_loc, home\_loc, start\_face)  
 self.internal\_state = np.ones((15,15))

سپس ما تابع برنامه را تعریف می کنیم که با توجّه به وضعیّت فعلی خانه ی جاری ، اقدام مناسب را انجام می دهد:

def program(self):  
 self.internal\_state[self.loc[0], self.loc[1]] = 0  
 if self.dirty:  
 self.action = 'Suck'  
 elif self.internal\_state.any():  
 if self.bump:  
 self.action = 'Right Turn'  
 if self.bearing[1] == 1 and self.loc[1] < 15:  
 self.internal\_state = self.internal\_state[:,:self.loc[1]+1]  
 elif self.bearing[0] == 1 and self.loc[0] < 15:  
 self.internal\_state = self.internal\_state[:self.loc[0]+1,:]  
 elif self.internal\_state[self.loc[0]+self.bearing[0],  
 self.loc[1]+self.bearing[1]] == 0:  
 self.action = 'Right Turn'  
 else:  
 self.action = 'Move'  
 else:  
 if self.home:  
 self.action = 'Power Down'  
 elif self.bump:  
 if self.bearing[0] == -1:  
 self.action = 'Left Turn'  
 else:  
 self.action = 'Right Turn'  
 elif min(self.bearing) == -1 :  
 self.action = 'Move'  
 else:  
 if self.bearing[0] == 1:  
 self.action = 'Right Turn'  
 else:  
 self.action = 'Left Turn'  
 print('Action :' , self.action)

3. ماژول جاروبرقی یا Vacuum :

ابتدا کتابخانه ی numpy و ماژول های محیط (Environs) و عامل (Agency) را وارد می کنیم:

import numpy as np  
import Environs  
import Agency

سپس تابع امتیازدهی را تعریف می کنیم:

def f\_scoring(scores, agents, state):  
 for i, agent in enumerate(agents):  
 if agent.action != 'Power Down':  
 scores[i] -= 1  
 if agent.action == 'Suck' and state[agent.loc[0], agent.loc[1]] == 1:  
 scores[i] += 100  
 if agent.action == 'Power Down' and (agent.loc-agent.home\_loc).sum()!=0:  
 scores[i] -= 1000  
 return scores

بعد از آن تابع وضعیت فاقد خانه را تعریف می کنیم:

def f\_homeless(scores, agents, state):  
 for i, agent in enumerate(agents):  
 if agent.action != 'Power Down':  
 scores[i] -= 1  
 if agent.action == 'Suck' and state[agent.loc[0], agent.loc[1]] == 1:  
 scores[i] += 100  
 return scores

سپس تابع عملیات را تعریف می کنیم که با توجه به وضعیت فعلی خانه ی جاری اقدام لازم را انجام می دهد و وضعیت نهایی را پس از عملیات بر می گرداند:

def f\_action(agents, state):  
 for agent in agents:  
 if agent.action == 'Suck':  
 state[agent.loc[0], agent.loc[1]] = 0  
 elif (agent.action == 'Move'  
 and min(agent.loc + agent.bearing) >= 0  
 and min(state.shape - agent.loc - agent.bearing) > 0  
 and state[agent.loc[0] + agent.bearing[0],  
 agent.loc[1] + agent.bearing[1]] != 2):  
 agent.loc += agent.bearing  
 elif agent.action == 'Right Turn':  
 agent.bearing = [agent.bearing[1], -agent.bearing[0]]  
 elif agent.action == 'Left Turn':  
 agent.bearing = [-agent.bearing[1], agent.bearing[0]]  
 return state

و در انتها تابع ارزیابی محیط را تعریف می کنیم که با توجه به وضعیت های مختلف محیط و اقدامات انجام شده روی آن و تغییرات ایجاد شده در آن ، معیار کارآیی را اندازه گیری می کند و امتیاز عملگر را باز می گرداند:

def run\_eval\_environment(state, update, agents, performance):  
 scores = [0 for \_ in range(len(agents))]  
 while any([agent.action != 'Power Down' for agent in agents]):  
 for agent in agents:  
 agent.get\_percept(state)  
 agent.program()  
 scores = performance(scores, agents, state)  
 state = update(agents, state)  
 return scores

برای اجرای این برنامه در انتهای ماژول جاروبرقی قطعه کد زیر را وارد کرده و با استفاده از انواع عامل ها و محیط های تعریف شده در ماژول های مربوطه ، نوع محیط و عامل مورد نظر را انتخاب می کنیم و با استفاده از تابع امتیاز دهی و عملیات و ارزیابی تعریف شده در ماژول جاروبرقی ، میزان امتیاز و معیار کارآیی عملگر را دریافت و بررسی می کنیم:

if \_\_name\_\_ == '\_\_main\_\_':

در این بخش با استفاده از یک حلقه ی for محیط مربوط به کلاس MiniMax2 مورد استفاده و ارزیابی قرار می گیرد:

for S in Environs.MiniMax2Package():  
 environment1 = S.grid  
 agents = [Agency.TrivialTableLookupAgent(np.array([0,0]), np.array([0,0]), 'N')]  
 print(environment1)  
 print('The Environment Scale Is :' , np.shape(environment1))  
 print(run\_eval\_environment(environment1, f\_action, agents, f\_scoring))  
  
 environment2 = S.grid  
 agents = [Agency.TrivialTableLookupAgent(np.array([0,0]), np.array([0,0]), 'S')]  
 print(environment2)  
 print('The Environment Scale Is :' , np.shape(environment2))  
 print(run\_eval\_environment(environment2, f\_action, agents, f\_scoring))  
  
 environment3 = S.grid  
 agents = [Agency.TrivialTableLookupAgent(np.array([0,0]), np.array([0,0]), 'E')]  
 print(environment3)  
 print('The Environment Scale Is :' , np.shape(environment3))  
 print(run\_eval\_environment(environment3, f\_action, agents, f\_scoring))  
  
 environment4 = S.grid  
 agents = [Agency.TrivialTableLookupAgent(np.array([0,0]), np.array([0,0]), 'W')]  
 print(environment4)  
 print('The Environment Scale Is :' , np.shape(environment4))  
 print(run\_eval\_environment(environment4, f\_action, agents, f\_scoring))

و در این بخش محیط مربوط به کلاس LimitedRandom مورد استفاده و ارزیابی قرار می گیرد:

environment5 = Environs.LimitedRandom().grid  
 agents = [Agency.EmptyRoomInternalStateReflexAgent(np.array([0,0]), np.array([0,0]), 'N')]  
 print(environment5)  
 print('The Environment Scale Is :' , np.shape(environment5))  
 print(run\_eval\_environment(environment5, f\_action, agents, f\_scoring))  
  
 environment6 = Environs.LimitedRandom().grid  
 agents = [Agency.EmptyRoomInternalStateReflexAgent(np.array([0,0]), np.array([0,0]), 'S')]  
 print(environment6)  
 print('The Environment Scale Is :' , np.shape(environment6))  
 print(run\_eval\_environment(environment6, f\_action, agents, f\_scoring))  
  
 environment7 = Environs.LimitedRandom().grid  
 agents = [Agency.EmptyRoomInternalStateReflexAgent(np.array([0,0]), np.array([0,0]), 'E')]  
 print(environment7)  
 print('The Environment Scale Is :' , np.shape(environment7))  
 print(run\_eval\_environment(environment7, f\_action, agents, f\_scoring))  
  
 environment8 = Environs.LimitedRandom().grid  
 agents = [Agency.EmptyRoomInternalStateReflexAgent(np.array([0,0]), np.array([0,0]), 'W')]  
 print(environment8)  
 print('The Environment Scale Is :' , np.shape(environment8))  
 print(run\_eval\_environment(environment8, f\_action, agents, f\_scoring))