lets first begin by importing necessary libraries

This notebook serves as supporting material for topics covered in Lab 01 , lab 02 and the assignment for the Module Knowledge representation COMP9016

In [64]:

```
from agents import *
from notebook import psource
```

OVERVIEW

An agent, as defined in 2.1, is anything that can perceive its **environment** through sensors, and act upon that environment through actuators based on its **agent program**. This can be a dog, a robot, or even you. As long as you can perceive the environment and act on it, you are an agent. This notebook will explain how to implement a simple agent, create an environment, and implement a program that helps the agent act on the environment based on its percepts.

PEAS approach demystified

Peas stands for -

1-Performance

2-Environment

3-Actuators

4-Sensors

The pawn in our game represents the agent we can have just one or many characters in the game.

The environment is the place where the pawn interacts with things suppose he eats a cherry then cherry is a part of environment.

Performance is something that the player is looking forward to do -example maximizing the score it can also be minimizing say deaths for example

the actions that our agent can take in the environment say using his hands he could eat the cherry, then his hands are actuators because they help him perform a task in the environment that change the environment.

Sensors are the objects that help us perceive the environment, since we can't tell information about the environment to our agent then our agent will have some sensors of his own to detect the world around him. so he can use his eyes to see if it is a cherry

All in all Cherry is a thing in environment.

person is a pawn or agent placed in the environment.

His health is the measure of performance, if he eats cherry his performance increases

actuators are his hands that help him grab the cherry and eat it

eyes are his sensors that help him identify if a cherry is there.

before we begin lets check out what really is our goal for the lab 01

1- We need to specify using peas for our own problem

2- We need to have atleast 4 actions

3- We need to have atleast 4 percepts

So how do we really use this approach ?

To better explain the idea I will be creating my own PEAS for a simulation of a classroom

AGENT- my prof for this subject would be my central pawn

ENV - will be the class room that consist of students, canvas, meeting , free time

Performance - clearing all doubts

Actuators- our char oriley can perform at the moment only 4 tasks

```
1- he can move down

2- he can move up

3- he can smile ( Coz i love his smile :) )

4- he can upload assignment on canvas
```

AGENT

Let us now see how we define an agent. Run the next cell to see how `Agent` is defined in agents module. but lets just see what do we have inside the agent class in the library aima

In [65]:

```
psource(Agent)
```

```
class Agent(Thing):
    """An Agent is a subclass of Thing with one required slot,
    .program, which should hold a function that takes one argument, the
    percept, and returns an action. (What counts as a percept or action
    will depend on the specific environment in which the agent exists.)
    Note that 'program' is a slot, not a method. If it were a method,
    then the program could 'cheat' and look at aspects of the agent.
    It's not supposed to do that: the program can only look at the
    percepts. An agent program that needs a model of the world (and of
    the agent itself) will have to build and maintain its own model.
    There is an optional slot, .performance, which is a number giving
    the performance measure of the agent in its environment."""

    def __init__(self, program=None):
        self.alive = True
        self.bump = False
        self.holding = []
        self.performance = 0
        if program is None or not isinstance(program, collections.Callable):
            print("Can't find a valid program for {}, falling back to default.".format(
                self.__class__.__name__))

            def program(percept):
                return eval(input('Percept={}; action? '.format(percept)))

        self.program = program

    def can_grab(self, thing):
        """Return True if this agent can grab this thing.
        Override for appropriate subclasses of Agent and Thing."""
        return False
```

The `Agent` has two methods.

- `__init__(self, program=None)` : The constructor defines various attributes of the Agent. These include
  - `alive` : which keeps track of whether the agent is alive or not

- `bump` : which tracks if the agent collides with an edge of the environment (for eg, a wall in a park)
- `holding` : which is a list containing the `Things` an agent is holding,
- `performance` : which evaluates the performance metrics of the agent
- `program` : which is the agent program and maps an agent's percepts to actions in the environment. If no implementation is provided, it defaults to asking the user to provide actions for each percept.

- `can_grab(self, thing)` : Is used when an environment contains things that an agent can grab and carry. By default, an agent can carry nothing.

Agent's moves cost him -1 points for each block

- -ve points: moving every step costs him one point

Agent lands on students - he clears the doubts and smiles

- gives him +100 points

Agent lands on canvas block- he uploads an assignment

- gives him -10 points

Agent can get stuck in meeting - which will push him back 2 steps and the meeting block will disappear

- puts him 1 block behind without costing him anything
- the backward movement does not decrease his performance

agent can have free time- if he gets free time he will move two blocks ahead

- instantly puts the agent two blocks ahead in from current facing direction but this still costs him penalty

He can get two types of points ,either positive or negative :

-ve points: moving every step costs him one point

+ve points: clearing doubts and after smiling gives him 100 points

this way our agent is only dependent upon the number of students he finds so that he can solve their doubts and smile if he smiles after clearing the doubts he gets **100 pts coz I ALREADY MENTIONED HIS SMILE IS WORTH 10 TIMES MORE THAN THE ASSIGNMENT .**

Anyway focussing back to work , lets visualize our environment which has two objects or classes say

ENVIRONMENT

Now, let us see how environments are defined. Running the next cell will display an implementation of the abstract `Environment` class.

In [66]:

```
#psource(Environment)
```

`Environment` class has lot of methods! But most of them are incredibly simple, so let's see the ones we'll be using in this notebook.

- `thing_classes(self)` : Returns a static array of `Thing` sub-classes that determine what things are allowed in the environment and what aren't
- `add_thing(self, thing, location=None)` : Adds a thing to the environment at location
- `run(self, steps)` : Runs an environment with the agent in it for a given number of steps.
- `is_done(self)` : Returns true if the objective of the agent and the environment has been completed

The next two functions must be implemented by each subclasses of `Environment` for the agent to recieve percepts and execute actions

- `percept(self, agent)` : Given an agent, this method returns a list of percepts that the agent sees at the current time
- `execute_action(self, agent, action)` : The environment reacts to an action performed by a given agent. The changes may result in agent experiencing new percepts or other elements reacting to agent input.

SIMPLE AGENT AND ENVIRONMENT

SIMPLE AGENT AND ENVIRONMENT

Let's begin by using the `Agent` class to creating our first agent - oriley.

In [67]:

```
class oriley(Agent): #the name of the agent class that we want to keep , this class inherits agent
s class
    def smile(self, thing):
        print("Oriley: smiled at {}.".format(self.location))

    def assignment(self, thing):
        print("Oriley: gave assignment on canvas at {}.".format( self.location))

    def meeting(self,thing):
        print("Oriley: was busy at the meeting at {}.".format( self.location))
        print("Oriley: he has to be moved back 2 blocks  behind at {}.".format( self.location-2))

    def free_time(self,thing):
        print("Oriley: found some free time hence he moved 2 blocks at {}.".format( self.location+2
))

oriley = oriley()
```

```
Can't find a valid program for oriley, falling back to default.
```

What we have just done is create an instance of oriley who can only feel what's in his location (since he's a very busy man), and can go to students to clear doubts or upload assignment or can be in a meeting or if he has free time he will move fast. Let's see if he's alive...

In [68]:

```
print(oriley.alive) #this tells our oriley is alive
```

```
True
```

Love the way he smiles No ?

love the way he smiles : 😁 me too !! I like his style of teaching , he is always confident and makes me feel motivated thats why his smile is worth 100 pts

This is our prof. How cool is he? Well, he's a very busy person lets help him get back to work. For him to do this, we need to give him a program. But before that, let's create a college for our professor to teach.

ENVIRONMENT - CIT

A cit is an example of an environment because our prof oriley can perceive and act upon it. The **Environment** class is an abstract class, so we will have to create our own subclass from it before we can use it.

In [69]:

```
class student(Thing):#all these are blocks placed in env hence have to be things
    pass

class canvas(Thing):#all these are blocks placed in env hence have to be things
    pass

class meeting(Thing):#all these are blocks placed in env hence have to be things
    pass

class free_time(Thing):#all these are blocks placed in env hence have to be things
    pass

class cit(Environment):

    def percept(self, agent): #get values from the sensors at agent's current location
        '''return a list of things that are in our agent's location'''
        things = self.list_things_at(agent.location)
        return things

    def execute_action(self, agent, action):#executes an action depending upon the action
parameter
        #over here it has 4 options (1-move down,2- smile ,3- upload assignment 4- move up )
        '''changes the state of the environment based on what the agent does.'''

        if action == "move down":#if move down
            print('{} decided to {} at location: {}'.format(str(agent)[1:-1], action,
agent.location+1))
            agent.movedown(1) #call a move down function that moves the agent's position
            agent.performance-=1

        elif action == "move up":#if move up
            print('{} decided to {} at location: {}'.format(str(agent)[1:-1], action,
agent.location-1))
            agent.moveup(1) #call a move up function that moves the agent's position
            agent.performance-=1

        elif action == "smile":#if action is eating
            items = self.list_things_at(agent.location, tclass=student)#list the thing at agent's c
urrent pos if it's
            #student then smile
            if len(items) != 0: #if its not empty
                if agent.smile(items[0]): #check if the thing is actully matching to the action set
for smile
                    print('{} smiled at {} at location: {}'#agent smiles on the stundents
                          .format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))
                    self.delete_thing(items[0]) #we delete the block coz it's already performed
                    agent.performance+=100

        elif action == "assignment": #if action is assignment
            items = self.list_things_at(agent.location, tclass=canvas)#look that current pos should
have canvas in it
            if len(items) != 0: # if the list is not empty and has object canvas
```

```
                if agent.assignment(items[0]):
                    print('{} gave assignment on {} at location: {}' #we upload assignment on
canvas
                          .format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))
                    self.delete_thing(items[0]) #we delete the block coz it's already performed
                    agent.performance-=100

        elif action == "meeting": #if action is assignment
            items = self.list_things_at(agent.location, tclass=meeting)#look that current pos shoul
d have meeting in it
            if len(items) != 0: # if the list is not empty and has object meeting
                if agent.meeting(items[0]): #we check if the thing is meeting
                    print('{} was stuck in a {} at location: {}'
                          .format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))
                    self.delete_thing(items[0]) #we delete the block coz it's already performed
                    #now he has to move back 2 steps
                    print('{} was forced to move back 2 steps now he is at location: {}'.format(str
(agent)[1:-1], agent.location-2))
                    agent.moveup(1)

        elif action == "free_time": #if action is assignment
            items = self.list_things_at(agent.location, tclass=free_time)#look that current pos shc
uld have free_time in it
            if len(items) != 0: # if the list is not empty
                if agent.free_time(items[0]): #and has the first instance of the list as free_time
                    print('{} faced {} at location: {} so he will speed it up 2 blocks  '
                          .format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))
                    self.delete_thing(items[0]) #we delete the block coz it's already performed
                    print('{} was forced to move ahead 2 steps now at location: {}'.format(str(agen
t)[1:-1], agent.location+2))
                    agent.movedown(1)


    def is_done(self): #this function defines our performance if its true we are finished, we can
rest
        '''By default, we're done when we can't find a live agent,
        but to prevent killing our awesome prof, we will stop before itself - when there is no
more objects or blocks'''
        #any retuns if one condition is true
        no_blocks = not any(isinstance(thing, student) or isinstance(thing, canvas) or isinstance(t
hing, meeting) or isinstance(thing, free_time) for thing in self.things)
        #if blocks is not in things then the above statement will be true else false
        dead_agents = not any(agent.is_alive() for agent in self.agents)
        #if any agent is dead dead_Agent=True
        return dead_agents or no_blocks #return the status oring both and if either is true just
return true meaning we are done
```

PROGRAM - Oriley

Now that we have a **cit** Class, we re-implement our **oriley** to be able to move down/up and smile, give assignment or be in meeting only if it is present.

In [70]:

```
class oriley(Agent):
    location = 1


    def moveup(self,val):
        self.location -= val#change location to be -1 now

    def movedown(self,val): #change location to be +1 now
        self.location += val

    def smile(self, thing):
        '''returns True upon success or False otherwise'''
        if isinstance(thing, student):
            return True
        return False

    def assignment(self, thing):
        ''' returns True upon success or False otherwise'''
        if isinstance(thing, canvas):
            return True
        return False
```

```
        return False

    def meeting(self, thing):
        ''' returns True upon success or False otherwise'''
        if isinstance(thing, meeting):
            return True
        return False

    def free_time(self, thing):
        ''' returns True upon success or False otherwise'''
        if isinstance(thing, free_time):
            return True
        return False
```

Now its time to implement a **program** module for our oriley. A program controls how he acts upon its environment which is cit. Our program will be very simple, and is shown in the table below.

Percept:	Feel student	Feel canvas block	Feel meeting block	Feel free slot
Action:	solve doubt and smile	upload assignment	makes him go back 2 steps	makes him find free slots so move him two blocks away

In [71]:

```
def program(percepts):
    '''Returns an action based on the oriley's percepts'''
    for p in percepts:
        if isinstance(p, student):
            return 'smile'
        elif isinstance(p, canvas):
            return 'assignment'
        elif isinstance(p, meeting):
            return 'meeting'
        elif isinstance(p, free_time):
            return 'free_time'
    return 'move down' #in general he is always moving down every time in the 1-d world as in blind
dog example
```

Let's now run our simulation by creating cit with some students, canvas blocks, meetings and free time for our oriley.

In [72]:

```
#class name can't be same as the variable stored in it
#initialize all the classes
#1- college , 2- person , 3 - things in env

college = cit()
person = oriley(program)
students = student()

canv = canvas()
m1=meeting()
t1=free_time()


oriley_loc=1
student_instances=2
student_limit=10
canvas_limit=52
canvas_instances=3


#can be feed to make random values
import random
val_1=random.randint(2,student_limit)
val_2=random.randint(10,canvas_limit)

#only one instance of the thing is allowed by default in class
college.add_thing(students,2)
college.add_thing(canv,5)
college.add_thing(m1,3)
college.add_thing(t1,4)

college.add_thing(person, oriley_loc)
```

```
college.run(4)#total actions including fwd and back ward
print("final performance score out of the simulation is {}".format(person.performance))
```

```
oriley decided to move down at location: 2
oriley smiled at student at location: 2
oriley decided to move down at location: 3
oriley was stuck in a meeting at location: 3
oriley was forced to move back 2 steps now he is at location: 1
final performance score out of the simulation is 98
```

college.run(4) makes oriley to do 4 moves

1. so he move from 1 to loc 2 where he smiles
2. he smiles gets +100
3. he moves from 2 to 3
4. at 3 he is stuck in the meeting and pushed back

so the total performance here is 98 which is dependent of the actions that our agent performed

In [73]:

```
college.run(5) #moves specified number of steps in the environment
print("final performance score out of the simulation is {}".format(person.performance))
```

```
oriley decided to move down at location: 3
oriley decided to move down at location: 4
oriley faced free_time at location: 4 so he will speed it up 2 blocks
oriley was forced to move ahead 2 steps now at location: 6
oriley gave assignment on canvas at location: 5
final performance score out of the simulation is -4
```

Perfect! Note how the simulation stopped after all the blocks are finished, this ends our simulation, as we had defined before. Let's add some more canvas block and see if our agent can reach it.

In [74]:

```
college.add_thing(canv, 17) #this value should be more than previously used value
college.run(20)
```

```
oriley decided to move down at location: 6
oriley decided to move down at location: 7
oriley decided to move down at location: 8
oriley decided to move down at location: 9
oriley decided to move down at location: 10
oriley decided to move down at location: 11
oriley decided to move down at location: 12
oriley decided to move down at location: 13
oriley decided to move down at location: 14
oriley decided to move down at location: 15
oriley decided to move down at location: 16
oriley decided to move down at location: 17
oriley gave assignment on canvas at location: 17
```

Above, we learnt to implement an agent, its program, and an environment on which it acts. However, this was a very simple case. Let's try to add complexity to it by creating a 2-Dimensional environment!

AGENTS IN A 2D ENVIRONMENT

For us to not read so many logs of what our dog did, we add a bit of graphics while making our Park 2D. To do so, we will need to make it a subclass of **GraphicEnvironment** instead of Environment. Parks implemented by subclassing **GraphicEnvironment** class adds these extra properties to it:

- Our park is indexed in the 4th quadrant of the X-Y plane.
- Every time we create a park subclassing **GraphicEnvironment**, we need to define the colors of all the things we plan to put into the park. The colors are defined in typical RGB digital 8-bit format, common across the web.
- Fences are added automatically to all parks so that our dog does not go outside the park's boundary - it just isn't safe for blind dogs to be outside the park by themselves! **GraphicEnvironment** provides `is_inbounds` function to check if our dog tries to

leave the park.

First let us try to upgrade our 1-dimensional `Park` environment by just replacing its superclass by `GraphicEnvironment`.

In [75]:

```
class student(Thing):#all these are blocks placed in env hence have to be things
    pass

class canvas(Thing):#all these are blocks placed in env hence have to be things
    pass

class meeting(Thing):#all these are blocks placed in env hence have to be things
    pass

class free_time(Thing):#all these are blocks placed in env hence have to be things
    pass

class Bump(Thing):#all these are blocks placed in env hence have to be things
    pass

class cit2d(GraphicEnvironment):#this time graphic env

    def percept(self, agent): #get values from the sensors at agent's current location
        '''return a list of things that are in our agent's location'''
        things = self.list_things_at(agent.location)
        return things

    def execute_action(self, agent, action):#executes an action depending upon the action
parameter
        #over here it has 4 options (1-move down,2- smile ,3- upload assignment 4- move up )
        '''changes the state of the environment based on what the agent does.'''

        if action == "move down":#if move down
            print('{} decided to {} at location: {}'.format(str(agent)[1:-1], action,
agent.location[1]+1))
            agent.movedown(1) #call a move down function that moves the agent's position
            agent.performance-=1

        elif action == "move up":#if move up
            print('{} decided to {} at location: {}'.format(str(agent)[1:-1], action,
agent.location[1]-1))
            agent.moveup(1) #call a move up function that moves the agent's position
            agent.performance-=1

        elif action == "smile":#if action is eating
            items = self.list_things_at(agent.location, tclass=student)#list the thing at agent's c
urrent pos if it's
            #student then smile
            if len(items) != 0: #if its not empty
                if agent.smile(items[0]): #check if the thing is actully matching to the action set
for smile
                    print('{} smiled at {} at location: {}'#agent smiles on the stundents
                          .format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))
                    self.delete_thing(items[0]) #we delete the block coz it's already performed
                    agent.performance+=100

        elif action == "assignment": #if action is assignment
            items = self.list_things_at(agent.location, tclass=canvas)#look that current pos should
have canvas in it
            if len(items) != 0: # if the list is not empty and has object canvas
                if agent.assignment(items[0]):
                    print('{} gave assignment on {} at location: {}' #we upload assignment on
canvas
                          .format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))
                    self.delete_thing(items[0]) #we delete the block coz it's already performed
                    agent.performance-=100

        elif action == "meeting": #if action is assignment
            items = self.list_things_at(agent.location, tclass=meeting)#look that current pos shoul
d have meeting in it
            if len(items) != 0: # if the list is not empty and has object meeting
                if agent.meeting(items[0]): #we check if the thing is meeting
                    print('{} was stuck in a {} at location: {}'
                          .format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))
                    self.delete_thing(items[0]) #we delete the block coz it's already performed
                    #now he has to move back 2 steps
```

```
                    print('{} was forced to move back 2 steps now he is at location: {}'.format(str
(agent)[1:-1], agent.location-2))
                    agent.moveup(2)

        elif action == "free_time": #if action is assignment
            items = self.list_things_at(agent.location, tclass=free_time)#look that current pos sho
uld have free_time in it
            if len(items) != 0: # if the list is not empty
                if agent.free_time(items[0]): #and has the first instance of the list as free_time
                    print('{} faced {} at location: {} so he will speed it up 2 blocks  '
                          .format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))
                    self.delete_thing(items[0]) #we delete the block coz it's already performed
                    print('{} was forced to move ahead 2 steps now at location: {}'.format(str(agen
t)[1:-1], agent.location+2))
                    agent.movedown(2)


    def is_done(self): #this function defines our performance if its true we are finished, we can
rest
        '''By default, we're done when we can't find a live agent,
        but to prevent killing our awesome prof, we will stop before itself - when there is no
more objects or blocks'''
        #any retuns if one condition is true
        no_blocks = not any(isinstance(thing, student) or isinstance(thing, canvas) or isinstance(t
hing, meeting) or isinstance(thing, free_time) for thing in self.things)
        #if blocks is not in things then the above statement will be true else false
        dead_agents = not any(agent.is_alive() for agent in self.agents)
        #if any agent is dead dead_Agent=True
        return dead_agents or no_blocks #return the status oring both and if either is true just
return true meaning we are done


class oriley(Agent):
    location = [0,1] # change location to a 2d value
    performance=0

    def moveup(self,val):
        self.location[1] -= val#change location to be -1 now

    def movedown(self,val): #change location to be +1 now
        self.location[1] += val

    def smile(self, thing):
        '''returns True upon success or False otherwise'''
        if isinstance(thing, student):
            return True
        return False

    def assignment(self, thing):
        ''' returns True upon success or False otherwise'''
        if isinstance(thing, canvas):
            return True
        return False

    def meeting(self, thing):
        ''' returns True upon success or False otherwise'''
        if isinstance(thing, meeting):
            return True
        return False

    def free_time(self, thing):
        ''' returns True upon success or False otherwise'''
        if isinstance(thing, free_time):
            return True
        return False
```

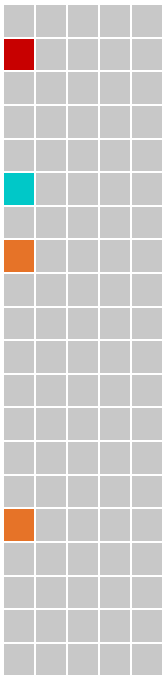
Now let's test this new park with our same dog, food and water. We color our dog with a nice red and mark food and water with orange and blue respectively.

In [76]:

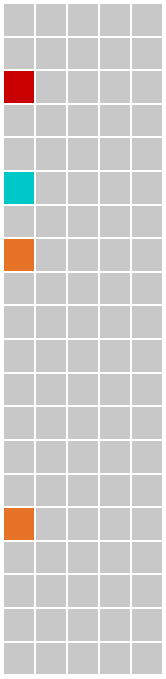
```
college = cit2d(5,20, color={'oriley': (200,0,0), 'student': (0, 200, 200), 'canvas': (230, 115, 40)
, "meeting":(110,50,100),"free_time":(50,100,200)}) # park width is set to 5, and height to 20
person = oriley(program)
dogfood = student()
```

```
water = canvas()
college.add_thing(person, [0,1])
college.add_thing(dogfood, [0,5])
college.add_thing(water, [0,7])
morewater = canvas()
college.add_thing(morewater, [0,15])
print("oriley starts at (1,1) facing downwards, lets see if he can find any blocks")
college.run(20)
```

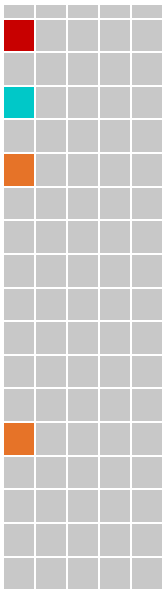
```
oriley starts at (1,1) facing downwards, lets see if he can find any blocks
```



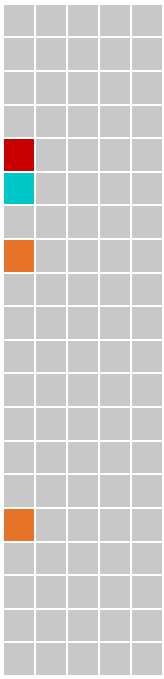
```
oriley decided to move down at location: 2
```



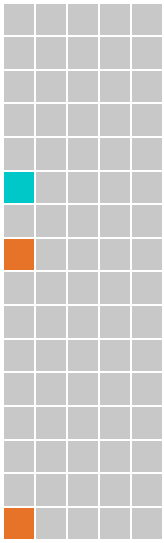
```
oriley decided to move down at location: 3
```



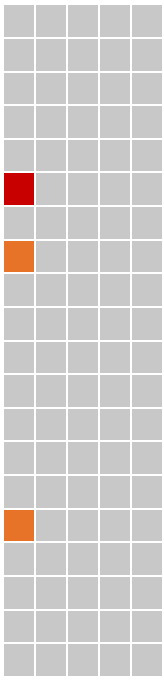
oriley decided to move down at location: 4



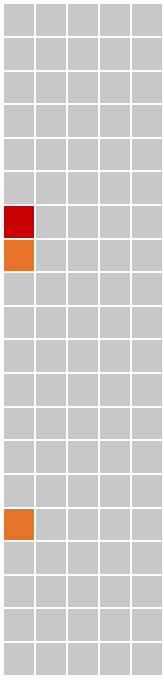
oriley decided to move down at location: 5



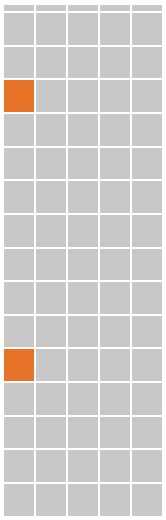
oriley smiled at student at location: [0, 5]



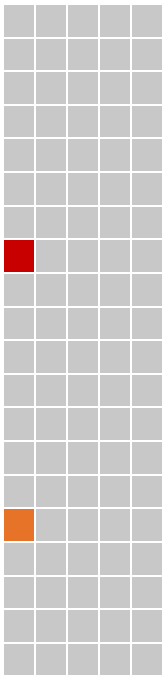
oriley decided to move down at location: 6



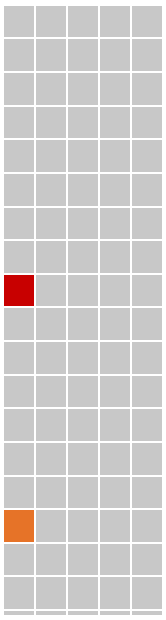
oriley decided to move down at location: 7



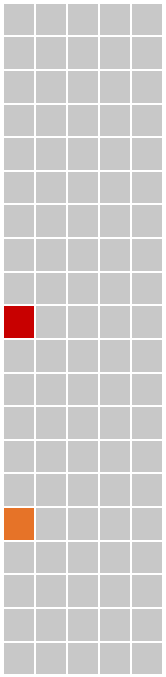
oriley gave assignment on canvas at location: [0, 7]



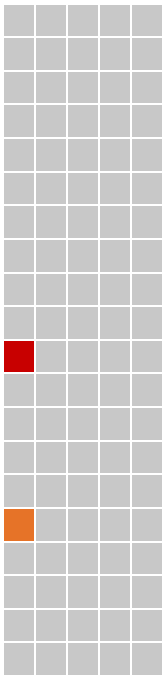
oriley decided to move down at location: 8



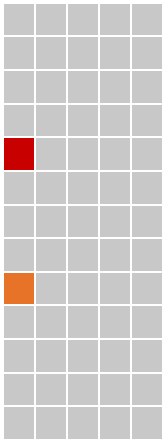
oriley decided to move down at location: 9



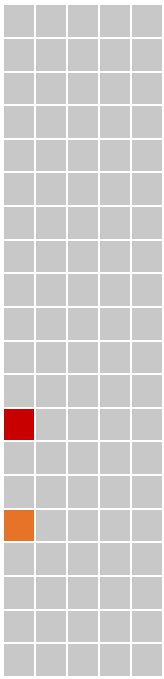
oriley decided to move down at location: 10



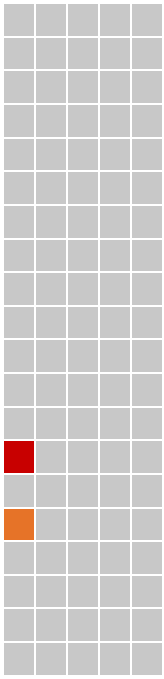
oriley decided to move down at location: 11



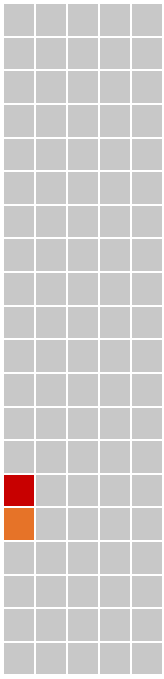
oriley decided to move down at location: 12



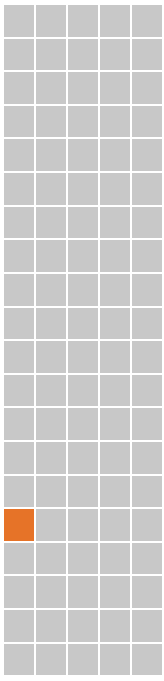
oriley decided to move down at location: 13



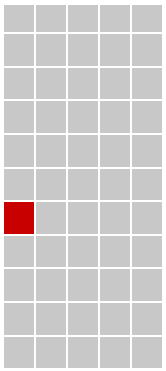
```
oriley decided to move down at location: 14
```

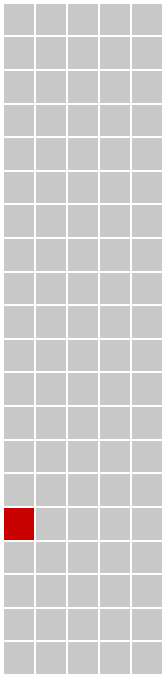


```
oriley decided to move down at location: 15
```



```
oriley gave assignment on canvas at location: [0, 15]
```





Adding some graphics was a good idea! We immediately see that the code works, but our oriley doesn't make any use of the 2 dimensional space available to him. Let's make him more energetic so that he turns and moves forward, instead of always moving down. In doing so, we'll also need to make some changes to our environment to be able to handle this extra motion.

PROGRAM - Energeticoriley

Let's make our oriley turn or move forwards at random these are 2 different tasks explicitly- except when he's at the edge of our env which is cit - in which case we make him change his direction explicitly by turning to avoid trying to leave the cit. However, our oriley cant forsee because we didn't had any more functionality so he wouldn't know which way to turn - he'd just have to try arbitrarily.

Percept:	Feel Food	Feel Water			Feel Nothing
Action:	eat	drink	**Remember being at Edge :**	At Edge	Not at Edge
			Action :	Turn Left / Turn Right (50% - 50% chance)	Turn Left / Turn Right / Move Forward (25% - 25% - 50% chance)

how does our code actually works ? and how does PEAS stand in this example ?

the first thing that happens when we run the code is it initializes the environment and our agent at the set location

after that it takes my current agent's location

then calls the function percept() function in the environment class, this is basically sensing what is present in the block It's the "S" in the PEAS approach.

Running percepts return the things present at my current location of the agent

after the location we will then want to see which thing am I holding ,

so we will match the thing by all sorts of things that we know so far by calling the program() function in our agents class

the program function maps 'thing' holding into all sorts of things that i know so far and then returns the action string that has to be performed if thing is an instance of some other thing i.e. predefined.

action string can be something like "turn left" or "move forward" etc depending upon the task

then we call the execute action in the environment class to make that action happen by giving the action string, this action will change the location or points whatever we want so in our case if this return action string is smile then we match this action string to the corresponding predefined strings in the function execute action.

after it matches we will again check that the block is the block where we want to make smile action perfomed which is a student block.

if everything goes well, we will make the action move on agent , this is the actutor "A" in the peas approach and after performing this action my Performance "P" will change and some time the environment "E" can also change as in the case of meeting and free time where my agent moves ahead or back in the environment.

if my action is give smile or give assignment it will not change the environment but only the performance "P".

there can also be a case that affects both the performance and environment.

interesting thing to note here is indirectly changing the position also affects my agents performance as every move costs him some -ve points so say he is stuck in meeting block where he is pushed back 2 blocks this might result him to move extra 2 blocks to acheive his goal meaning he will get extra -2 points since he landed on meeting block, otherwise he would have done fine. thus we can say Environment change may or may not alter performance.

There can also be some tasks that clearly changes performance and environment.

say suppose oriley eats a bonus cherry that makes him move 2 blocks at once instead of just 1 and not costing him any -ve points to move, In this hypothetical example oriley can get improved score and environment also changes.

At the end we check the termination stage by calling is_done() function if this returns true we will stop and end the runs.

but what can make this function return False , If our agent oriley he dies or we are out of the blocks meaning he has accomplished or had all blocks present in the environment. This will eventually result in termination stage.

Explaination of Code Functions:

so what are the classes present in the code:

- Environment Class: being inherited by cit2D class
- Agent Class: being inherited by EnergeticOriley class

lets define all the methods and variables in the Energetic oriley (Agent) class:

- Location: defines the current location if our agent.
- direction: defines the current direction that my agent is facing.
- moveforward(): moves oriley one step ahead in the direction he is facing.
- movebackward(: moves oriley one step back in the opposite direction he is facing, but doesn't change his current direction
- turn(): changes the direction to opposite side basically flips him 180 degrees.
- smile(): given a thing checks if the thing is an instance of student.this will return False if the thing is not student.
- assignment(): given a thing checks if the thing is an instance of canvas.this will return False if the thing is not canvas.
- free_time(): given a thing checks if the thing is an instance of free_time.this will return False if the thing is not free_time.
- meeting(): given a thing checks if the thing is an instance of meeting.this will return False if the thing is not meeting.
- program(): given all the percepts(things that my agent is holding), returns the action string corresponding to the thing that is present. This action string will be later on used by the **Environment** to execute actions.

In [77]:

```
import random
from agents import *

class Energeticoriley(Agent):
    """4th execution"""
    location = [0,1] #set starting location
    direction = Direction("down") #tells which direction to go set it to down

    def moveforward(self, success=True):#just move one block in the current direction
```

```
        '''moveforward possible only if success (i.e. valid destination location)'''
        if not success:#dont do anything if success is not given
            return
        if self.direction.direction == Direction.R: #if the directio is right move one point
            self.location[0] += 1
        elif self.direction.direction == Direction.L:#if the directio is left move one point
            self.location[0] -= 1
        elif self.direction.direction == Direction.D: ##if the directio is down move one point
            self.location[1] += 1
        elif self.direction.direction == Direction.U: #if the directio is up move one point
            self.location[1] -= 1

    def movebackward(self, success=True):#just move one block back in the current direction
        '''moveforward possible only if success (i.e. valid destination location)'''
        if not success:#dont do anything if success is not given
            return
        if self.direction.direction == Direction.R: #if the direction is right move one point
            self.location[0] -= 1 #substract it from it or simply flipping the location 1 block whi
ch it was supposed to do
        elif self.direction.direction == Direction.L:#if the direction is left move one point
            self.location[0] += 1 #
        elif self.direction.direction == Direction.D: ##if the direction is down move one point
            self.location[1] -= 1
        elif self.direction.direction == Direction.U: #if the direction is up move one point
            self.location[1] += 1


    def turn(self, d): #makes him flip the direction
        self.direction = self.direction + d

    def smile(self, thing):
        '''returns True upon success or False otherwise
        checks the block and returns if the block is a smile wala block'''
        if isinstance(thing, student):
            return True
        return False

    def assignment(self, thing):
        ''' returns True upon success or False otherwise'''
        if isinstance(thing, canvas):
            return True
        return False

    def meeting(self, thing):
        ''' returns True upon success or False otherwise'''
        if isinstance(thing, meeting):
            return True
        return False

    def free_time(self, thing):
        ''' returns True upon success or False otherwise'''
        if isinstance(thing, free_time):
            return True
        return False
        #if you want to add another thing just add another function name with thing tasknand check
if its
        #an instance of the thing

def program(percepts):
    """2- what are these things
    get an action string for these things"""
    '''Returns an action based on it's percepts'''
    for p in percepts:
        if isinstance(p, student):
            return 'smile'
        elif isinstance(p, canvas):
            return 'assignment'
        elif isinstance(p, meeting):
            return 'meeting'
        elif isinstance(p, free_time):
            return 'free_time'
        if isinstance(p,Bump): # then check if you are at an edge and have to turn
            turn = False
            choice = random.choice((1,2));
        else:
            choice = random.choice((1,2,3,4)) # 1-right, 2-left, others-forward #for the moment res
rict
```

```
            #to just one possible change that is 4 plus directions
    if choice == 1:
        return 'turnright'
    elif choice == 2:
        return 'turnleft'
    else:
        return 'moveforward' #if nothing works just go with the move fwd command
```

lets define all the methods and variables in the Energetic oriley (Agent) class:

- percept(): Returns all the things that my agent is holding. takes the agent class.
- execute_action(): Performs an action based on the action string generated by the program function in the agent()

  *if this string is smile : +100 points and delete the student block (Thing)

  *if this string is assignment: -10 points and delete the canvas block (Thing)

  *if this string is meeting: push my agent 1 step back in the same run, dont cost him any penalty, delete the meeting block(thing)

  *if this string is free_time: push my agent one step ahead in the same run, but this move still costs him movement cost it just takes no extra run.

  *if this string is move_backward he will move back one location, costs him nothing.

  *if this string is move_forward then he will move one block ahead, still costs him -1 point.

- is_done():returns True if we are out of blocks or agent died anywhere. basically to know when we want the game to stop

reflexive oriley continued

we need to change our environment such that it prevents the agent from crossing it from the boundry

we dont want oriley to go out of cit 😂

In [78]:

```
class student(Thing):#all these are blocks placed in env hence have to be things
    pass

class canvas(Thing):#all these are blocks placed in env hence have to be things
    pass

class meeting(Thing):#all these are blocks placed in env hence have to be things
    pass

class free_time(Thing):#all these are blocks placed in env hence have to be things
    pass

class Bump(Thing):#all these are blocks placed in env hence have to be things
    pass
class cit2D(GraphicEnvironment):

    def percept(self, agent):
        """1- list what are the things at agent's location"""
        '''return a list of things that are in our agent's location'''
        things = self.list_things_at(agent.location)
        loc = copy.deepcopy(agent.location) # find out the target location
        #Check if agent is about to bump into a wall
        if agent.direction.direction == Direction.R:#for right wall the agent will go 1 unit
            loc[0] += 1
        elif agent.direction.direction == Direction.L:
            loc[0] -= 1
        elif agent.direction.direction == Direction.D:
            loc[1] += 1
        elif agent.direction.direction == Direction.U:
            loc[1] -= 1
        if not self.is_inbounds(loc):
            things.append(Bump())
        return things
```

```
    def execute_action(self, agent, action):
        """3"""
        '''changes the state of the environment based on what the agent does.'''
        if action == 'turnright':
            print('{} decided to {} at location: {}'.format(str(agent)[1:-1], action,
agent.location))
            agent.turn(Direction.R)

        elif action == 'turnleft':
            print('{} decided to {} at location: {}'.format(str(agent)[1:-1], action,
agent.location))
            agent.turn(Direction.L)

        elif action == 'moveforward':
            print('{} decided to move {}wards at location: {}'.format(str(agent)[1:-1],
agent.direction.direction, agent.location))
            agent.performance-=1
            agent.moveforward()

        elif action == 'movebackward':
            print('{} decided to move {}wards at location: {}'.format(str(agent)[1:-1],
agent.direction.direction, agent.location))
            #does not cost him performance to go down
            agent.movebackward()

        elif action == "smile":#if action is eating
            items = self.list_things_at(agent.location, tclass=student)#list the thing at agent's c
urrent pos if it's
            #student then smile
            if len(items) != 0: #if its not empty
                if agent.smile(items[0]): #check if the thing is actully matching to the action set
for smile
                    print('{} smiled at {} at location: {}'#agent smiles on the stundents
                          .format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))
                    self.delete_thing(items[0]) #we delete the block coz it's already performed
                    agent.performance+=100

        elif action == "assignment": #if action is assignment
            items = self.list_things_at(agent.location, tclass=canvas)#look that current pos should
have canvas in it
            if len(items) != 0: # if the list is not empty and has object canvas
                if agent.assignment(items[0]):
                    print('{} gave assignment on {} at location: {}' #we upload assignment on
canvas
                          .format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))
                    self.delete_thing(items[0]) #we delete the block coz it's already performed
                    agent.performance-=10

        elif action == "meeting": #if action is assignment
            items = self.list_things_at(agent.location, tclass=meeting)#look that current pos shoul
d have meeting in it
            if len(items) != 0: # if the list is not empty and has object meeting
                if agent.meeting(items[0]): #we check if the thing is meeting
                    print('{} was stuck in a {} at location: {}'
                          .format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))
                    self.delete_thing(items[0]) #we delete the block coz it's already performed
                    #now he has to move back 2 steps
                    agent.movebackward()
                    print('{} was forced to move back 1 steps now he is at location: {}'.format(str
(agent)[1:-1], agent.location))


        elif action == "free_time": #if action is assignment
            items = self.list_things_at(agent.location, tclass=free_time)#look that current pos sho
uld have free_time in it
            if len(items) != 0: # if the list is not empty
                if agent.free_time(items[0]): #and has the first instance of the list as free_time
                    print('{} faced {} at location: {} so he will speed it up 1 blocks  '
                          .format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))
                    self.delete_thing(items[0]) #we delete the block coz it's already performed
                    agent.performance+=1
                    agent.moveforward()
                    print('{} was forced to move ahead 2 steps now at location: {}'.format(str(agen
t)[1:-1], agent.location))

    def is_done(self): #this function defines our performance if its true we are finished, we can
```

```
rest
        '''By default, we're done when we can't find a live agent,
        but to prevent killing our awesome prof, we will stop before itself - when there is no
more objects or blocks'''
        #any retuns if one condition is true
        no_blocks = not any(isinstance(thing, student) or isinstance(thing, canvas) or isinstance(t
hing, meeting) or isinstance(thing, free_time) for thing in self.things)
        #if blocks is not in things then the above statement will be true else false
        dead_agents = not any(agent.is_alive() for agent in self.agents)
        #if any agent is dead dead_Agent=True
        return dead_agents or no_blocks #return the status oring both and if either is true just
return true meaning we are done
```

this block of code will replicate agents ipynb

how to control boundry conditions ?

we will make sure that the oriley never crosses boundry by not placing any move forward block- free time or move backward block- meeting

over the edges or the boundry because this will push him 1 block and can force him to go out of the environment

I have also added the random environment for every run by using random library in python

what do all the colours mean in the example :

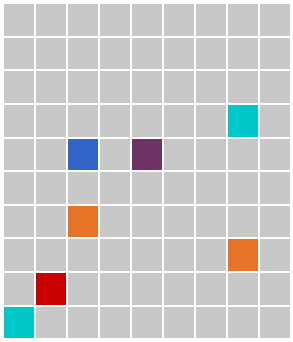
Colors:	Red	orange	light blue	navy blue	purple
what do these colors represent:	My agent: Oriley	Canvas	student	free_time block	meeting block

In [79]:

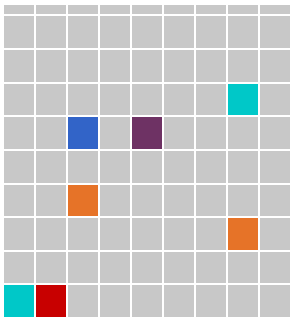
```
rows=random.randint(6,12) #rows- the total width horizontal
cols=random.randint(6,12)#cols- total depth vertical
college = cit2D(rows,cols, color={'Energeticoriley': (200,0,0) , 'student': (0, 200, 200), 'canvas'
: (230, 115, 40), "meeting":(110,50,100),"free_time":(50,100,200)})
#oriley- red #orange-canvas #light blue-student # purple-meeting #navy blue =free_time
oriley = Energeticoriley(program)
student1 = student()
canv1 = canvas()
m1= meeting()
f1=free_time()

blocks=[]

while True:
    oriley_location=[random.randint(0,rows-1),random.randint(0,cols-1)] #students can appear any wh
ere
    #print(oriley_location)
    if (oriley_location in blocks):
        continue
    else:
        blocks.append(oriley_location)
        break
while True:
    student1_location=[random.randint(0,rows-1),random.randint(0,cols-1)] #students can appear any
where
    if (student1_location in blocks):
        continue
    else:
        blocks.append(student1_location)
        break

while True:
    canv1_location=[random.randint(0,rows-1),random.randint(0,cols-1)] #students can appear any whe
re
    if (canv1_location in blocks):
        continue
    else:
        blocks.append(canv1_location)
        break
```

```
while True:
    m1_location=[random.randint(1,rows-2),random.randint(1,cols-2)]#students can appear any where
    if (m1_location in blocks):
        continue
    else:
        blocks.append(m1_location)
        break

while True:
    f1_location=[random.randint(1,rows-2),random.randint(1,cols-2)]#students can appear any where
    if (f1_location in blocks):
        continue
    else:
        blocks.append(f1_location)
        break

print(blocks)
college.add_thing(oriley, blocks[0])
college.add_thing(student1, blocks[1])
college.add_thing(canv1, blocks[2])
college.add_thing(m1,blocks[3]) #you cant place meeting block on the edge
college.add_thing(f1,blocks[4]) #you cant add free time near the boundry it will place the bot out
of the screen

student2 = student() #add another instance of student
canv2 = canvas()     #add another canvas block

while True:
    canv2_location=[random.randint(0,rows-1),random.randint(0,cols-1)] #students can appear any whe
re
    if (canv2_location in blocks):
        continue
    else:
        blocks.append(canv2_location)
        break

while True:
    student2_location=[random.randint(0,rows-1),random.randint(0,cols-1)] #students can appear any
where
    if (student2_location in blocks):
        continue
    else:
        blocks.append(student2_location)
        break
print("blocks:{}".format(blocks))
college.add_thing(student2,blocks[5])
college.add_thing(canv2, blocks[6])
print("oriley started at {}, facing down. Let's see if he found any student or canvas!".format(ori
ley_location))
college.run(5)
```

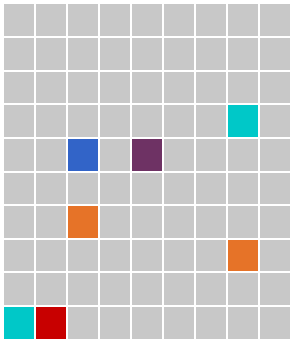
```
[[1, 8], [0, 9], [7, 7], [4, 4], [2, 4]]
blocks:[[1, 8], [0, 9], [7, 7], [4, 4], [2, 4], [7, 3], [2, 6]]
oriley started at [1, 8], facing down. Let's see if he found any student or canvas!
```



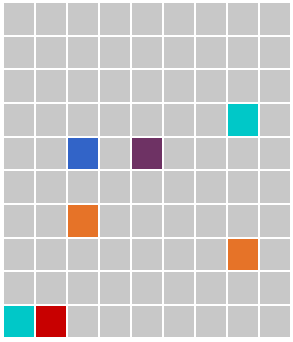
```
Energeticoriley decided to move downwards at location: [1, 8]
```



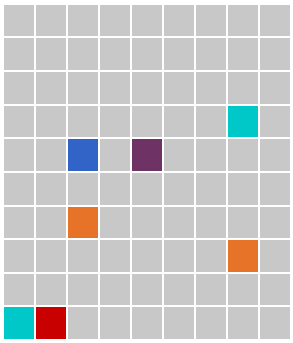
Energeticoriley decided to turnleft at location: [1, 9]



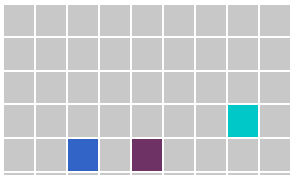
Energeticoriley decided to turnright at location: [1, 9]



Energeticoriley decided to turnright at location: [1, 9]



Energeticoriley decided to turnright at location: [1, 9]

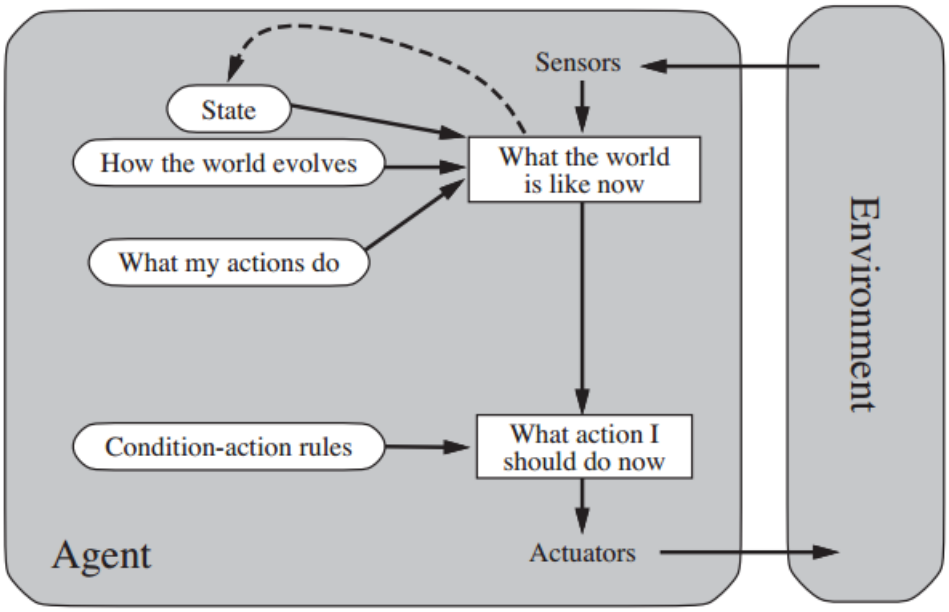


In [80]:

```
print("final performance score out of the simulation is {}".format(oriley.performance))
```

```
final performance score out of the simulation is -1
```

model based agent of oriley :)



In model based approach oriley will have to look all the previous instances and make his decisions based on that so in my case he will look into history of locations he has travelled and tries to calculate the new move's location.

He tries to simulate the location of the new move and if this new move is present in the history of all the moves that he has travelled then the bot will not make any move in this direction instead he will again pick a random move.

this ensures that oriley does not land on the same block again.

for example if he is on block 1 say and he moved to block 2 now his random move will make him move back to block 1 he will not move to the same block again, so he will pick all the other random moves apart from block 1.

what is the pseudo code of model based agent ? and how do we implement it in our model ?

1. oriley will be initialized with a history variable an array of elements
2. he picks up a random direction and then waits for the move forward command
3. if he gets the move forward command he doesn't directly moves he calls a simulated moveforward function which returns him the location value where he will land if he makes that move
4. If this location value is present in the history meaning he has already travelled this block he will not move what he was originally planning to , this will waste his one run but will not cost him to decrease the performance and further waste his time. If this move simulted returns him a block that has never been present in the history of all the possible moves he will make that move and also add immideatly to the history of moves that he has travelled.
5. If he cannot move or change his location to the moves he has already made he can be in a dead lock situation where he can't get out and this is the problem of this type of agent.
6. at the end after moving he will also check for the termination stage where he will check if all the blocks are taken or somehow he dies(of course he will not die because we haven't added any block that kills him, because we love him ❤□). those are the two cases that can stop the program and then we can print his performance score.
7. he goes back to step 2, if step 6 is not acheived.

In [18]:

```
import random
from agents import *
history=[]

class Energeticoriley(Agent):
    """4th execution"""
    location = [0,1] #set starting location
    direction = Direction("down") #tells which direction to go set it to down

    def simulate_location(self):
        """this block simultes executed action by taking execution string and return the possibile
location
        after our agent makes that decision, this is just a simulation it will not change agent's
real location
        we will later on check if the simulated move is present in history or not to make our real
action work"""


    def simulated_mf(self):
        """this actually replicates the simulated move forward location
        returns the simulated location"""
        temp_location=copy.deepcopy(self.location)
        if self.direction.direction == Direction.R: #check the current direction and make updates
in temp_location
            temp_location[0] += 1
        elif self.direction.direction == Direction.L:#if the directio is left move one point
            temp_location[0] -= 1
        elif self.direction.direction == Direction.D: ##if the directio is down move one point
            temp_location[1] += 1
        elif self.direction.direction == Direction.U: #if the directio is up move one point
            temp_location[1] -= 1
        return temp_location

    def simulated_mb(self):
        """this actually replicates the simulated move forward location
        returns the simulated location"""
        temp_location=copy.deepcopy(self.location)
        if self.direction.direction == Direction.R: #check the current direction and make updates
in temp_location
            temp_location[0] -= 1
        elif self.direction.direction == Direction.L:#if the directio is left move one point
            temp_location[0] += 1
        elif self.direction.direction == Direction.D: ##if the directio is down move one point
            temp_location[1] -= 1
        elif self.direction.direction == Direction.U: #if the directio is up move one point
            temp_location[1] += 1
        return temp_location


    def moveforward(self, success=True):#just move one block in the current direction
        '''moveforward possible only if success (i.e. valid destination location)'''
        if not success:#dont do anything if success is not given
            return
        if self.direction.direction == Direction.R: #if the directio is right move one point
            self.location[0] += 1
        elif self.direction.direction == Direction.L:#if the directio is left move one point
            self.location[0] -= 1
        elif self.direction.direction == Direction.D: ##if the directio is down move one point
            self.location[1] += 1
        elif self.direction.direction == Direction.U: #if the directio is up move one point
            self.location[1] -= 1
        print("kuch bhi :",history)
        print(self.location)
        history.append(self.location)

    def movebackward(self, success=True):#just move one block back in the current direction
        '''moveforward possible only if success (i.e. valid destination location)'''
        if not success:#dont do anything if success is not given
            return
        if self.direction.direction == Direction.R: #if the direction is right move one point
            self.location[0] -= 1 #substract it from it or simply flipping the location 1 block whi
ch it was supposed to do
        elif self.direction.direction == Direction.L:#if the direction is left move one point
            self.location[0] += 1 #
        elif self.direction.direction == Direction.D: ##if the direction is down move one point
```

```
            self.location[1] -= 1
        elif self.direction.direction == Direction.U: #if the direction is up move one point
            self.location[1] += 1
        history.append(self.location)


    def turn(self, d): #makes him flip the direction
        self.direction = self.direction + d

    def smile(self, thing):
        '''returns True upon success or False otherwise
        checks the block and returns if the block is a smile wala block'''
        if isinstance(thing, student):
            return True
        return False

    def assignment(self, thing):
        ''' returns True upon success or False otherwise'''
        if isinstance(thing, canvas):
            return True
        return False

    def meeting(self, thing):
        ''' returns True upon success or False otherwise'''
        if isinstance(thing, meeting):
            return True
        return False

    def free_time(self, thing):
        ''' returns True upon success or False otherwise'''
        if isinstance(thing, free_time):
            return True
        return False
        #if you want to add another thing just add another function name with thing tasknand check
if its
        #an instance of the thing

def program(percepts):
    """2- what are these things
    get an action string for these things"""
    '''Returns an action based on it's percepts'''
    for p in percepts:
        if isinstance(p, student):
            return 'smile'
        elif isinstance(p, canvas):
            return 'assignment'
        elif isinstance(p, meeting):
            return 'meeting'
        elif isinstance(p, free_time):
            return 'free_time'
        if isinstance(p,Bump): # then check if you are at an edge and have to turn
            turn = False
            choice = random.choice((1,2));
        else:
            choice = random.choice((1,2,3,4)) # 1-right, 2-left, others-forward #for the moment res
rict
            #to just one possible change that is 4 plus directions
    if choice == 1:
        return 'turnright'
    elif choice == 2:
        return 'turnleft'
    else:
        return 'moveforward' #if nothing works just go with the move fwd command
```

In [19]:

```
class cit2D(GraphicEnvironment):

    history.append(Energeticoriley.location)

    def percept(self, agent):
        """1- list what are the things at agent's location"""
        '''return a list of things that are in our agent's location'''
        things = self.list_things_at(agent.location)
        loc = copy.deepcopy(agent.location) # find out the target location
```

```
        #Check if agent is about to bump into a wall
        if agent.direction.direction == Direction.R:#for right wall the agent will go 1 unit
            loc[0] += 1
        elif agent.direction.direction == Direction.L:
            loc[0] -= 1
        elif agent.direction.direction == Direction.D:
            loc[1] += 1
        elif agent.direction.direction == Direction.U:
            loc[1] -= 1
        if not self.is_inbounds(loc):#checks if the location is valid
            things.append(Bump())
        return things

    def execute_action(self, agent, action):
        """3"""
        '''changes the state of the environment based on what the agent does.'''
        if action == 'turnright':
            print('{} decided to {} at location: {}'.format(str(agent)[1:-1], action,
agent.location))
            agent.turn(Direction.R)

        elif action == 'turnleft':
            print('{} decided to {} at location: {}'.format(str(agent)[1:-1], action,
agent.location))
            agent.turn(Direction.L)

        elif action == 'moveforward':
            #here action simulation takes place if we get True means we can make the action happen
else we will say
            #move not made and do nothing.
            temp_location=agent.simulated_mf()
            if(temp_location not in history):
                print('{} decided to move {}wards at location: {}'.format(str(agent)[1:-1],
agent.direction.direction, agent.location))
                agent.performance-=1
                agent.moveforward()
            else:
                print("This move will let agent go on {} which he has already travelled".format(tem
p_location))

        elif action == 'movebackward':
            temp_location=agent.simulated_mb()
            if(temp_location not in history):
                print('{} decided to move {}wards at location: {}'.format(str(agent)[1:-1],
agent.direction.direction, agent.location))
                #does not cost him performance to go down
                agent.movebackward()
            else:
                print("This move will let agent go on {} which he has already travelled".format(tem
p_location))

        elif action == "smile":#if action is eating
            items = self.list_things_at(agent.location, tclass=student)#list the thing at agent's c
urrent pos if it's
            #student then smile
            if len(items) != 0: #if its not empty
                if agent.smile(items[0]): #check if the thing is actully matching to the action set
for smile
                    print('{} smiled at {} at location: {}'#agent smiles on the stundents
                          .format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))
                    self.delete_thing(items[0]) #we delete the block coz it's already performed
                    agent.performance+=100

        elif action == "assignment": #if action is assignment
            items = self.list_things_at(agent.location, tclass=canvas)#look that current pos should
have canvas in it
            if len(items) != 0: # if the list is not empty and has object canvas
                if agent.assignment(items[0]):
                    print('{} gave assignment on {} at location: {}' #we upload assignment on
canvas
                          .format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))
                    self.delete_thing(items[0]) #we delete the block coz it's already performed
                    agent.performance-=10

        elif action == "meeting": #if action is assignment
            items = self.list_things_at(agent.location, tclass=meeting)#look that current pos shoul
d have meeting in it
```

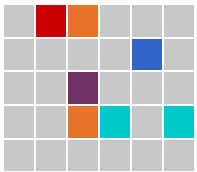
```
            if len(items) != 0: # if the list is not empty and has object meeting
                if agent.meeting(items[0]): #we check if the thing is meeting
                    print('{} was stuck in a {} at location: {}'
                          .format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))
                    self.delete_thing(items[0]) #we delete the block coz it's already performed
                    #now he has to move back 2 steps
                    temp_location=agent.simulated_mb()
                    if(temp_location not in history):
                        agent.movebackward()
                        print('{} was forced to move back 1 steps now he is at location: {}'.format
(str(agent)[1:-1], agent.location))
                    else:
                        print("This move will let agent go on {} which he has already travelled".fo
rmat(temp_location))

        elif action == "free_time": #if action is assignment
            items = self.list_things_at(agent.location, tclass=free_time)#look that current pos shc
uld have free_time in it
            if len(items) != 0: # if the list is not empty
                if agent.free_time(items[0]): #and has the first instance of the list as free_time
                    print('{} faced {} at location: {} so he will speed it up 1 blocks  '
                          .format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))
                    self.delete_thing(items[0]) #we delete the block coz it's already performed
                    temp_location=agent.simulated_mf()
                    if(temp_location not in history):
                        agent.performance+=1
                        agent.moveforward()
                        print('{} was forced to move ahead 2 steps now at location: {}'.format(str(
agent)[1:-1], agent.location))
                    else:
                        print("This move will let agent go on {} which he has already travelled".fo
rmat(temp_location))

    def is_done(self): #this function defines our performance if its true we are finished, we can
rest
        '''By default, we're done when we can't find a live agent,
        but to prevent killing our awesome prof, we will stop before itself - when there is no
more objects or blocks'''
        #any retuns if one condition is true
        no_blocks = not any(isinstance(thing, student) or isinstance(thing, canvas) or isinstance(t
hing, meeting) or isinstance(thing, free_time) for thing in self.things)
        #if blocks is not in things then the above statement will be true else false
        dead_agents = not any(agent.is_alive() for agent in self.agents)
        #if any agent is dead dead_Agent=True
        return dead_agents or no_blocks #return the status oring both and if either is true just
return true meaning we are done
```

In [20]:

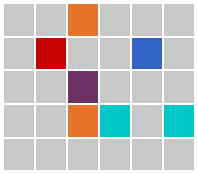
```
rows=random.randint(5,12) #rows- the total width horizontal
cols=random.randint(5,12)#cols- total depth vertical
college = cit2D(rows,cols, color={'Energeticoriley': (200,0,0) , 'student': (0, 200, 200), 'canvas'
: (230, 115, 40), "meeting":(110,50,100),"free_time":(50,100,200)})
#oriley- red #orange-canvas #light blue-student # purple-meeting #navy blue =free_time
oriley = Energeticoriley(program)
student1 = student()
canv1 = canvas()
m1= meeting()
f1=free_time()

blocks=[]

while True:
    oriley_location=[random.randint(0,rows-1),random.randint(0,cols-1)] #students can appear any wh
ere
    #print(oriley_location)
    if (oriley_location in blocks):
        continue
    else:
        blocks.append(oriley_location)
        break
while True:
    student1_location=[random.randint(0,rows-1),random.randint(0,cols-1)] #students can appear any
where
    if (student1_location in blocks):
        continue
```

```
        continue
    else:
        blocks.append(student1_location)
        break

while True:
    canv1_location=[random.randint(0,rows-1),random.randint(0,cols-1)] #students can appear any whe
re
    if (canv1_location in blocks):
        continue
    else:
        blocks.append(canv1_location)
        break

while True:
    m1_location=[random.randint(1,rows-2),random.randint(1,cols-2)]#students can appear any where
    if (m1_location in blocks):
        continue
    else:
        blocks.append(m1_location)
        break

while True:
    f1_location=[random.randint(1,rows-2),random.randint(1,cols-2)]#students can appear any where
    if (f1_location in blocks):
        continue
    else:
        blocks.append(f1_location)
        break

print(blocks)
college.add_thing(oriley, blocks[0])
college.add_thing(student1, blocks[1])
college.add_thing(canv1, blocks[2])
college.add_thing(m1,blocks[3]) #you cant place meeting block on the edge
college.add_thing(f1,blocks[4]) #you cant add free time near the boundry it will place the bot out
of the screen

student2 = student() #add another instance of student
canv2 = canvas()     #add another canvas block

while True:
    canv2_location=[random.randint(0,rows-1),random.randint(0,cols-1)] #students can appear any whe
re
    if (canv2_location in blocks):
        continue
    else:
        blocks.append(canv2_location)
        break

while True:
    student2_location=[random.randint(0,rows-1),random.randint(0,cols-1)] #students can appear any
where
    if (student2_location in blocks):
        continue
    else:
        blocks.append(student2_location)
        break
print("blocks:{}".format(blocks))
college.add_thing(student2,blocks[5])
college.add_thing(canv2, blocks[6])
print("oriley started at {}, facing down. Let's see if he found any student or canvas!".format(ori
ley_location))
college.run(5)
```

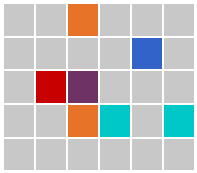
```
[[1, 0], [5, 3], [2, 0], [2, 2], [4, 1]]
blocks:[[1, 0], [5, 3], [2, 0], [2, 2], [4, 1], [3, 3], [2, 3]]
oriley started at [1, 0], facing down. Let's see if he found any student or canvas!
```



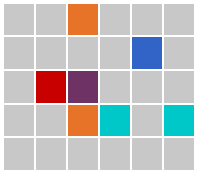
```
Energeticoriley decided to move downwards at location: [1, 0]
kuch bhi : [[0, 1]]
[1, 1]
```



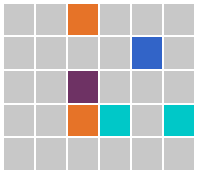
```
Energeticoriley decided to move downwards at location: [1, 1]
kuch bhi : [[0, 1], [1, 2]]
[1, 2]
```



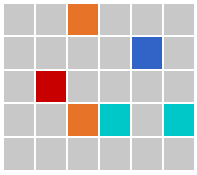
```
Energeticoriley decided to turnleft at location: [1, 2]
```



```
Energeticoriley decided to move rightwards at location: [1, 2]
kuch bhi : [[0, 1], [2, 2], [2, 2]]
[2, 2]
```



```
Energeticoriley was stuck in a meeting at location: [2, 2]
Energeticoriley was forced to move back 1 steps now he is at location: [1, 2]
```



In [21]:

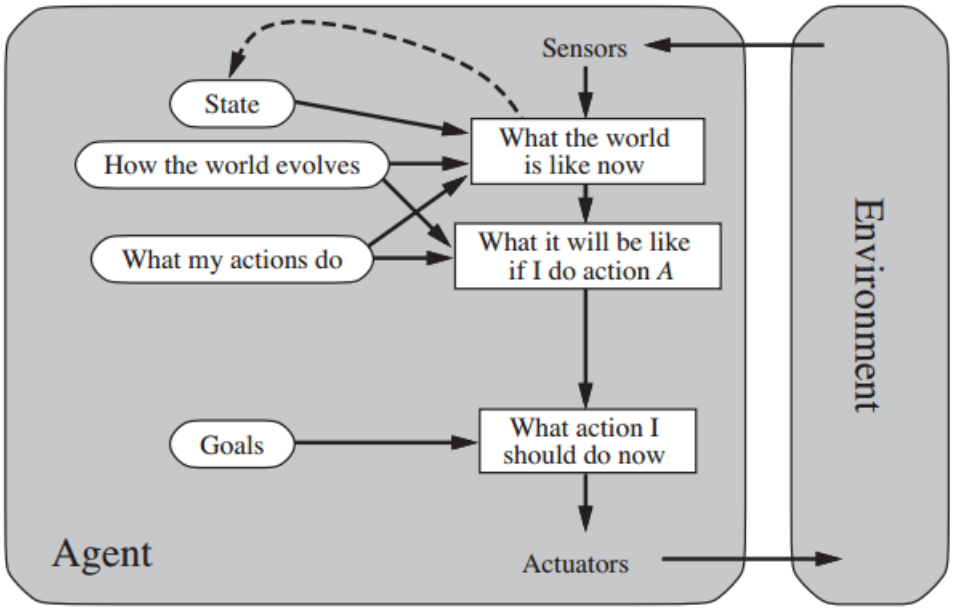
```
#notice repeated history , this is because the environment gets refreshed after every run
print(history)
del history
```

```
[[0, 1], [1, 2], [1, 2], [1, 2], [1, 2]]
```

what is a Goal based agent ? and how to implement it

so what is my GOAL and how is my agent trying to reach it's goal ?

- my GOAL here is to travel all blocks including (canvas and free_time) with minimum performance dips
- a goal based agent tries to maximize his goal in our case the goal is to cover all the block in with minimum performance drops because every move costs him -1 point.
- if you carefully see the picture below explains goal based agents but it is only slightly different from the one shown in model based , the only difference here is that he has an additional block asking him what will my action do if i take this move, previosuly he was asking the same question but only with respect to not travelling the same block again that wastes his time. Now he is asking that same question to improve his performance which is directly linked to his goal of travelling through all block with least performance dips



So how can he do that ?

it is pretty simple if the simulated move that we were making him in the model based approach, if it lands him on a meeting block and canvas block he will try to avoid it aditionally apart from the history of events that he has seen.

and rest every thing will be normal as in model based agent.

However he still needs to finish all the blocks to win hence he is less priorotizing the neagtive blocks such as meeting and canvas because both of them will ruin his performance so he will and only will get the blocks if he has taken all the other blocks so we also need to tell the agent how many blocks are totally placed so if he is out of smiles and free_time block he will start taking the penalty blocks aswell because attending meetings and giving assignment on canvas is still a mandatory task but far less important than making people happy.

By this way he is pushing his priority to student block that gives him +100 pts and the free_time block where he can move 1 step ahead and take lectures timely□

what is my pseudo code for Goal based ?

1. Start at random location with no history or 1st block as history
2. make random moves until he is close to a block say(falls in 1 nearest neighbour of the block)
3. after he reaches close to his block simulated move function will return him the location along with any perks/ penalty of that block
4. if there is a perk block meaning he benefits from the block say(student block that gives him +100 points or say free_time block that moves his location 1 block ahead which saves his total run time by one unit) he will make that move.
5. if the block is a penalty block he will waste his one run and will look for another possible move set that he can make apart from the penalty block
6. at this point he has decided wether to take step 4 or step 5 he will then move to the block which the either of the above statement asks him to.
7. at the end he will look for termination stage if the termination stage is satisfied meaning is_done() function returns true, he will stop and we can print his performance

Problem with this type of agent suppose he has travelled all sorts of block before and his only resort left is to face the penalty block and move ahead he will still try all the possible combinations of the moves he can take but it's actually None because the moves that he will pick will either lead him to penalty or a previous block.

so how to determine what do to now ?

- if he is stuck near penalty block for too long he has to decide wether he needs to travel back in history overriding the history or he wants to face the penalty and then try again.
- since our bot is short sighted meaning he can only see short term goals that can lead him to long term penalties, then he will actually compare the penalties, say moving on a meeting block wlll reduce his score by one point so he can possibly take that move but if the choice is between the canvas block where he loses -10 points he would prefer to go back in history than to lose -10 points.

To make a very optimized goal based agent we have to take alot of things into account such has the degree of penalty, track of moves, and sometimes the probability of the block to be a perk based or penalty based block and then deciding what to do.

Also we need to take into the dead lock conditions where the bot will not be able to decide anything because either the move is travelled or leads him to penalty , then if he decides to travels back to the history of blocks that he has already travelled what is his long term loss.

it could be that moving into a penalty block and facing a -1 point in score can lead him to his final block faster than he would actually do if he is pushed back in history and then he is figuring his way out by again making random moves.

So A proper GOAL based agent has to look for long term benefits instead of short term rewards, which is extremely hard to predict without knowing the Environment.

we can add slightly more functionality that makes him a little better than his previous approaches to maximize the goal but to get an optimal solution is still pretty hard to manage.

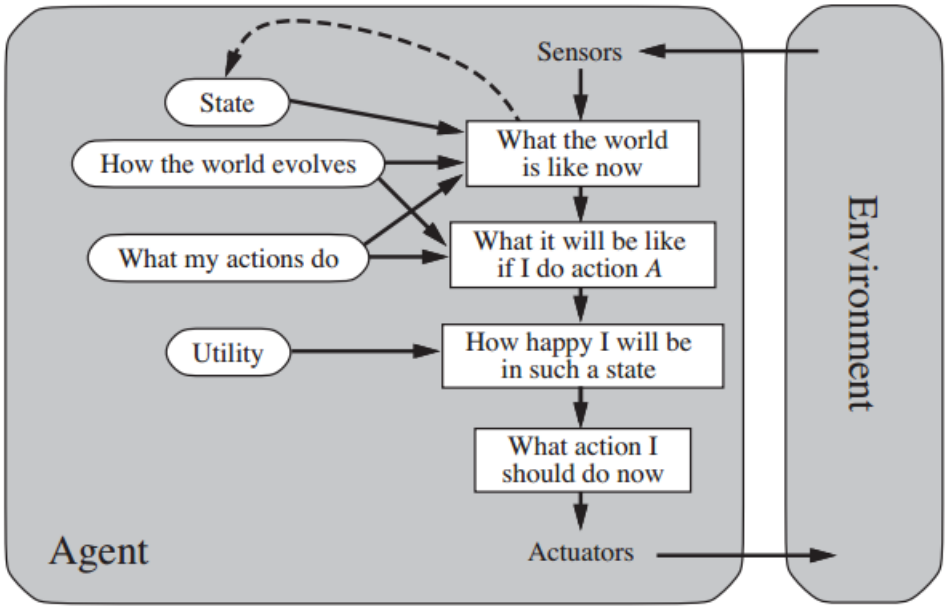
Can we do better than a Goal based agent ?

the answer to the above question is Yes. we can do better than the goal based agent. In the book Artificial Intelligence , A modern approach by Stuart Russel, It is explained that the above are not the only types of agents present.

There are two other agent types namely : utility based and learning agents

what is a utility based agent ?

The below diagram explains the working of utility based agent, as we can see it is very similar to the goal based agent but it has an extra block of how happy that my agent becomes ?



This is something which goal based lacked. In order to reach a particular goal there can be many different ways to do, but what we are interested is in a way that makes my agent the most happy.

happy is a very abstract word , how do we define happy in our problem above ?

so in the example of Oriley his goal is to cover all blocks in the environment by covering minimum performance dips.

but there can be multiple paths that can lead my agent to this goal, which one to take then ?

In the goal based agent above, I mentioned the drawback when he is stuck between two choices either to move on canvas block and get -10 points or travel back in the history of paths which will ask him to move the blocks that he has previously seen.

A goal based agent will compare both of the penalties and which ever seems less harmful to the performance state which is its goal, he will take that action.

so suppose our agent know that after moving to block canvas he will reach all the blocks that he has travelled or you can consider another example where after stepping to canvas block he can then reach the student block which will give him +100 pts. he will then prefer to take the penalty of -10 and then move to student block giving him another +100 points(so the net is +90 on his performance) rather than going back in the history of moves it has already performed and trying to find a way back to student block while avoiding canvas block.

If he decides to back in history and take the moves to reach his final block even then he has to face the canvas block because our objective is to finish all the blocks. so why go back and make repeated moves if you can take the penalty and immideatley reach the terminatation stage with net addition of +90 points.

This analysis of which punishment is better is what seperates utility based with goal based.

In very simple words if I have to explain the difference , I would say, In goal based he is simply trying to avoid penalty blocks while trying to reach his goal. In utility based he is avoiding the penalty blocks but also comparing them to what extent do these affect my overall performance.

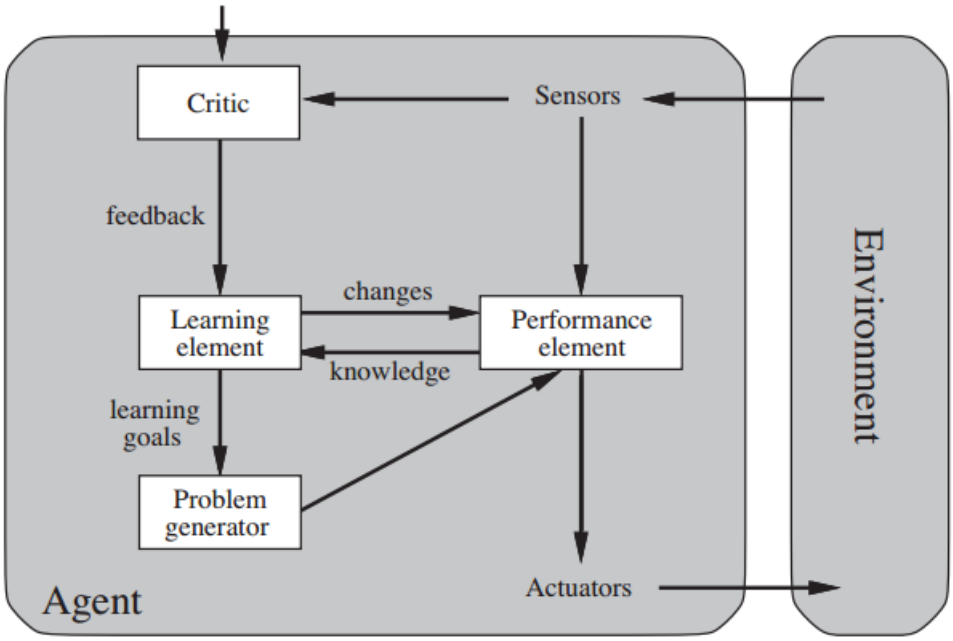
If he really has to choose only between two penalty blocks then which one to choose that helps me better reach his goal state.

below is a very funny example,that shows the utility based agent when mario is not going for the short term reward that is taking the mushroom that makes him bigger and add one extra life if he hits the wompus, he is more focussed on reach it's goal in min distance because he missed the mushroom in the first place, so he didn't go back to take the mushroom instead he moved forward and decided to choose and go on which will not increase his distance,and make him closer to his goal.

Can we do even better ?

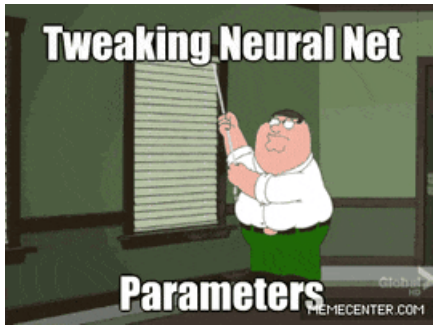
Yes, suppose after implementing the utility based agent it is somehow still stuck somewhere in a dead lock condition where he cannot go back or move further and dies. we need our agent to remember the exact configurations that he made and caused him to die. This learning parameter is added to our agents memory and can make him not fall into the same trap again.

Performance standard

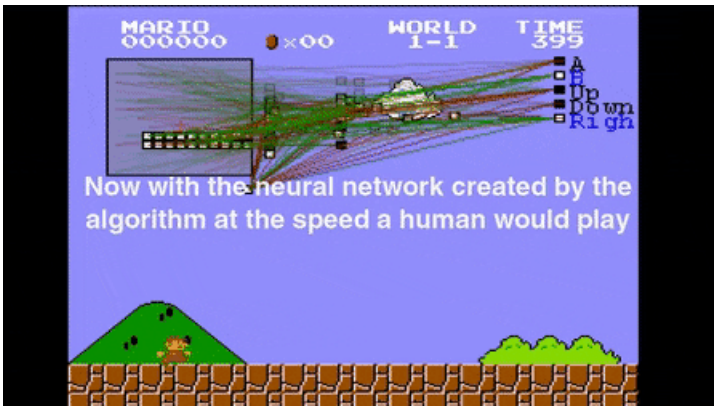


Such a model that learns from it's mistakes or looks for better path that improves it's state based on all the previous approaches it took(accounting all the parameters such as history, penalty and reward ranking, deadlock stage etc)

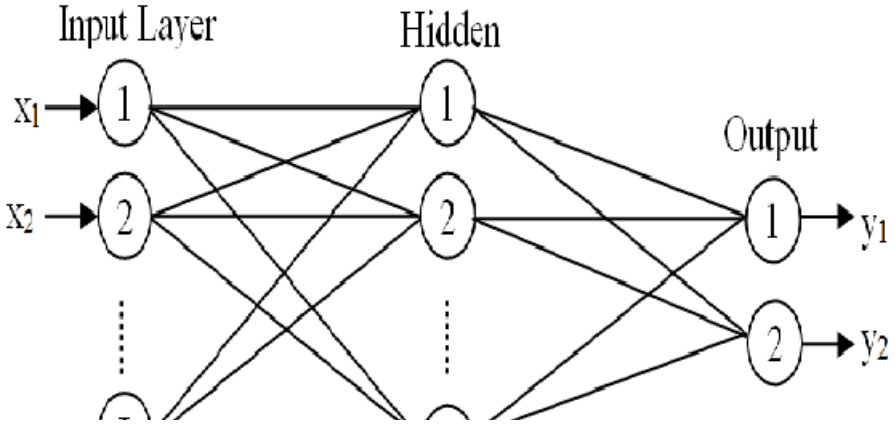
So how to do this , upon research I found the an LSTM approach with Artificial Neural Networks can help acheive the goal.

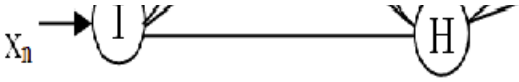


In the below gif it shows how ANN is deciding what to move in the final layer of a Neural Net by keeping memory into account.



the last layer has nodes equivalent to all the possible moves the mario can make based upon the raw environment that it takes.





the last layer does an operation is called as Softmax which is basically calculates the probability of all the events happening. That are predicted by neural nets to win the game.

Softmax function

$$P(y=j \mid \theta^{(i)}) = \frac{e^{\theta^{(i)}}}{\sum_{j=0}^{k} e^{\theta_k^{(i)}}}$$

$$\text{where } \Theta = w_0x_0 + w_1x_1 + \ldots + w_kx_k = \sum_{i=0}^{k} w_i x_i = w^T x$$

This softmax function ensures that no probability caluculates is turned to negative value and in the end after we apply the softmax we just pick the node with maximum value and turn all the other nodes off.

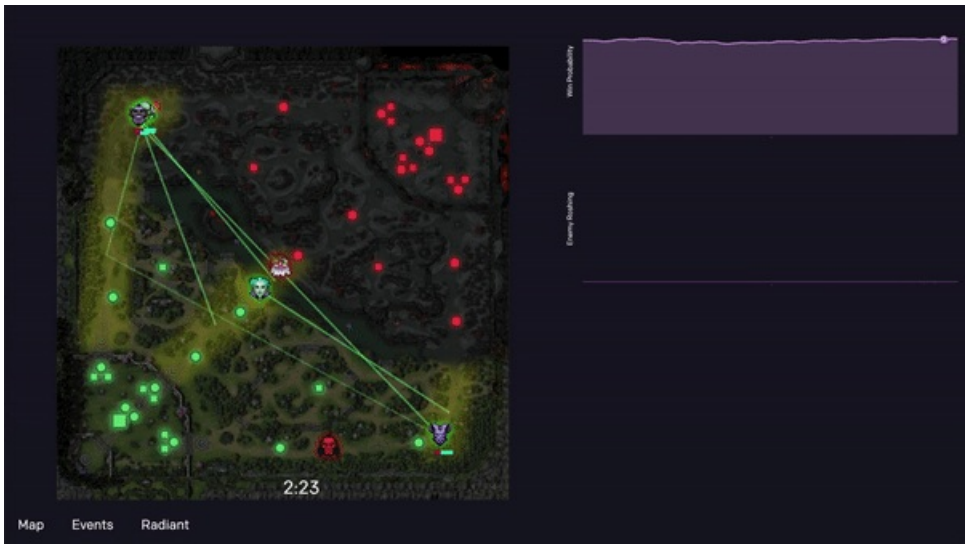
The below gif running on a CNN network that takes raw image pixels and compute the result in a node corresponding to which ever move it wants to make.

This example clearly illustrates the learning agent because it's not learning on a set of predefined rules it is learning and making it's own rules based on past experiences it has encountered.

Also it is practically impossible to write n number of rules if the agent takes many paramters do decide it's next move.

we will not go into much detail of this example as it involves alot of concepts from a Computer science domain called Deep learning and would divert us from our topic , but this is just to show what a truly intelligent agent looks like. (I hope I am able to explain the very basics of learning agent without going much in detail and still giving a flavour of it.

another very real world example is OpenAi used in the game of Dota which was able to beat all the pro players in the league. if you want to learn more of how interanlly the neural nets were trained or what is the working of the model. you can refer this video link -https://www.youtube.com/watch?v=DzzFSyzv1p0 but i will not include this in my ipython notebook because it requires alot of explaination into an alltogether different domain



2 UnInformed Search Techniques

Before we begin I would like to explain what exactly do these 3 words mean.

1. Uninformed = Not Knowing
2. Search= to find Something
3. Techniques= Methods

So all together they mean, finding a group of searching strategies that can help me find an element in an unexplored environment.

this unexplored word is really important suppose we had a sorted array then our classical search techniques could have helped us such as binary search, moreover think like you have to search some element in a tree like structure then how would you do it ?

also note that you have no idea what you will see in the next node unlike in binary search where if you hit the mid point of the array, you know that elements above have higher value than the element in the middle and all the preceeding elements have lower value.

So in such cases Uninformed Search Techniques can actually save the day.

but the key point is it has to me unexplored, meaning we don't know anything about the data.

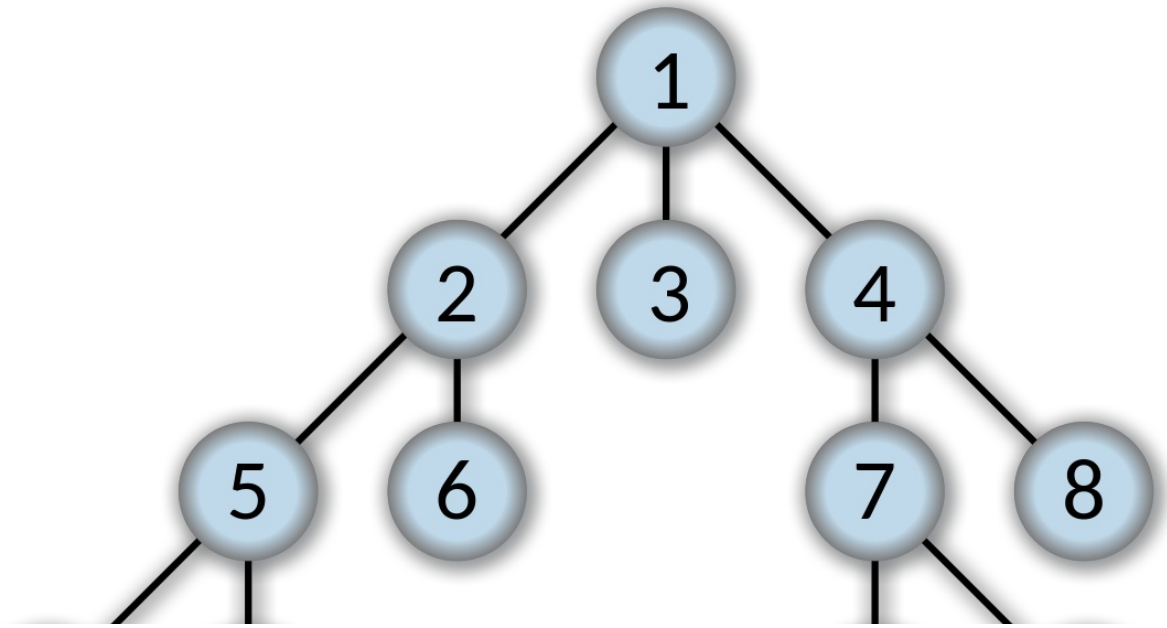
how many search techniques are there ?

There are many types of uninformed search alogorithms, but these are a few popular ones. That are used.

- Breadth-first Search
- Depth-first Search
- Depth-limited Search
- Iterative deepening depth-first search
- Uniform cost search
- Bidirectional Search

but we will only cover the first 3.

Before we begin with anything I want you to visualize the tree structure given below -



The node having "1" is called the root node because this is at the start of the tree and this is where I will begin exploring my tree.

and lets Suppose i am trying to find "10" in the tree , but the catch is that I don't know where it is.

so How should I explore the tree ?

one approach could be I go and check the root node then all the elements in the second layer i.e right after root node, in the example above, it will be **(2,3,4)**

now we will look through 2 then 3 and then 4 , unfortunately none of them is the node that i am looking forward to i.e. "10"

so what do we do now ?

pretty simple we traverse the nodes in the next layer i.e. (5,6,7,8)

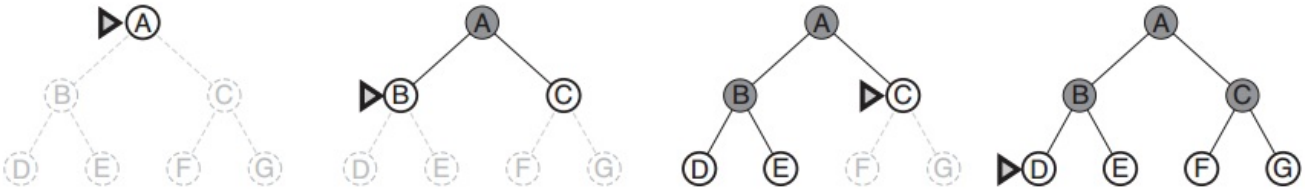
but the the problem is nodes are now divided, meaning we can either take 5,6 from node 2 or we can take 7,8 from node 4.

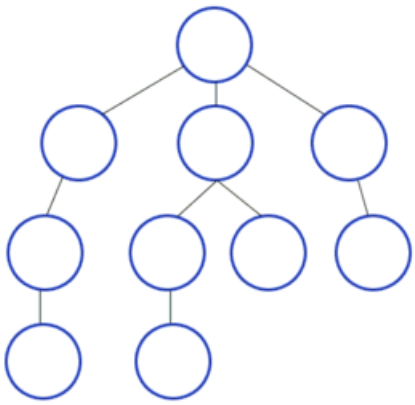
which path to follow ?

will take the path that has the first diversion meaning if we are moving from left to right I will move / traverse in in the paths available in 2 i.e 5,6 and if there is nothing that I can find, I will jump up and go to the next path i.e. 4 and look for the nodes directly below it i.e. 7,8.

But the Key fact is **I will not traverse the next layer unless all the nodes are explored in the same layer** The above assumption is what defines our first Search Technique that is called BFS, meaning Breadth First Search.

The below diagram should show how we traverse in the BFS search technique. Please try to corelate that with the above Example that I explained.





Implementation of BFS on search :

The data structure we will use to implement BFS is a queue and follow the FIFO (First in First out) approach. the

1. first element will be the root node i visited
2. if no match then move to the next location and add 1 to depth variable now d+=1
3. then traverse the next layer and look for the elements over there (while traversing t+=1 for every node without match)
4. since we now don't need the root node as reference we will delete the root node from queue (meaning FIFO)
5. my reference changes immidiately to the next node right after 1. i.e 2 so i will go down below every element in the next layer of two until can't find anything.
6. the deletion and new node of referencing will occur only when i couldn't find any node = goal in my current layer

6. the deletion and new node of referencing will occur only when I couldn't find any node = goal in my current layer.
7. at the end I will return [t , d] value if i find my goal.

the t is the node number in the layer and d is the depth number.

below is the Pseudo code for BFS: (this is the implementation that i was talking about)

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

hold on we have some other techniques, but before we discuss them let's see what BFS did wrong.

While traversing If i wanted to reach node "4" I could directly go from "2" if I tried vertical movement instead of Horizontal. so these nodes are travelled extra, moreover for any node at layer d i need to see all the nodes uptill layer "d-1" this is the drawback of the assumpltion that we made, remember travelling and finishing the nodes in the same layer ?

thus i can conclude If i want to go to any node in the tree i have to travel all the nodes before,

suppose there are N number of nodes before that node,

My time complexity will be O(N+1), why +1 ? because we are also taking time to reach the final node.

or we can say O(N), where now N means total number of nodes (including the final NODE)

suppose there are 4 nodes connected from every node and there are 5 layers of nodes.

how many nodes will I traverse in total , it will be 4^5 (meaning 4 per node^5 depth or layers)

if the no of nodes connected to every node is b (say) and total number of depth is d

then my time complexity become O(b^d)

below image shows the total no travelled in every layer is an order of b to the power d

$$b + b^2 + b^3 + \cdots + b^d = O(b^d)$$

Similarly we also need to keep the track of all the nodes that we have visited hence the sapce complexity is :

$$b + b^2 + b^3 + \cdots + b^d = O(b^d)$$

if you think this number is small consider a tree with many layers and branches, this will be time to traverse it.

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes

10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

lets try to look for a different approach, instead of going horizontal, let's traverse the node vertically first

Such an approach is called DFS, **Depth First Search**

take the same example that I explained above for BFS.

We want to find the node "10"

the assumption here is that i first travel the deepest node in the end of the tree if we can't find our goal state then we will move back one level up from the last node (which will be the deepest node), so lets try to visulaize what will happen?

First I will look at root node which is 1 so it's not my goal step we move down in the tree.

now reach the second layer we check the first element of layer 2 ,which is 2 since 2 is not the goal state aswell,

I will move now to to the next layer, first node of my layer 3 is 5 which is also not the goal state.

move another layer deep , now we are node 9 again not equal to goal state.

since 9 is at the deepest layer, We cannot move any further than this hence we move one step back from the node where we currently are, meaning from 9 i will go to it's immidiate predecessor that is 5, we can also delete 9 from the memory because it is not the goal state and no nodes lie below it.

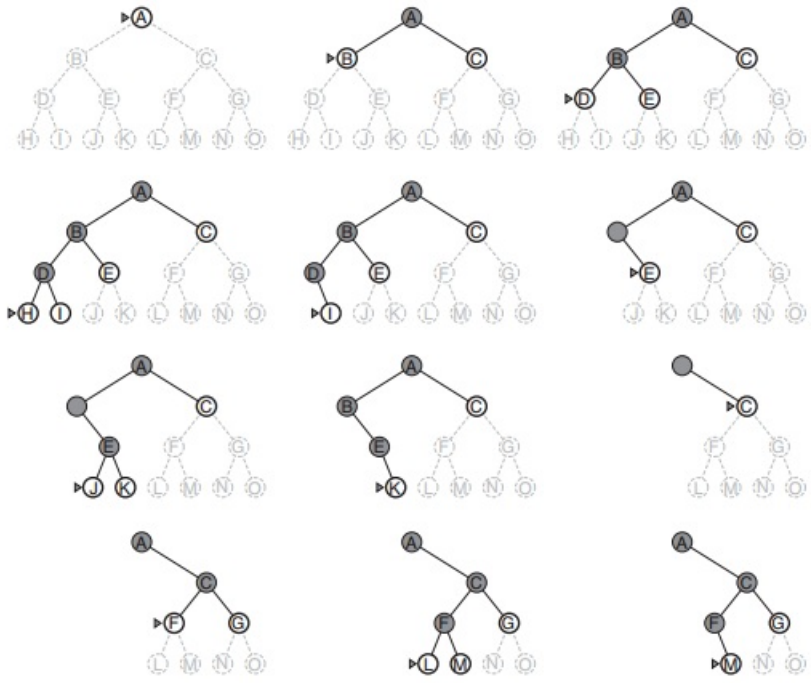
so now we are back at 5 from 5 we go to the next layer

(If we can reach/ else we will travesre back one node above 5 and then try to go deeper again)

from five we go to node 10 which is the next branch of the tree

Volla, we found our Goal state Now,

the below diagram explains the traversing of the tree



there are subtle key differences in BFS and DFS approach

1. if BFS we priorotize branches over the depth meaning we cover all the branches first and then go deeper to see all the branches in the next layer but in DFS we explore all the depth first then switch the branch.

We will only go upthe tree if all the possible nodes below my current node are explored.

1. in DFS I update my memory by deleting the elements in the deepest node and then deleting branches if the entire branch has been travelled.
2. Time depends upon the location of the block placed (but this is a relative statement i will explain below)

2. Time depends upon the location of the block placed (but this is a relative statement I will explain below)

it took us to see only 5 nodes to reach our goal in DFS while BFS took us 10 nodes to reach our destination.(but this is a relative statement i will explain below).

suppose every node has 4 branching and depth =5 then my worst time will take will be 4^5 5 being the best.

in this approach suppose i had to go to block 4 in the daigram then i would traverse through (1,2,5,9,10,6,3) in DFS If I had used BFS then it would have taken me to traverse (1,2,3) only.

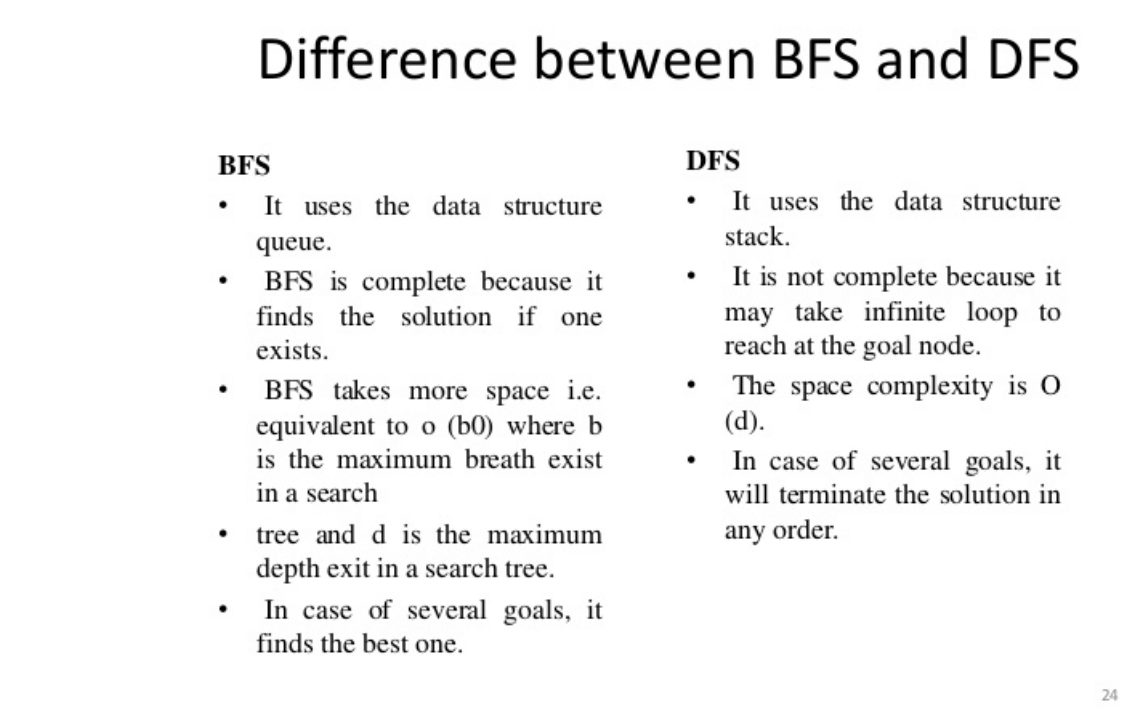
so we cannot be sure about the the time complexity of the algorithm because it depends upon the location of goal state node placed. Also the worst case time for both of the above algorithms will be same as they might have to traverse through all the nodes in the tree as a last resort.

they key difference why DFS is a more practical approach is because of the Space complexity as after reaching the deepest node I can delete my memory of Nodes that i have stored.

the total size in the memory will always be equal to the the number of nodes visited (a) * maximum depth reached so far (m)

since my total number of nodes visited so far can never be more than depth at a particular time because after we reach the maximum depth we delete the last node and move one block above.

the below picture explains the DFS and BFS difference with the context to the examples that I gave.



can we make our search technique better ?

The answer is YES and NO both, we can improve but it will cause certain other drawbacks another search technique called **limited Depth first** search limits the maximum number of nodes that we can visit If it does not find any optimal node it will return us False (meaning the element is not present).

but there is a catch with this approach- imagine if the depth limit is set to be 10 for a node of 100, then my algorithm will not explore anything below the depth of 10 and if the goal state is at depth (say 11) it will return me False that the element is not present, however there is an advantage now instead of traversing 100 nodes, I am only traversing the 10 nodes meaning time taken will be less. So it could be an advantage for some situations where the goal note is under the depth limit but can be a demerit if the depth node is above the limit set.

the pseudo code for depth milited is explained below:

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
```

```
else
    cutoff_occurred? ← false
    for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        result ← RECURSIVE-DLS(child, problem, limit − 1)
        if result = cutoff then cutoff_occurred? ← true
        else if result ≠ failure then return result
    if cutoff_occurred? then return cutoff else return failure
```

What is forward and backward chaining ? How are they useful ?

forward chaining is process to infer not known information from the basic facts, where as the backward chaining is the process where we try to find the basic facts from complex facts.

Wait a minute, I will explain this concept in much more detail.

"It is raining , The road is wet"

second statement is dependent on the first statement: As it is raining it is most likely that the road will be wet

moreover we can also infer the first statement from the second: As the road is wet most likely it would have rained.

Mathematically we can represent, them in two different variables

- It is raining (A)
- The road is wet (B)

and we know that both of them are interdependent on each other.

then,

A=>B (shows that A implies B)

B=>A (shows that B implies A)

then they both cause each other.

Forward chaining Vs Backward chaining

1. Forward chaining starts from known facts and applies inference rule to extract more data unit it reaches to the goal. Backward chaining starts from the goal and works backward through inference rules to find the required facts that support the goal.
2. FC is a bottom-up approach
3. BC is a top-down approach
4. Forward chaining is known as data-driven inference technique as we reach to the goal using the available data. Backward chaining is known as goal-driven technique as we start from the goal and divide into sub-goal to extract the facts.
5. Forward chaining reasoning applies a breadth-first search strategy. Backward chaining reasoning applies a depth-first search strategy.
6. Forward chaining tests for all the available rules Backward chaining only tests for few required rules.
7. Forward chaining is suitable for the planning, monitoring, control, and interpretation application. Backward chaining is suitable for diagnostic, prescription, and debugging application.
8. Forward chaining can generate an infinite number of possible conclusions. Backward chaining generates a finite number of possible conclusions.
9. It operates in the forward direction. It operates in the backward direction.
10. Forward chaining is aimed for any conclusion. Backward chaining is only aimed for the required data.

Source: https://www.javatpoint.com/difference-between-backward-chaining-and-forward-chaining

how to implement search techniques in my example:

• States: placement of oriley in my environment (CIT)

• Initial state: initial state of oriley is decided by the random number picked.

• Actions: oriley can change directions and make moves.

• Transition model: oriley can land on student , canvas, free, time, meeting

• Transition model: oriley can land on student , canvas, free_time, meeting.

• Goal test: oriley is alive and he has landed on all things block in environment (but in search example oriley reaches the block that holds students)

• Path cost: the distance of neighbouring block is one unit , the total distance he covers to reach student is his path cost

b3	b7	b11	b15
b2	b6	b10	b14
b1	b5	b9	13
b0	b4	b8	b12

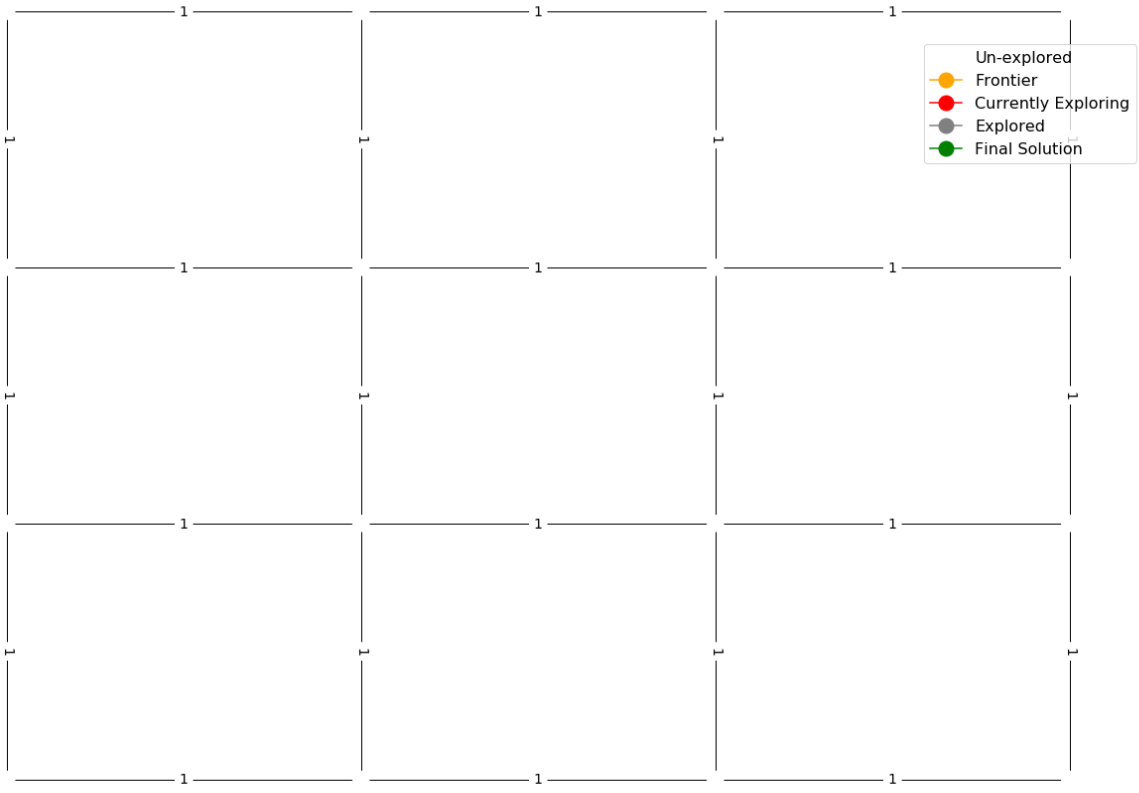
say i will put my student block at b7 and oriley at b2 now oriley will try to find shortest path using uninformed search techniques

In [51]:

```
from search import *
from notebook import psource, heatmap, gaussian_kernel, show_map, final_path_colors, display_visual
, plot_NQueens

# Needed to hide warnings in the matplotlib sections
import warnings
warnings.filterwarnings("ignore")

%matplotlib inline
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib import lines

from ipywidgets import interact
import ipywidgets as widgets
from IPython.display import display
import time

# suppose the grid is 4*4

canvas=[1,2]
free_time=[2,1]
meeting=[3,1]

romania_map = UndirectedGraph(dict(
    b0=dict(b1=1, b4=1),
    b1=dict(b0=1, b2=1, b5=1),
    b2=dict(b1=1, b6=1, b3=1),
    b3=dict(b2=1, b7=1),
    b4=dict(b0=1, b5=1, b8=1),
    b5=dict(b1=1, b4=1, b9=1, b6=1),
    b6=dict(b5=1, b7=1, b10=1, b2=1),
    b7=dict(b3=1, b6=1, b11=1),
    b8=dict(b4=1, b9=1, b12=1),
    b9=dict(b5=1, b8=1, b10=1, b13=1),
    b10=dict(b6=1, b9=1, b11=1, b14=1),
    b11=dict(b7=1, b10=1, b15=1),
    b12=dict(b8=1, b13=1),
    b13=dict(b9=1, b12=1, b14=1),
    b14=dict(b13=1, b10=1, b15=1),
    b15=dict(b11=1, b14=1)))

romania_map.locations = dict(
    b0=(0,0),b1=(0,1),b2=(0,2),b3=(0,3),
    b4=(1,0),b5=(1,1),b6=(1,2),b7=(1,3),
    b8=(2,0),b9=(2,1),b10=(2,2),b11=(2,3),
    b12=(3,0),b13=(3,1),b14=(3,2),b15=(3,3))

#b2= oriley
```

```
#b14=student

romania_problem = GraphProblem('b1', 'b2', romania_map)

romania_locations = romania_map.locations
print(romania_locations)
print(romania_map.locations.items())

# node colors, node positions and node label positions
node_colors = {node: 'white' for node in romania_map.locations.keys()}
node_positions = romania_map.locations
node_label_pos = { k:[v[0],v[1]-10]  for k,v in romania_map.locations.items() }
edge_weights = {(k, k2) : v2 for k, v in romania_map.graph_dict.items() for k2, v2 in v.items()}

romania_graph_data = {  'graph_dict' : romania_map.graph_dict,
                        'node_colors': node_colors,
                        'node_positions': node_positions,
                        'node_label_positions': node_label_pos,
                         'edge_weights': edge_weights
                     }

show_map(romania_graph_data)
```

```
{'b0': (0, 0), 'b1': (0, 1), 'b2': (0, 2), 'b3': (0, 3), 'b4': (1, 0), 'b5': (1, 1), 'b6': (1, 2),
'b7': (1, 3), 'b8': (2, 0), 'b9': (2, 1), 'b10': (2, 2), 'b11': (2, 3), 'b12': (3, 0), 'b13': (3,
1), 'b14': (3, 2), 'b15': (3, 3)}
dict_items([('b0', (0, 0)), ('b1', (0, 1)), ('b2', (0, 2)), ('b3', (0, 3)), ('b4', (1, 0)), ('b5',
(1, 1)), ('b6', (1, 2)), ('b7', (1, 3)), ('b8', (2, 0)), ('b9', (2, 1)), ('b10', (2, 2)), ('b11',
(2, 3)), ('b12', (3, 0)), ('b13', (3, 1)), ('b14', (3, 2)), ('b15', (3, 3))])
```



In [56]:

```
def tree_breadth_search_for_vis(problem):
    """Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    Don't worry about repeated paths to a state. [Figure 3.7]"""

    # we use these two variables at the time of visualisations
    iterations = 0
```

```
    all_node_colors = []
    node_colors = {k : 'white' for k in problem.graph.nodes()}

    #Adding first node to the queue
    frontier = deque([Node(problem.initial)])

    node_colors[Node(problem.initial).state] = "orange"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    while frontier:
        #Popping first node of queue
        node = frontier.popleft()

        # modify the currently searching node to red
        node_colors[node.state] = "red"
        iterations += 1
        all_node_colors.append(dict(node_colors))

        if problem.goal_test(node.state):
            # modify goal node to green after reaching the goal
            node_colors[node.state] = "green"
            iterations += 1
            all_node_colors.append(dict(node_colors))
            return(iterations, all_node_colors, node)

        frontier.extend(node.expand(problem))

        for n in node.expand(problem):
            node_colors[n.state] = "orange"
            iterations += 1
            all_node_colors.append(dict(node_colors))

        # modify the color of explored nodes to gray
        node_colors[node.state] = "gray"
        iterations += 1
        all_node_colors.append(dict(node_colors))

    return None

def breadth_first_tree_search(problem):
    "Search the shallowest nodes in the search tree first."
    iterations, all_node_colors, node = tree_breadth_search_for_vis(problem)
    return(iterations, all_node_colors, node)

all_node_colors = []
romania_problem = GraphProblem('b2', 'b14', romania_map)
a, b, c = breadth_first_tree_search(romania_problem)
display_visual(romania_graph_data, user_input=False,
               algorithm=breadth_first_tree_search,
               problem=romania_problem)
```

In [62]:

```
def tree_depth_search_for_vis(problem):
    """Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    Don't worry about repeated paths to a state. [Figure 3.7]"""

    # we use these two variables at the time of visualisations
    iterations = 0
    all_node_colors = []
    node_colors = {k : 'black' for k in problem.graph.nodes()}

    #Adding first node to the stack
    frontier = [Node(problem.initial)]

    node_colors[Node(problem.initial).state] = "orange"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    while frontier:
        #Popping first node of stack
        node = frontier.pop()
```

```
        # modify the currently searching node to red
        node_colors[node.state] = "red"
        iterations += 1
        all_node_colors.append(dict(node_colors))

        if problem.goal_test(node.state):
            # modify goal node to green after reaching the goal
            node_colors[node.state] = "green"
            iterations += 1
            all_node_colors.append(dict(node_colors))
            return(iterations, all_node_colors, node)

        frontier.extend(node.expand(problem))

        for n in node.expand(problem):
            node_colors[n.state] = "orange"
            iterations += 1
            all_node_colors.append(dict(node_colors))

        # modify the color of explored nodes to gray
        node_colors[node.state] = "gray"
        iterations += 1
        all_node_colors.append(dict(node_colors))

    return None

def depth_first_tree_search(problem):
    "Search the deepest nodes in the search tree first."
    iterations, all_node_colors, node = tree_depth_search_for_vis(problem)
    return(iterations, all_node_colors, node)

all_node_colors = []
romania_problem = GraphProblem('b2', 'b14', romania_map)
display_visual(romania_graph_data, user_input=False,
               algorithm=depth_first_tree_search,
               problem=romania_problem)
```

In [63]:

```
#depth limited search
def depth_limited_search_graph(problem, limit = -1):
    '''
    Perform depth first search of graph g.
    if limit >= 0, that is the maximum depth of the search.
    '''
    # we use these two variables at the time of visualisations
    iterations = 0
    all_node_colors = []
    node_colors = {k : 'white' for k in problem.graph.nodes()}

    frontier = [Node(problem.initial)]
    explored = set()

    cutoff_occurred = False
    node_colors[Node(problem.initial).state] = "orange"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    while frontier:
        # Popping first node of queue
        node = frontier.pop()

        # modify the currently searching node to red
        node_colors[node.state] = "red"
        iterations += 1
        all_node_colors.append(dict(node_colors))

        if problem.goal_test(node.state):
            # modify goal node to green after reaching the goal
            node_colors[node.state] = "green"
            iterations += 1
            all_node_colors.append(dict(node_colors))
            return(iterations, all_node_colors, node)
```

```
        elif limit >= 0:
            cutoff_occurred = True
            limit += 1
            all_node_color.pop()
            iterations -= 1
            node_colors[node.state] = "gray"


        explored.add(node.state)
        frontier.extend(child for child in node.expand(problem)
                        if child.state not in explored and
                        child not in frontier)

        for n in frontier:
            limit -= 1
            # modify the color of frontier nodes to orange
            node_colors[n.state] = "orange"
            iterations += 1
            all_node_colors.append(dict(node_colors))

        # modify the color of explored nodes to gray
        node_colors[node.state] = "gray"
        iterations += 1
        all_node_colors.append(dict(node_colors))

    return 'cutoff' if cutoff_occurred else None


def depth_limited_search_for_vis(problem):
    """Search the deepest nodes in the search tree first."""
    iterations, all_node_colors, node = depth_limited_search_graph(problem)
    return(iterations, all_node_colors, node)

all_node_colors = []
romania_problem = GraphProblem('b2', 'b14', romania_map)
display_visual(romania_graph_data, user_input=False,
               algorithm=depth_limited_search_for_vis,
               problem=romania_problem)
```

Explaining forward chaining:

Given problem statement:

1. It's Plagarism for CitStudents to share assignments with other cit Students
2. Shivam is a student of CIT
3. Shivam has submitted his solution as assignment
4. Shivam got solutions from Sudeep
5. Solutions make Assignment
6. Sudeep is a student at CIT

Prove: Sudeep, did plagarism ?

lets formulate it into fist order logic

X,Y- person, S- Soultions

Plagarism(X,Y): Cit_student(X) ^ Cit_student(Y) ^ Make_assignment(S) ^ share(X,Y,S)

Cit_student(X)

submit(X,S)

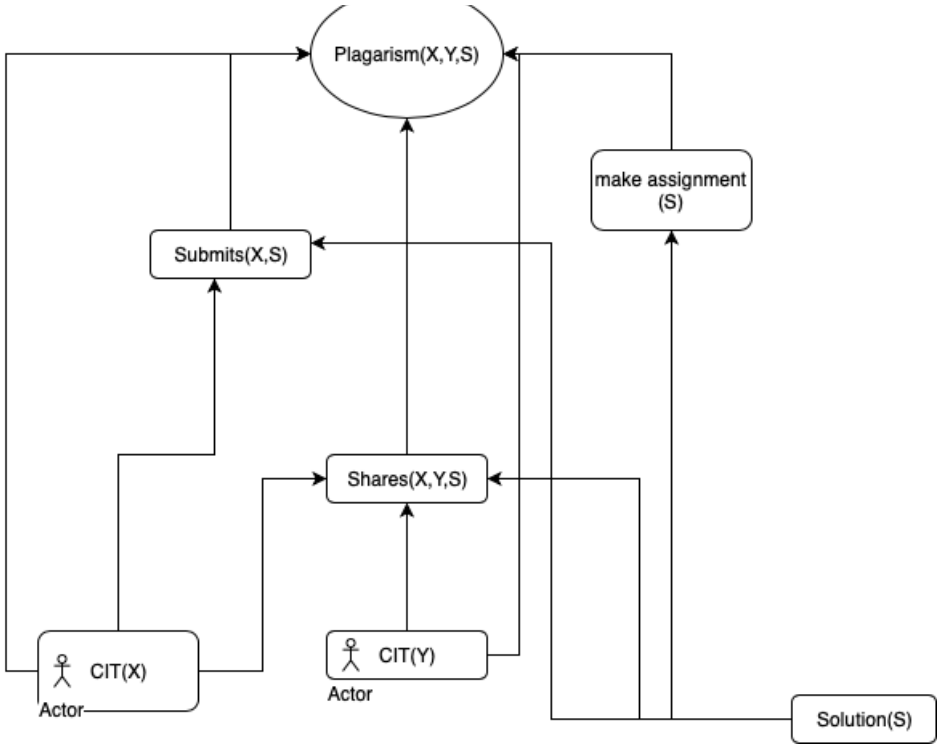
Shares(X,Y,S)

make assignment(S)

Cit_student(Y)

find: Plagarism(X,Y,S)

can be better explained with the graph:



According to forward chaining I can make many new rules and justify the statements such as:

1. If a person not form cit shares the a solution that makes the assignment and submits it, it is not counted as plagarism. inference- Cit students can take help from outside
2. if a student doesn't submit the solution that makes the assignmen that he shared with another cit student it is not plagarism.
3. A solution submitted by citstudents that does not make the problem is also considered as not plagarism.
4. if the students are not from cit and they shared submitted the solution that makes the assignment. it's not plagarism.
5. an assignment that is submitted by two cit students that makes the assignment but was not shared is not plagarism.

these are new new information derived from the basic information.

Thus this chaining process can help our bot better understand about the scenarios that he is not been aware of.

many times this flow chart can save us explore unidentified conditions that are not pre written in the agent's memory and he has to make an action.

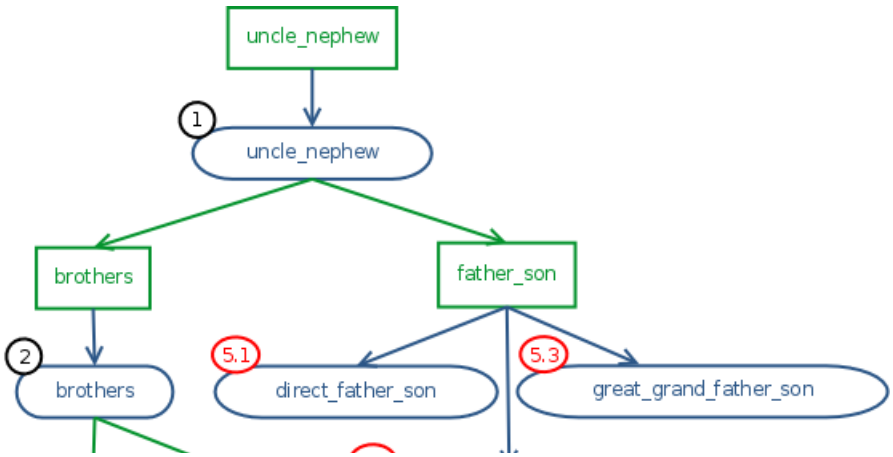
Well with chaining mechanism he can make actions based on the inputs,

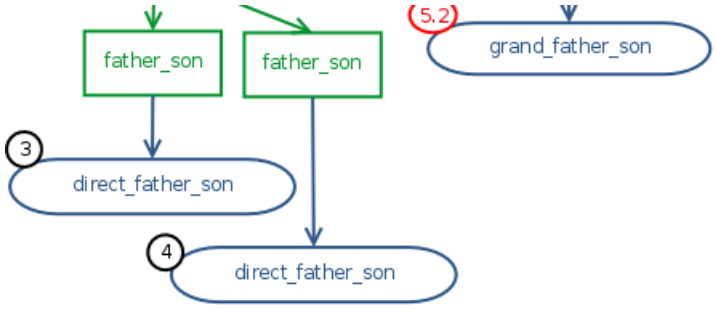
for example, Plagarism (X,Y,S) is only when there is all five conditions TRUE.

1. Student A is CIT Student
2. Student B is CIT Student
3. Solution S solves assignment
4. there was a sharing of Solution S between A and B
5. the solution S was submitted by student A

what is Backward chaining ?

the below diagram shows that we are trying to infer basic facts for complex facts





when we try to infer basic facts from a complex statement. Imagine the same graph structure but instead arrows pointing downwards.

what kind of questions it answers ?

If there is plagarism By X and Y two students.

and plagarism is defined as when a same shared solution that makes the assignment is submitted by either of the student from cit.

1. can we now say one of the student was from CIT ?
2. can we say one Cit student shared the assignment ?
3. did the solution submitted made the assignment ?
4. was the shared solution submitted
5. is S a solution ?

how to implement chaining in our example :

1. Some Blocks are things
2. Some things are students
3. Oriley gives smiles to students if he is on student block and no other block
4. landing on student block makes oriley smile
5. oriley is smiling
6. oriley is landing on things
7. oriley has to land on a block

Using forward chaining to prove that the current location of oriley is a student block ?

the given diagram explains the working of chaining in our game example of our agent

Chaining can help my agent and environment make rational decisions, by figuring out new facts based on the previosu facts.

In the above example it is trying to identify if oriley's position is on the student block, by asking 3 questions :

1. am I landing on a thing ?
2. am i oriley ?
3. am i smiling ?

(basically my bot asks three questions)

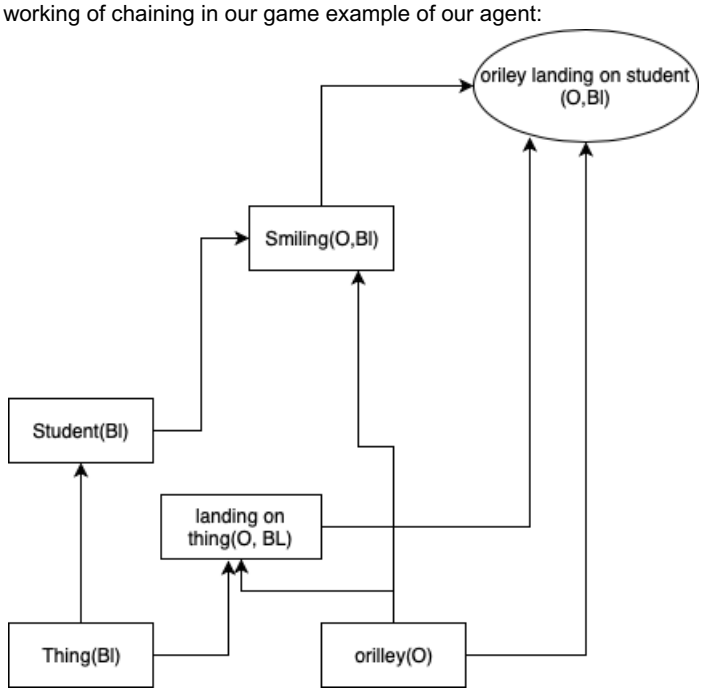
If any of the above question's answer is No then most likely I am not landing on stiudent block

this action can make my agent improve or decrease his performance in my gaem if he is landing on student bloc and is smiling he will get +100 points

on the converse the same diagram with flipped arrows can be used to answer the questions such as is the person smiling landing on a block oriley ?

or oriley is smiling , is he landing on a student block ?

the given diagram explains the working of chaining in our game example of our agent:



In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: