CS 331: INTRODUCTION TO ARTIFICIAL INTELLIGENCE

ASSIGNMENT 1:

Total Marks: 100

Instructor: Dr. Mian Muhammad Awais

TAs: Habiba Farrukh, Aiza Anjum Chaudhry, Fatima Tariq, Hamza Mahmood and Maryam Khalid

Submission Deadline: September 16th, 2015 11:59 PM

Honor Code

You are to do this assignment by yourself. All of the submitted code should be your creation and a result of your own thought process.

No cooperation is allowed under any circumstances. Any help outside of the course staff is prohibited. You are also given the responsibility of reporting any cooperation incidences to the course staff.

All code will be checked for similarities, and cases will be promptly forwarded to the Disciplinary Committee for action.

Instructions and Submission Details

- You will need to download and setup Python
- There are 6 tasks to complete. Implement each task in a separate .py file
- Make sure that you have properly commented your code
- You should submit a zip folder with all your files in it. Name the folder "CS331-Assignment1-Your_roll_number". Example: CS331-Assignment1-17100302.zip
- Submit this folder on LMS BEFORE the deadline. **No late submissions will be accepted.**
- In case your lms isn't working, email your code to the TA's. Only timely submissions will be considered.
- **Send only the code files specified in each part**.

This assignment requires some novice use of the Python programming language. If you are unfamiliar, it is highly recommended that you learn the basics first. You can find some material on the following links:
learnpython.org ,
tutorialspoint.com/python/ ,
And ofcourse your friendly neighbourhood google.com

Please feel free to consult with the TAs if you need help with setting up a python environment.

PROBLEM 1: PALINDROME [10]

We start off this assignment with a basic programming problem. You are given a string and you have to check whether that string is a palindrome or not. A palindrome is a string that remains the same if read from left to right and vice versa such as “aabbaa” or “Tacocat”.

The file for this part should be named: **palindrome.py**

PROBLEM 2: QUEUE-THE-STACK [15]

This is a tricky problem for some of you. You have to implement a queue using stack. You will use ‘QueueTheStack.py’ and ‘util.py’ to complete this task.
In this problem, use the stack class to implement a queue data structure. Hint: You can use two stacks to make a queue.
Some important things to notice in the code are:

```
from util import Stack
```

This line imports the Stack class from ‘util.py’ file which you have downloaded, since you will be using stacks to implement a queue. The program will give an error if you have not downloaded the file into the folder containing your code files. There is a small test case in main function. The file for this part should be named **QueuetheStack.py**

NOTE: Please, refer to the Appendix to understand the concepts of stack, queue and class.

PROBLEM 3: STACK-THE-QUEUE [15]

This problem is similar to the previous one. You have to implement a stack using queue. You will use ‘StackTheQueue.py’ and ‘util.py’ for this task.

Edit ‘StackTheQueue.py’ to implement ‘StackTheQueue’ class so that it acts as a stack. Hint: You can use two Queues to implement a stack. The file for this part should be named **StacktheQueue.py**

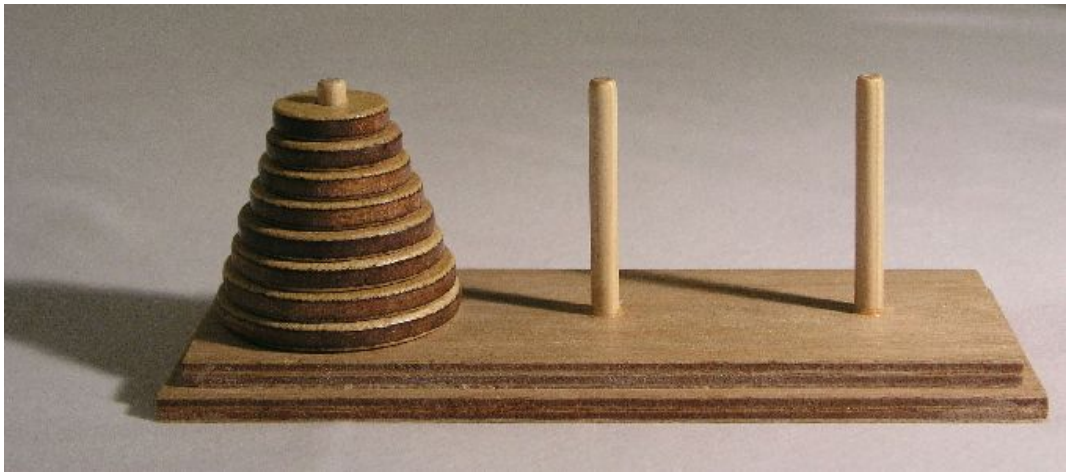
PROBLEM 4: OOP [10]

Define a class Vehicle and its subclasses Car and Bike.
The classes Car and Bike have an init function which takes the name of the vehicle (“Car” or “Bika”) and the number of wheels as arguments.
The Vehicle class must have a ‘getName’ function. Override this function in both subclasses in order to print the name of the respective vehicles.
The subclasses should also have a method named ‘describeMe’ which should print the name of the vehicle as well as its number of wheels.

Implement this task in a file named **vehicles.py**

PROBLEM 5: TOWER OF HANOI [20]

There is a story about an Indian temple in Kashi Vishawanath which contains a large room with three time-worn posts in it surrounded by 64 golden disks. Brahmin priests used to move these disks, in accordance with the immutable rules of the Brahma, since that time hence is considered a puzzle. According to the legend, when the last move of the puzzle will be completed, the world will end. Cool right?



Here you will get to practice implementing the Tower of Hanoi.

The rules of the game are very simple, but the solution is not so obvious. The game "Towers of Hanoi" uses three rods: the “source”, the “helper” and the “target”. A number of disks is stacked in decreasing order from the bottom to the top of one rod, i.e. the largest disk at the bottom and the smallest one on top. The disks build a conical tower.

The aim of the game is to move the tower of disks from one rod to another rod.

The following rules have to be obeyed:

- Only one disk may be moved at a time.
- Only the most upper disk from one of the rods can be moved in a move
- It can be put on another rod, if this rod is empty or if the most upper disk of this rod is larger than the one which is moved.

Problem: You have to define the function given below

def hanoi (n, source, helper, target)

Where n is the number of disks, source, helper and target being the three poles storing the disks. Along with it you have to run the program such that it follows the above mentioned rules and prints the source,helper and target containers at every step till the puzzle is solved.

Hint: Recall the Stack data structure. You may use an iterative or recursive solution.

Implement this task in a file named **hanoi.py**

PROBLEM 6: GRID NAVIGATION [30]

The purpose of this exercise is to acquaint you with the python game engine we will be using in the upcoming assignments.

One of the main uses of artificial intelligence in games is to perform *path planning*, the search for a sequence of movements through the virtual environment that gets an agent from one location to another without running into any obstacles. For now we will assume static obstacles. In order for an agent to engage in path planning, there must be a topography for the agent to traverse that is represented in a form that can be efficiently reasoned about. The simplest topography is a grid. Think of an imaginary lattice of cells superimposed over an environment such that an agent can be in one cell at a time. Moving in a grid is relatively

straightforward: from any cell, an agent can traverse to any of its four (or eight) neighboring cells.

You must superimpose a grid over an arbitrary, given game world terrain consisting of obstacles, so that an agent can navigate by moving left, right, up, or down from cell to cell. The code to generate the grid should work on any terrain such that an agent can never collide with an obstacle. The grid is a 2D array of booleans such that a False in any particular cell means the Agent cannot walk into the cell and a True in any particular cell means the Agent can walk into the cell. We have provided two different types of Navigator: RandomGridNavigator and GreedyGridNavigator. Neither are guaranteed to deliver an Agent to their final, desired destination.

When you click on the screen, you indicate where you want the Agent to traverse.

Step 1: Download and install the game engine. This must be done in two parts. First, you must install PyGame, the underlying rendering engine. See the installation instructions.

Step 2: Verify the installation is successful by running the game engine:
> python runbasic.py

You should see a top-down view of the game world, consisting of some obstacles (black polygons), some resources (crystal sprites), and an agent. This agent is the player avatar, meaning it is controlled by the player. When you click on a point on the screen, the agent will proceed directly there. runbasic.py uses the base Navigator class, which does not avoid collisions.

Step 3: Run the grid navigator code:
> python rungreedynavigator1.py

You will see the same thing as before with runbasic.py. However, when you click on the screen, the agent will not move. This agent uses RandomGridNavigator, which must first be completed by you.

Step 4: Modify mycreategrid.py and complete the myCreateGrid() function. myCreateGrid() must do two things:

1. It must create and return a 2D array of booleans such that grid[column][row] indicates the traversability of the cell at (column, row).

2. It must returns the dimensions of the 2D array in the form of (num_columns, num_rows).

The inputs to myCreateGrid are: a reference to the GameWorld and the cell size, which indicates the width and height of each cell. Grid cells can touch the borders of the game world. That is, a cell can have its upper left corner at point (0, 0).

Step 5: Test your implementation:
> python rungreedygridnavigator1.py
> python rungreedygridnavigator2.py
> python rungreedygridnavigator3.py

Additional testing can be done by making your own maps.
Submit the modified 'mycreategrid.py' for this task.

What you need to know

The game engine is object-oriented. The primary object is the **GameWorld**, which is a container for all other obstacles. Most importantly, the GameWorld object contains the terrain of the virtual environment. The terrain is represented as a list of **Obstacle** objects, which themselves are polygons---lists of points such that there is a line between every adjacent point (and the first and last points in the list. The GameWorld also manages the agents, bullets, resources (things that agents can gather) and computes collisions between all objects and obstacles. The GameWorld also does important stuff like run the real-time game loop and maintain the rendering windows, but you shouldn't need to worry about that. What you do need to know is that every iteration of the game loop, called a *tick*, the update method is called on all dynamic objects.

Below are the important bits of information about objects that you will be working with or need to know about for this assignment.

GameWorld

GameWorld is defined in core.py
Member functions:

- getPoints(): returns a list of all the corners of all obstacles (and the edges of the screen). A point is a tuple of the form (x, y).
- getLines(): returns a list of all the lines of all obstacles (and the screen boundaries). A line is a tuple of the form (point1, point2) where points are tuples of the form (x, y).
- getLinesWithoutBorders(): returns a list of all the lines of the obstacles, but does not include screen boundaries.
- getObstacles(): returns a list of obstacles, which are of type Obstacle.

Obstacle

Obstacle is defined in core.py. An Obstacle is a polygon through which Agents cannot move.

Member functions:

- getPoints(): returns a list of all corners in the polygon. A point is a tuple of the form (x, y).
- getLines(): returns a list of all lines in the polygon. A line is a tuple of the form (point1, point2) where points are tuples of the form (x, y).
- pointInside(point): returns true if a point (x, y) is inside the obstacle.

Agent

Agent is defined in core.py. Agent is the class type of the player avatar or non-player characters. Aside from drawing itself, an Agent knows how to move (which it inherits from its super-class Mover) and shoot. If it is moving to a particular destination, it updates its location every tick. Agents maintain a timer to control how often it can shoot.

While the Agent class does know how to move in a straight line toward a given point, it does not know how to move around an environment *without coliding with obstacles*. When instructed to move, it will move in a straight line from its current position to a target position. The intelligence in how to avoid obstacles is contained in a sub-component of the Agent, called the **Navigator**.

Member variables:

- moveTarget: the (x, y) point to which the agent has been instructed to move to. Used for interpolating the Agent's current position at any given tick.
- navigator: an object that tells the agent how to move.

Member functions:

- moveToTarget(point): Instructs the Agent to move straight to the point (x, y), ignoring the existence of obstacles.
- navigateTo(point): Instructs the Agent to create a path through the environment that avoids collisions. This function invokes the navigator's computePath() functionality.
- isMoving(): returns true if the agent is currently moving.
- getMoveTarget(): returns the point that the agent is currently moving toward.
- stopMoving(): stops the Agent from moving

Navigator

Navigator is defined in core.py. A Navigator contains the smarts for how to get around in the game world without running into obstacles. Think of it as a brain that gets attached to an agent that controls its movement. Its primary function is to compute a path between two points that steers the Agent clear of any obstacles. A path is a set of intermediate way-points that the agent should navigate to in pursuit of arriving safely at its ultimate destination. Path planning can be done in many different ways and different AI techniques will sub-class from Navigator. Once a path is computed, it sends call-back messages to the Agent to move from intermediate way-point to intermediate way-point.

Member variables:

- agent: pointer back to the Agent object that is being guided by the AI.
- world: pointer to the GameWorld object
- source: the point (x, y) from which navigation started.
- destination: the point (x, y) to which the Agent must traverse.
- path: a list of points to traverse in order that is guaranteed not to result in a collision with an obstacle.

Member functions:

- computePath(source, destination): Find a path through the terrain (causing path to be not None) and call back to the Agent to start moving. This default functionality just instructs the agent to move straight to the destination. This function will be overridden by sub-classes implementing particular path planning techniques.
- doneMoving(): the Navigator invokes this function when the agent has reached its moveTarget. doneMoving contains logic to determine what to do next. If there is a path, it will select the next point in the path as the next moveTarget and call back to the Agent.
- checkpoint(): called when the Agent reaches a point on the path.

- smooth(): optimizes the path to take shortcuts whenever possible and thereby create smoother, more efficient motion.

GridNavigator

GridNavigator is defined in gridnavigator.py. GridNavigator specializes the Navigator "brain" to work on grid topologies.

Member variables:

- grid: A two-dimensional array of booleans such that grid[row][column] indicates whether the cell can be entered by an agent.
- dimensions: A tuple (columns, rows) indicating the number of columns and rows in the grid.
- cellSize: A scalar value indicating the height and width of a cell. It is based on the size of the agent.

Member functions:

- createGrid(world): Function creates the grid by looking at the obstacles in the world and marking cells in the grid as true if traversable or false if non-traversable. This function sets the grid member variable and the dimensions member variable. To do this, createGrid() calls myCreateGrid(world, agent), a non-member function that returns two values: the 2D array of the grid and the dimensions in the form (columns, rows).

GreedyGridNavigator

GreedyGridNavigator is defined in gridnavigator.py. The GreedyGridNavigator does two things. First, it creates a gird-based path network, as described above. Second, it causes the Agent to navigate to a destination by trying to find the neighbor (left, right, up, down) that is closest to the destination. Greedy navigation may fail, so the path terminates after 100 cells at which point the Agent moves directly to its destination from the last point reached. Thus, the Agent can possibly collide with obstacles if the greedy path does not reach the destination before the threshold is reached.

Member functions:

- computePath(source, destination): Find a path through the grid (causing path to be not None) and call back to the Agent to start moving. The path is created by finding the closest cell to the source and then selecting successor cell that move the agent closer to the destination until the closest cell to the destination is found. If the path length

exceeds 100, then the Agent will be sent to its destination without further collision avoidance.

Miscellaneous utility functions

Miscellaneous utility functions are found in utils.py.

- distance(point1, point2): returns the distance between two points. Points are tuples of the form (x, y).
- calculateIntersectPoint(point1, point2, point3, point4): returns a point (x, y) at the intersection of two lines, or None if the lines are parallel. One line is between point1 and point2 and the other line between point3 and point4.
- rayTrace(point1, point2, line): returns the intersection point (x, y) if a beam between point1 and point2 crosses the given line.
- rayTraceWorld(point1, point2, worldlines): performs a ray trace against every line in worldlines and returns the first intersection point found. worldlines is a list of lines of the form ((x1, y1), (x2, y2)).
- rayTraceNoEndpoints(point1, point2, line): same as rayTrace(), but doesn't check collisions with the end points of the two lines.
- rayTraceWorldNoEndpoints(point1, point2, worldlines): same as rayTraceWorld(), but doesn't check end points of any lines compared against each other.
- pointInsidePolygonPoints(point, listofpoints): returns true if point is within a polygon defined by listofpoints. Points are tuples of form (x, y).
- pointInsidePolygonLines(point, lines): returns true if point is within a polygon defined by lines. The point is of the form (x, y). Lines is a list of tuples of the form ((x1, y1), (x2, y2)).
- drawCross(surface, point, color, size, width): draw a cross on a PyGame drawing surface. Point is the center of the cross, a tuple of the form (x, y). Color is a tuple of the form (red, green, blue) with values between 0 and 255, each. size is the length of the lines in the cross. width is the width of the lines.

Hints

Debugging within the game engine can be hard. Print statements will be one possible way of figuring out what is going on.

It can also be helpful to draw debugging information, such as lines and points, to the screen. To draw a cross on the screen use the following:

drawCross(self.world.background, point) --- draws a cross on the screen, but will be erased at the next tick.

drawCross(self.world.debug, point) --- draws a cross on the screen, which will remain visible from that point on.

In general, you should be able to get all the data you need from the GameWorld, Obstacles objects, and the Agent through getter functions. If you find yourself directly accessing member variables of other objects, you may want to rethink your approach.

It is good to test your techniques on new maps. If you want to make new maps, make a copy of one of the run*.py files and edit the call to world.initializeTerrain(). initializeTerrain(polygons, color, linewidth, sprite) instantiates the physical Obstacle objects (specifically ManualObstacle objects) in a game. A list of polygons is given, such that there is one polygon for each obstacle. A polygon is a list of points where a point is a tuple of the form (x, y). The color of the line is given, in the form (red, green, blue) where each values is between 0 and 255. The default color is black (0, 0, 0). Linewidth indicates how thick to make the line of the polygon. Obstacles will be filled with repeating sprites if the optional sprite parameter is given as a string pathname to an image file.

Sometimes the game world is larger than the screen resolution of a computer. The size of the screen is independent of the size of the game world. GameWorld can be initialized with three parameters. The first is a seed to help reproduce random values. If seed is None, the current system time is used as the seed. The second is the dimensions of the world in the form (x, y). The third is the dimensions of the screen in the form (x, y). The dimensions of the screen should be equal to or smaller than the dimensions of the world.

Visit this link for the documentation:
http://game-ai.gatech.edu/sites/default/files/documents/documentation/gaige-object-documentation.pdf

APPENDIX

Please read the whole class section before coming to office hours for clarification.

CLASS:

Class is a simple phenomenon in programming languages to depict concepts of real world to programming world. Let's try to understand the concept of classes with the help of an example.

We want to represent the concept of a room in our program. We will build a structure (class) in a program to depict this concept. Now, a room contains various things like, table, chair, bed, window, doors, etc. Our class Room should contain all or some of these items. The class definition in Python will go like this:

Note: Do not copy paste the code as it will cause problems for Python interpreter.

```
class Room:
    def __init__(self,doors,windows):
        self.doors = doors
        self.windows = windows
        self.table = 0
        self.chair = 0
        self.bed = 0
        self.maxBeds = 2
    def addTable(self,table):
        self.table += table
    def addChair(self,chair):
        self.chair += chair
    def addBed(self,bed):
        if (self.bed + bed > self.maxBeds):
            return False
        else:
            self.bed += bed
            return True
    def getTable(self):
        return self.table
```

```
    def getChair(self):
        return self.chair
    def getWindows(self):
        return self.windows
    def getDoors(self):
        return self.doors
    def getBeds(self):
        return self.bed
#Main Method starts here
if __name__ == '__main__':
    noOfDoors = 2
    noOfWindows = 1
    #Instantiate an Object of Room
    myRoom = Room(noOfDoors, noOfWindows)
    myRoom.addTable(1)
    myRoom.addChair(3)
    print 'Doors: ', myRoom.getDoors()
    print 'Windows: ',myRoom.getWindows()
    print 'Tables: ',myRoom.getTable()
    print 'Chairs: ',myRoom.getChair()
    for i in range(0,3):
        if myRoom.addBed(1):
            print 'bed added'
        else:
            print 'no bed added'
```

There are some points to highlight in this code:

1- `def __init__(self,doors,windows):` defines the constructor of a class. What is a constructor? This is a function used to define/construct an object of a class, for instance, in main method `myRoom = Room(noOfDoors, noOfWindows)` instantiates a new room variable (here named `myRoom`). The constructor is called by using the class name (in this case `Room`). Using this variable you can access the variables and use other functions present in the class. Like, `myRoom.addBed(1)` adds a new bed into the room.

2- self is a variable for the class to refer to itself. This is used in all the functions' input arguments. However, you will note that when calling the functions, the self variable is not passed. This is because Python interpreter passes it on its own.

3- A class can contain many variables and functions. Like, Room may contain some beds, windows, etc. There might be some variables which must be present, for instance, a room must contain a door and a window. Also, a room may contain no more than 2 beds and so on. This will be ensured by using functions.

4- There are adder functions like addBed() but no functions to exclude different objects. This can be done by adding more functions.

5- The for loop in main function tries to add more than 2 beds in the room. The function addBed() will return false on third iteration and 'no room added' will be printed, because the maximum number of beds in a room can be 2.

Output: The output should be like this:

```
Doors: 2
Windows: 1
Tables: 1
Chairs: 3
bed added
bed added
no bed added
```

STACK:

Stack is a special type of data structure/container with a Last-in-First-Out (LIFO) queuing policy. This means that whatever is inserted/pushed over the stack at last will come out first, for instance, if there is a list of numbers [1 2 3 4 5], and they are all pushed over the stack (in order of their indexes), then popping them out will give [5 4 3 2 1]. Other examples are stack of clothes, which most of us are not able handle correctly, that is, they get mixed up, when we try to pull a clothes out. See the following the example:

```
stack = Stack()□# Stack() is a class which depicts/functions as a stack
```
We have an empty container named `stack`.

```
>> stack.push(1)
>> # stack contains: 1
>> stack.push(2)
>> # stack contains : 2,1
>> stack.push(3)
>> # stack contains: 3,2,1
You can see code for Stack implementation in 'util.py'
| >> stack.pop()
| >> # stack contains: 2,1
| >> stack.pop()
| >> # stack contains: 1
| >> stack.pop()
| >> # stack is empty
```

I think now you would handle your clothes efficiently.

QUEUE:

Queue is another type of data structure/container with a First-in-First-Out (FIFO) queuing policy. This means that whatever comes first will come out first, for instance, if we enter these numbers [1 2 3 4 5] in order into the queue, then popping/dequeuing them all will give the numbers in the same order. Another example of this is queue in PDC, which some of us do not follow and get fined. See the following example:

```
>> queue = Queue()
>> # Queue() is a class which enacts/functions as a queue
>> # We have an empty Queue named 'queue'
>> queue.push(1) | >> queue.pop()
>> # queue contains: 1 | >> # queue contains: 1,2
>> queue.push(2)
>> # queue contains: 1,2
>> queue.push(3)
>> # queue contains: 1,2,3
You can see code for Queue implementation in 'util.py'
```

I think now you are able to follow the queue as you have very good concept of it.

```
| >> queue.pop()
| >> # queue contains: 1
| >> queue.pop()
| >> # queue is empty
```