MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

LAB MANUAL - B. Tech., CSE [AI&ML]

ARTIFICIAL INTELLIGENCE LABORATORY – CSE 2264

Department of Computer Science and Engineering

INSTRUCTIONS TO THE STUDENTS

Pre- Lab Session Instructions

1. Students should carry the Class notes, Lab Manual and the required stationery to every lab session
2. Be in time and follow the Instructions from Lab Instructors
3. Must Sign in the log register provided
4. Make sure to occupy the allotted seat and answer the attendance
5. Adhere to the rules and maintain the decorum

In- Lab Session Instructions

- Follow the instructions on the allotted exercises given in Lab Manual
- Show the program and results to the instructors on completion of experiments
- On receiving approval from the instructor, copy the program and results in the Lab record
- Prescribed textbooks and class notes can be kept ready for reference if required

General Instructions for the exercises in Lab

- The programs should meet the following criteria:
- o Programs should be interactive with appropriate prompt messages, error messages if any, and descriptive messages for outputs.
- o Use meaningful names for variables and procedures.
- Copying from others is strictly prohibited and would invite severe penalty during evaluation.
- The exercises for each week are divided under three sets:
- o Lab exercises – to be completed during lab hours
- o Additional questions – to be completed outside the lab or in the lab to enhance the skill
- In case a student misses a lab class, he/ she must ensure that the experiment is completed at students end or in a repetition class (if available) with the permission of the faculty concerned but credit will be given only to one day's experiment(s).
- Questions for lab tests and examination are not necessarily limited to the questions in the manual, but may involve some variations and / or combinations of the questions.

THE STUDENTS SHOULD NOT...

1. Bring mobile phones or any other electronic gadgets to the lab.
2. Go out of the lab without permission.

Course Objectives

The subject aims to provide the student with:

- an introduction to Artificial Intelligence techniques for building intelligent agents.
- an understanding of the basic issues of informed and uninformed searching techniques.
- an introduction to knowledge representation and reasoning techniques models.
- problem solving using expert systems.

Course Outcomes

Evaluation Plan

- Internal Assessment Marks : 60 Marks

- End semester assessment : 40 Marks
  - ✓ Duration: 2 hours
  - ✓ Total marks : Write up : 15 Marks
    Execution : 25 Marks

CONTENTS

Lab 1 – Python Basics-1 (Data Structures)

Tuples

Tuples are a built-in data structure in Python that are similar to lists, but with some key differences. Tuples are immutable, meaning their values cannot be changed once they are created. They are also usually used to store related values, as they allow you to group data together in a single object.

```
# Creating a tuple
my_tuple = (1, 2, 3, 4)

# Accessing elements in a tuple
print(my_tuple[0]) # Output: 1
print(my_tuple[-1]) # Output: 4

# Slicing a tuple
print(my_tuple[1:3]) # Output: (2, 3)

# Tuple concatenation
new_tuple = my_tuple + (5, 6)
print(new_tuple) # Output: (1, 2, 3, 4, 5, 6)

# Tuple repetition
print(my_tuple * 3) # Output: (1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4)
```

List

Lists are a built-in data structure in Python that are used to store an ordered collection of items. Lists are mutable, meaning their values can be changed after they are created. They can contain elements of different types, including other lists.

```
# Creating a list

my_list = [1, 2, 3, 4]


# Accessing elements in a list

print(my_list[0]) # Output: 1

print(my_list[-1]) # Output: 4


# Slicing a list

print(my_list[1:3]) # Output: [2, 3]


# List concatenation

new_list = my_list + [5, 6]

print(new_list) # Output: [1, 2, 3, 4, 5, 6]


# List repetition

print(my_list * 3) # Output: [1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
```

Data Dictionary

A data dictionary is a collection of descriptions of the variables in a dataset, including their names, types, and other characteristics. In Python, you can use a dictionary to store this information.

```
data_dictionary = {
    "variable_1": {"type": "string", "description": "Name of the person"},
    "variable_2": {"type": "integer", "description": "Age of the person"},
    "variable_3": {"type": "float", "description": "Height of the person in meters"}
}

def add_variable(variable_name, variable_type, description):
    data_dictionary[variable_name] = {"type": variable_type, "description": description}

def update_variable(variable_name, key, value):
    if variable_name in data_dictionary:
        data_dictionary[variable_name][key] = value
    else:
        print("Error: Variable not found in data dictionary.")

def delete_variable(variable_name):
    if variable_name in data_dictionary:
        del data_dictionary[variable_name]
    else:
        print("Error: Variable not found in data dictionary.")

def get_variable_info(variable_name):
    if variable_name in data_dictionary:
        return data_dictionary[variable_name]
    else:
        print("Error: Variable not found in data dictionary.")
        return None
```

Python code for stack implentation.

1. The Stack class initializes an empty list self.items to store the stack items.
2. The push method adds an item to the end of the list self.items, which represents the top of the stack.
3. The pop method removes and returns the last item from the list self.items. If the list is empty, it returns None.
4. The peek method returns the last item from the list self.items without removing it. If the list is empty, it returns None.
5. The is_empty method returns True if the list self.items is empty, and False otherwise.

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop() if self.items else None

    def peek(self):
        return self.items[-1] if self.items else None

    def is_empty(self):
        return not self.items
```

Implementation of a queue in Python using a list

```
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)

    def is_empty(self):
        return len(self.queue) == 0

    def size(self):
        return len(self.queue)
```

Lab Questions:

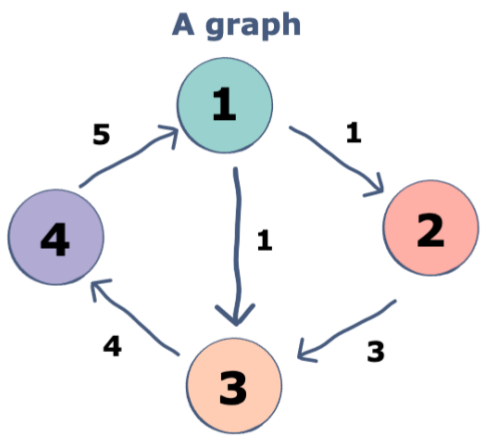
Q1. Implementation of a queue in Python using two stacks.

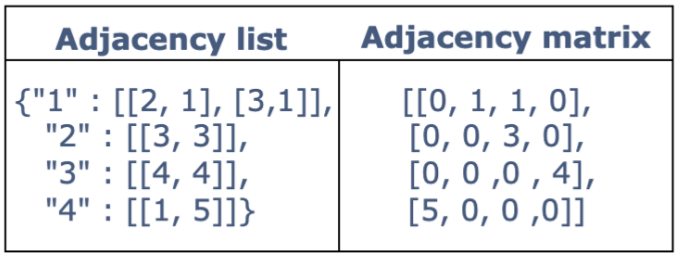
Description: A queue can be implemented using two stacks in Python by following the below steps:

1. Use two stacks, **stack1** and **stack2**, to implement the enqueue and dequeue operations.
2. In the enqueue operation, push the new element onto **stack1**.
3. In the dequeue operation, if **stack2** is empty, transfer all elements from **stack1** to **stack2**. The element at the top of **stack2** is the first element that was pushed onto **stack1** and thus represents the front of the queue. Pop this element from **stack2** to return it as the dequeued element.

Q2. Implement the following graph using python. Print the adjacency list and adjacency matrix.

[A graph is a data structure that consists of vertices that are connected via edges.]





Adjacency list	Adjacency matrix
{"1" : [[2, 1], [3,1]],	[[0, 1, 1, 0],
"2" : [[3, 3]],	[0, 0, 3, 0],
"3" : [[4, 4]],	[0, 0 ,0 , 4],
"4" : [[1, 5]]}	[5, 0, 0 ,0]]

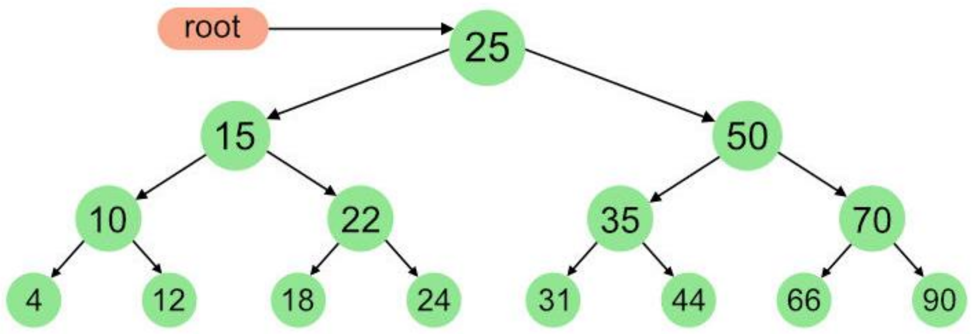
Q3. Create two list X and Y with some set of numerical values. Compute Euclidean distance for corresponding values in X and Y. Store the distance values in a separate list and sort them using Bubble sort algorithm.

Formula

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

p, q	= two points in Euclidean n-space
q_i, p_i	= Euclidean vectors, starting from the origin of the space (initial point)
n	= n-space

Q4. Implement the given binary search tree using Python and print the pre-order, in-order, and post-order tree traversal.



Expected Output:

InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

Lab 2 – Python Basics2 (Object Oriented Programming Concepts)

Class

The class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods. For example: if you have an employee class, then it should contain an attribute and method, i.e. an email id, name, age, salary, etc.

Syntax

```
class ClassName:
    <statement-1>
    .
    .
    <statement-N>
```

Object

The object is an entity that has state and behavior. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.

Everything in Python is an object, and almost everything has attributes and methods. All functions have a built-in attribute __doc__, which returns the docstring defined in the function source code.

When we define a class, it needs to create an object to allocate the memory. Consider the following example.

Example:

```
class car:
    def __init__(self,modelname, year):
        self.modelname = modelname
        self.year = year

    def display(self):

        print(self.modelname,self.year)


c1 = car("Toyota", 2016)

c1.display()
```

Instantiate an Object in Python

When we define a class only the description or a blueprint of the object is created. There is no memory allocation until we create its object. The objector instance contains real data or information.

Instantiation is nothing but creating a new object/instance of a class. Let's create the object of the above class we defined-

```
obj1 = Car()
```

And it's done! Note that you can change the object name according to your choice.

Try printing this object-

```
print(obj1)
```

Since our class was empty, it returns the address where the object is stored i.e 0x7fc5e677b6d8

You also need to understand the class constructor before moving forward.

Class constructor

Until now we have an empty class Car, time to fill up our class with the properties of the car. The job of the class constructor is to assign the values to the data members of the class when an object of the class is created.

There can be various properties of a car such as its name, color, model, brand name, engine power, weight, price, etc. We'll choose only a few for understanding purposes.

```
class Car:
    def __init__(self, name, color):
        self.name = name
        self.color = color
```

So, the properties of the car or any other object must be inside a method that we call __init__( ). This __init__() method is also known as the constructor method. We call a constructor method whenever an object of the class is constructed.

Now let's talk about the parameter of the __init__() method. So, the first parameter of this method has to be self. Then only will the rest of the parameters come.

The two statements inside the constructor method are –

self.name = name

self.color = color:

This will create new attributes namely name and color and then assign the value of the respective parameters to them. The "self" keyword represents the instance of the class. By using the "self" keyword we can access the attributes and methods of the class. It is useful in method definitions and in variable initialization. The "self" is explicitly used every time we define a method.

Note: You can create attributes outside of this __init__() method also. But those attributes will be universal to the whole class and you will have to assign the value to them.

Suppose all the cars in your showroom are Sedan and instead of specifying it again and again you can fix the value of car_type as Sedan by creating an attribute outside the __init__().

```
class Car:
    car_type = "Sedan"              #class attribute
    def __init__(self, name, color):
        self.name = name              #instance attribute
        self.color = color          #instance attribute
```

Here, Instance attributes refer to the attributes inside the constructor method i.e self.name and self.color. And, Class attributes refer to the attributes outside the constructor method i.e car_type.

Class methods

Methods are the functions that we use to describe the behavior of the objects. They are also defined inside a class.

The methods defined inside a class other than the constructor method are known as the instance methods. Furthermore, we have two instance methods here- description() and max_speed(). Let's talk about them individually-

description()- This method is returning a string with the description of the car such as the name and its mileage. This method has no additional parameter. This method is using the instance attributes.

max_speed()- This method has one additional parameter and returning a string displaying the car name and its speed.

Notice that the additional parameter speed is not using the "self" keyword. Since speed is not an instance variable, we don't use the self keyword as its prefix. Let's create an object for the class described above.

```
obj2 = Car("Honda City",24.1)
print(obj2.description())
print(obj2.max_speed(150))
```

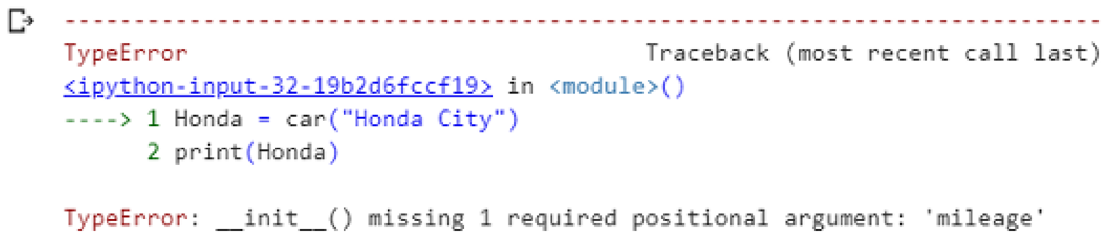
Creating more than one object of a class

```
class Car:
    def __init__(self, name, mileage):
        self.name = name
        self.mileage = mileage

    def max_speed(self, speed):
        return f"The {self.name} runs at the maximum speed of {speed}km/hr"
```

```
Honda = Car("Honda City",21.4)
print(Honda.max_speed(150))

Skoda = Car("Skoda Octavia",13)
print(Skoda.max_speed(210))
```

Passing the wrong number of arguments.

```
class Car:

    def __init__(self, name, mileage):
        self.name = name
        self.mileage = mileage

Honda = Car("Honda City")
print(Honda)
```



```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-32-19b2d6fccf19> in <module>()
----> 1 Honda = car("Honda City")
      2 print(Honda)

TypeError: __init__() missing 1 required positional argument: 'mileage'
```

Since we did not provide the second argument, we got this error.

Order of the arguments

```
class Car:

    def __init__(self, name, mileage):
        self.name = name
        self.mileage = mileage
    def description(self):
        return f"The {self.name} car gives the mileage of {self.mileage}km/l"


Honda = Car(24.1,"Honda City")
print(Honda.description())
```

Inheritance in Python Class

Inheritance is the procedure in which one class inherits the attributes and methods of another class. The class whose properties and methods are inherited is known as Parent class. And the class that inherits the properties from the parent class is the Child class.

The interesting thing is, along with the inherited properties and methods, a child class can have its own properties and methods.

How to inherit a parent class? Use the following syntax:

```
class parent_class:
body of parent class


class child_class( parent_class):
body of child class
```

Let’s see the implementation-

```
class Car:          #parent class


    def __init__(self, name, mileage):
        self.name = name
        self.mileage = mileage


    def description(self):
        return f"The {self.name} car gives the mileage of {self.mileage}km/l"
```

```
class BMW(Car):     #child class
    pass

class Audi(Car):     #child class
    def audi_desc(self):
        return "This is the description method of class Audi."
```

```
obj1 = BMW("BMW 7-series",39.53)
print(obj1.description())

obj2 = Audi("Audi A8 L",14)
print(obj2.description())
print(obj2.audi_desc())
```

We have created two child classes namely “BMW” and “Audi” who have inherited the methods and properties of the parent class “Car”. We have provided no additional features and methods in the class BMW. Whereas one additional method inside the class Audi.

Notice how the instance method description() of the parent class is accessible by the objects of child classes with the help of obj1.description() and obj2.description(). And also the separate method of class Audi is also accessible using obj2.audi_desc().

Encapsulation

Encapsulation, as I mentioned in the initial part of the article, is a way to ensure security. Basically, it hides the data from the access of outsiders. Such as if an organization wants to protect an object/information from unwanted access by clients or any unauthorized person then encapsulation is the way to ensure this.

You can declare the methods or the attributes protected by using a single underscore (_) before their names. Such as- self._name or def _method( ); Both of these lines tell that the attribute and method are protected and should not be used outside the access of the class and sub-classes but can be accessed by class methods and objects.

Though Python uses ‘ _ ‘ just as a coding convention, it tells that you should use these attributes/methods within the scope of the class. But you can still access the variables and methods which are defined as protected, as usual.

Now for actually preventing the access of attributes/methods from outside the scope of a class, you can use “private members“. In order to declare the attributes/method as private members, use double underscore (__) in the prefix. Such as – self.__name or def __method(); Both of these lines tell that the attribute and method are private and access is not possible from outside the class.

```
class car:


    def __init__(self, name, mileage):
        self._name = name                #protected variable
        self.mileage = mileage


    def description(self):
        return f"The {self._name} car gives the mileage of {self.mileage}km/l"
```

```
obj = car("BMW 7-series",39.53)


#accessing protected variable via class method
print(obj.description())


#accessing protected variable directly from outside
print(obj._name)
print(obj.mileage)
```

Notice how we accessed the protected variable without any error. It is clear that access to the variable is still public. Let us see how encapsulation works-

```
class Car:


    def __init__(self, name, mileage):
        self.__name = name              #private variable
        self.mileage = mileage


    def description(self):
        return f"The {self.__name} car gives the mileage of {self.mileage}km/l"
```

```
obj = Car("BMW 7-series",39.53)


#accessing private variable via class method
print(obj.description())


#accessing private variable directly from outside
print(obj.mileage)
print(obj.__name)
```

Polymorphism

This is a Greek word. If we break the term Polymorphism, we get “poly”-many and “morph”-forms. So Polymorphism means having many forms. In OOP it refers to the functions having the same names but carrying different functionalities.

```
class Audi:
  def description(self):
    print("This the description function of class AUDI.")
```

```
class BMW:
  def description(self):
    print("This the description function of class BMW.")
```

```
audi = Audi()
bmw = BMW()
for car in (audi,bmw):
 car.description()
```

When the function is called using the object audi then the function of class Audi is called and when it is called using the object bmw then the function of class BMW is called.

Data abstraction

We use Abstraction for hiding the internal details or implementations of a function and showing its functionalities only. This is similar to the way you know how to drive a car without knowing the background mechanism. Or you know how to turn on or off a light using a switch but you don't know what is happening behind the socket.

Any class with at least one abstract function is an abstract class. In order to create an abstraction class first, you need to import ABC class from abc module. This lets you create abstract methods inside it. ABC stands for Abstract Base Class.

```
from abc import ABC


class abs_class(ABC):
    Body of the class
```

Important thing is– you cannot create an object for the abstract class with the abstract method. For example-

```
from abc import ABC, abstractmethod
```

```
class Car(ABC):
    def __init__(self,name):
        self.name = name

 @abstractmethod
    def price(self,x):
        pass
obj = Car("Honda City")
```

```
from abc import ABC, abstractmethod

class Car(ABC):
    def __init__(self,name):
        self.name = name

    def description(self):
        print("This the description function of class car.")

    @abstractmethod
    def price(self,x):
        pass
class new(Car):
    def price(self,x):
        print(f"The {self.name}'s price is {x} lakhs.")
obj = new("Honda City")

obj.description()
obj.price(25)
```

Car is the abstract class that inherits from the ABC class from the abc module. Notice how I have an abstract method (price()) and a concrete method (description()) in the abstract class. This is because the abstract class can include both of these kinds of functions but a normal class cannot. The other class inheriting from this abstract class is new(). This method is giving definition to the abstract method (price()) which is how we use abstract functions.

After the user creates objects from new() class and invokes the price() method, the definitions for the price method inside the new() class comes into play. These definitions are hidden from the user. The Abstract method is just providing a declaration. The child classes need to provide the definition.

But when the description() method is called for the object of new() class i.e obj, the Car's description() method is invoked since it is not an abstract method.

Collections In Python :

What Are Collections In Python?

Collections in python are basically container data types, namely lists, sets, tuples, dictionary. They have different characteristics based on the declaration and the usage.

A list is declared in square brackets, it is mutable, stores duplicate values and elements can be accessed using indexes.

A tuple is ordered and immutable in nature, although duplicate entries can be there inside a tuple.

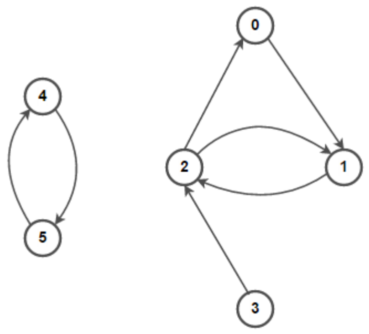
A set is unordered and declared in square brackets. It is not indexed and does not have duplicate entries as well.

A dictionary has key value pairs and is mutable in nature. We use square brackets to declare a dictionary.

These are the python’s general purpose built-in container data types. But as we all know, python always has a little something extra to offer. It comes with a python module named collections which has specialized data structures.

Lab Questions:

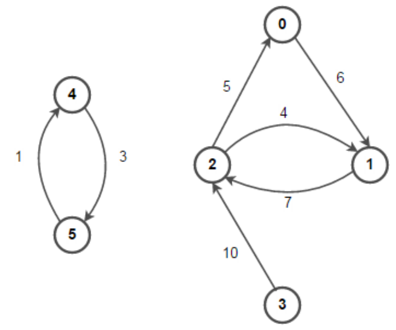
Q1. Implement the following directed unweighted graph using class, methods, and data structures of Python.



Expected output:

```
(0 —> 1)
(1 —> 2)
(2 —> 0) (2 —> 1)
(3 —> 2)
(4 —> 5)
(5 —> 4)
```

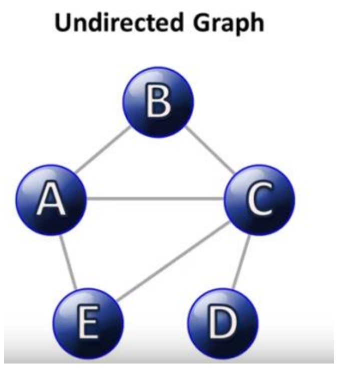
Q2. Implement the following directed weighted graph using class, methods, and data structures of Python.



Expected Output:

```
(0 —> 1, 6)
(1 —> 2, 7)
(2 —> 0, 5) (2 —> 1, 4)
(3 —> 2, 10)
(4 —> 5, 1)
(5 —> 4, 3)
```

Q3. Implement the following undirected weighted graph using class, methods, and data structures of Python. Print the adjacency list and adjacency matrix.



Expected Output:

Adjacency List:

```
["A:['B',
'C', 'E']",
"C:['A',
'B', 'D',
'E']",
"B:['A',
'C', 'D']",
"E:['A',
'C']",
"D:['B',
'C']"]
```

Adjacency Matrix

```
[[ 0.  1.  1.  0.  1.]
 [ 1.  0.  1.  1.  0.]
 [ 1.  1.  0.  1.  1.]
 [ 0.  1.  1.  0.  0.]
 [ 1.  0.  1.  0.  0.]]
```

Additional Questions:

Consider a situation where there is a single teller in a bank who can assist customers with their transactions. When a customer arrives at the bank, they join the end of a queue to wait for the teller. When the teller is available, they assist the first customer in the queue and remove them from the queue.

Description:

Customer class is defined to store information about each customer, such as their name and transaction. The Bank class is defined with a queue attribute to store instances of the Customer class, and methods to add customers to the queue (add_customer), serve the next customer in the queue (serve_customer), and check if the queue is empty (is_queue_empty). The code simulates customers arriving at the bank and being served by the teller. The teller serves customers in the order they arrive and removes them from the queue using the pop(0) method.

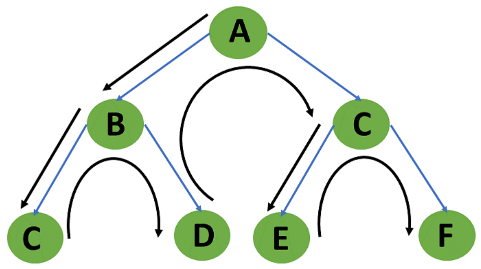
Lab 3 - Implementation of Depth First Search

Depth First Search

Depth-First Search or DFS algorithm is a recursive algorithm that uses the backtracking principle. It entails conducting exhaustive searches of all nodes by moving forward if possible and backtracking, if necessary. To visit the next node, pop the top node from the stack and push all of its nearby nodes into a stack. Topological sorting, scheduling problems, graph cycle detection, and solving puzzles with just one solution, such as a maze or a sudoku puzzle, all employ depth-first search algorithms. Other applications include network analysis, such as determining if a graph is bipartite.

What is a Depth-First Search Algorithm?

The depth-first search or DFS algorithm traverses or explores data structures, such as trees and graphs. The algorithm starts at the root node (in the case of a graph, you can use any random node as the root node) and examines each branch as far as possible before backtracking.



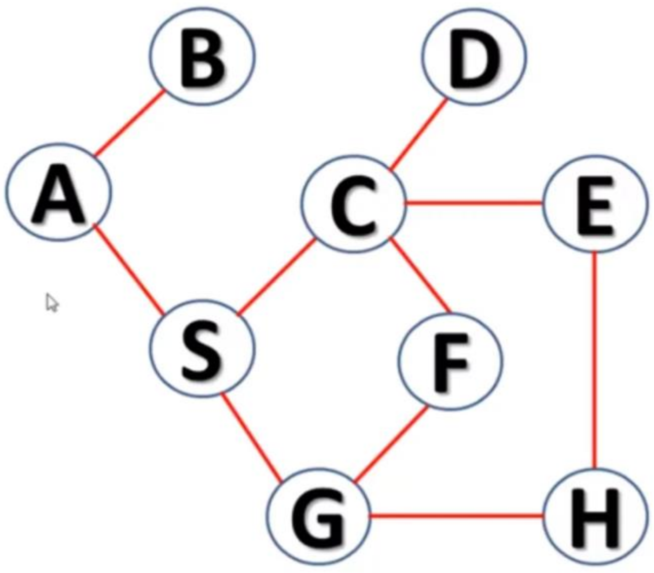
When a dead-end occurs in any iteration, the Depth First Search (DFS) method traverses a network in a deathward motion and uses a stack data structure to remember to acquire the next vertex to start a search.

Following the definition of the dfs algorithm, you will look at an example of a depth-first search method for a better understanding.

Example of Depth-First Search Algorithm

The outcome of a DFS traversal of a graph is a spanning tree. A spanning tree is a graph that is devoid of loops. To implement DFS traversal, you need to utilize a stack data structure with a maximum size equal to the total number of vertices in the graph.

To implement DFS traversal, you need to take the following stages.



Step 1: A is the root node. So since A is visited, we push this onto the stack.
Stack : A
Step 2: Let's go to the branch A-B. B is not visited, so we go to B and push B onto the stack.
Stack : A B
Step 3: Now, we have come to the end of our A-B branch and we move to the n-1th node which is A. We will now look at the adjacent node of A which is S. Visit S and push it onto the stack. Now you have to traverse the S-C-D branch, up to the depth ie upto D and mark S, C, D as visited.
Stack: A B S C D
Step 4: Since D has no other adjacent nodes, move back to C and traverse its adjacent branch E-H-G to the depth and push them onto the stack.
Stack : A B S C D E H G
Step 5: On reaching D, there is only one adjacent node ie F which is not visited. Push F onto the stack as well.

Stack : A B S C D E H G F

This stack itself is the traversal of the DFS.

Pseudocode

```
1. DFS(G,v)   ( v is the vertex where the search starts )
2.        Stack S := {};   ( start with an empty stack )
3.        for each vertex u, set visited[u] := false;
4.        push S, v;
5.        while (S is not empty) do
6.           u := pop S;
7.           if (not visited[u]) then
8.              visited[u] := true;
```

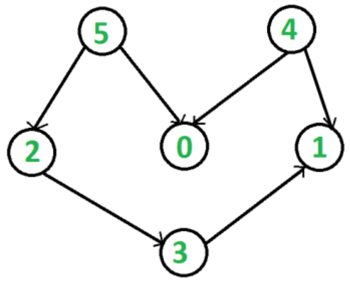
```
9.          for each unvisited neighbour w of uu
10.            push S, w;
11.         end if
12.       end while
13.    END DFS()
```

Code:

```
graph1 = {
    'A' : ['B','S'],
    'B' : ['A'],
    'C' : ['D','E','F','S'],
    'D' : ['C'],
    'E' : ['C','H'],
    'F' : ['C','G'],
    'G' : ['F','S'],
    'H' : ['E','G'],
    'S' : ['A','C','G']
}

def dfs(graph, node, visited):
    if node not in visited:
        visited.append(node)
        for k in graph[node]:
            dfs(graph,k, visited)
    return visited

visited = dfs(graph1,'A', [])
print(visited)
```

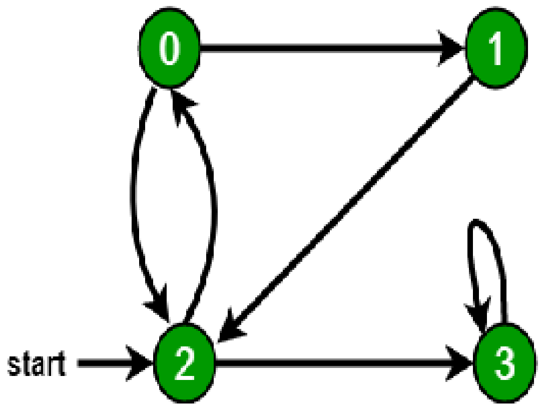
Lab exercises:

Q.1) Implement topological sorting using DFS algorithm for the following graph.



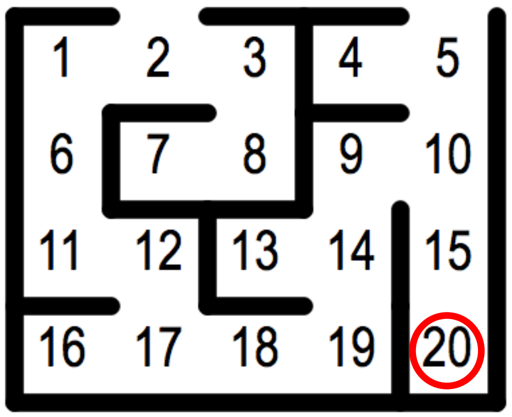
Note: Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering.
For example, a topological sorting of the following graph is "5 4 2 3 1 0". There can be more than one topological sorting for a graph. Another topological sorting of the following graph is "4 5 2 3 1 0". The first vertex in topological sorting is always a vertex with an in-degree of 0 (a vertex with no incoming edges).

Q.2) Consider the following directed graph for detecting cycles in the graph using DFS algorithm using Python.



Q.3) Write a Python program to solve the maze problem using DFS algorithm. The following is the problem statement and algorithm for the maze problem. Con

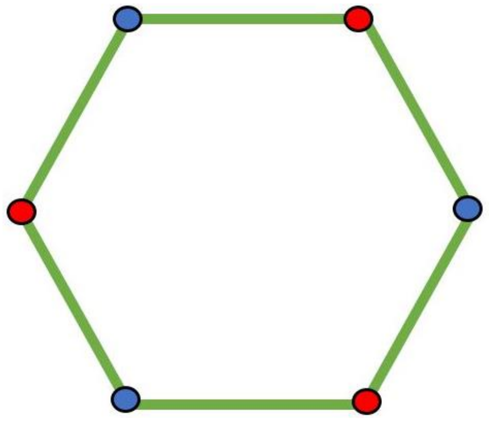
1. Enter the maze
2. If you have multiple ways, choose anyone and move forward
3. Keep choosing a way which was not seen so far till you exit the maze or reach dead end
4. If you exit maze, you are done.
5. If you reach dead end, this is wrong path, so take one step back, choose different path. If all paths are seen in this, take one step back and repeat



Additional questions:

Q.1) Write a Python program to solve 3x3 sudoku with Depth First Search algorithm.

Q.2) Write a Python code to check if a given graph is Bipartite using DFS.



Note: A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color.

Lab 4 – Implementation of Breadth First Search

Breadth First Search

The breadth-first search (BFS) algorithm is used to search a tree or graph data structure for a node that meets a set of criteria. It starts at the tree's root or graph and searches/visits all nodes at the current depth level before moving on to the nodes at the next depth level.

Breadth- First -Search:

Consider the state space of a problem that takes the form of a tree. Now, if we search the goal along each breadth of the tree, starting from the root and continuing up to the largest depth, we call it *breadth first search*.

Algorithm:

```
1. Create a variable called NODE-LIST and set it to initial state
2. Until a goal state is found or NODE-LIST is empty do
    a. Remove the first element from NODE-LIST and call it E. If NODE-LIST
       was empty, quit
    b. For each way that each rule can match the state described in E do:
        i.   Apply the rule to generate a new state
        ii.  If the new state is a goal state, quit and return this state
        iii. Otherwise, add the new state to the end of NODE-LIST
```

BFS illustrated:

Step 1: Initially fringe contains only one node corresponding to the source state A.

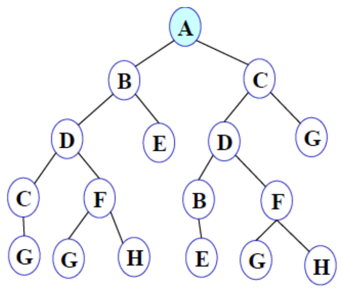


Figure 1

FRINGE: A

Step 2: A is removed from fringe. The node is expanded, and its children B and C are generated. They are placed at the back of fringe.

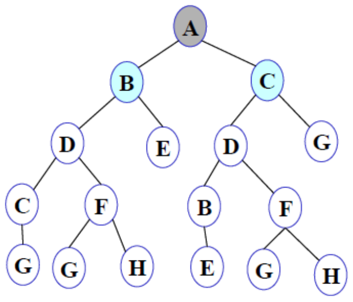


Figure 2

FRINGE: B C

Step 3: Node B is removed from fringe and is expanded. Its children D, E are generated and put at the back of fringe.

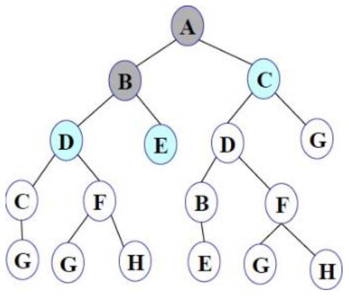


Figure 3

FRINGE: C D E

Step 4: Node C is removed from fringe and is expanded. Its children D and G are added to the back of fringe.

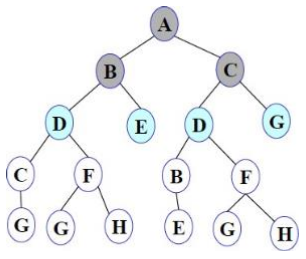


Figure 4

FRINGE: D E D G

Step 5: Node D is removed from fringe. Its children C and F are generated and added to the backof fringe.

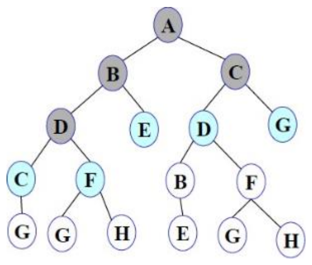


Figure 5

FRINGE: E D G C F

Step 6: Node E is removed from fringe. It has no children.

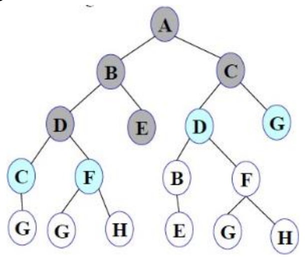


Figure 6

FRINGE: D G C F

Step 7: D is expanded; B and F are put in OPEN.

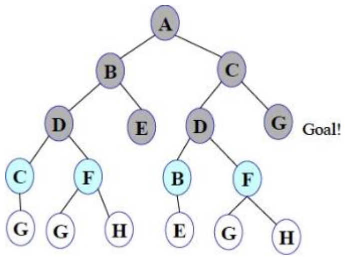
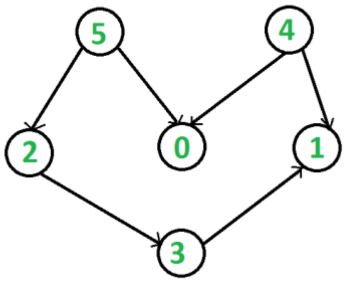


Figure 7

FRINGE: G C F B F

Lab Exercise 1:

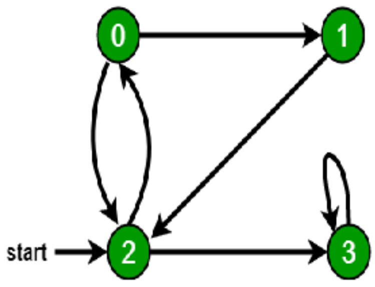
Q.1) Implement topological sorting using BFS algorithm for the following graph.



Note: Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering.

For example, a topological sorting of the following graph is "5 4 2 3 1 0". There can be more than one topological sorting for a graph. Another topological sorting of the following graph is "4 5 2 3 1 0". The first vertex in topological sorting is always a vertex with an in-degree of 0 (a vertex with no incoming edges).

Q.2) Consider the following directed graph for detecting cycles in the graph using BFS algorithm using Python.



Q3. Write python code for Traveling Salesman Problem (TSP) using Breadth First Search (BFS). Graph Given Below.

```
graph = {
    'A': {'B': 2, 'C': 3, 'D': 1},
    'B': {'A': 2, 'C': 4, 'D': 2},
    'C': {'A': 3, 'B': 4, 'D': 3},
    'D': {'A': 1, 'B': 2, 'C': 3}
}
```

Additional Exercise: Write a Python program to solve 3x3 sudoku with Depth First Search algorithm

Lab 05 – Implementation of Uniform cost search

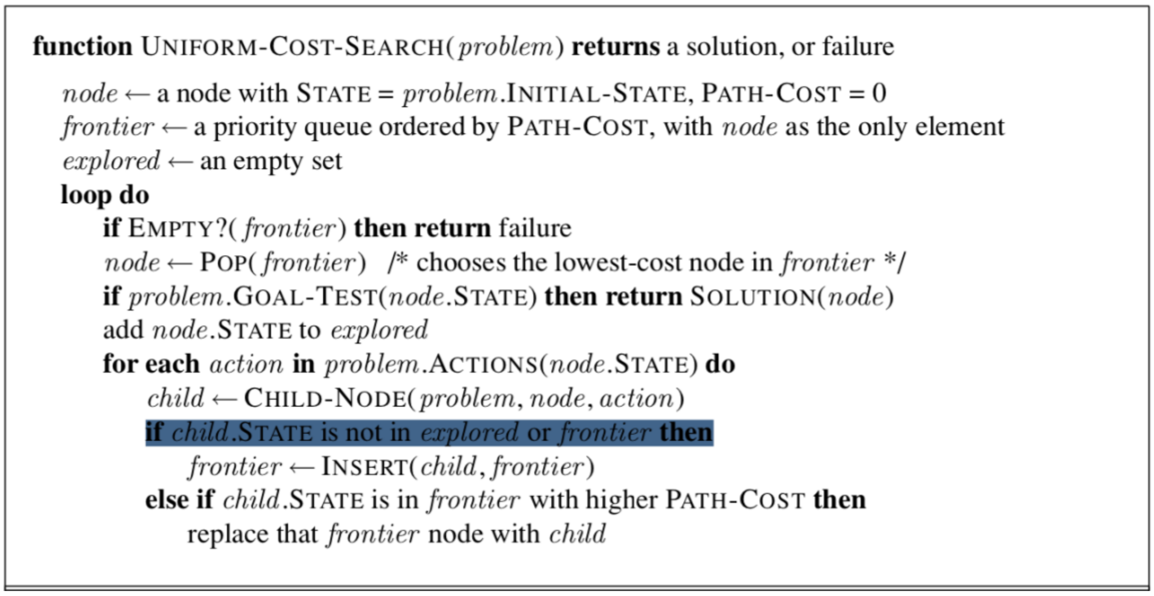
Uniform Cost Search is an algorithm used to move around a directed weighted search space to go from a start node to one of the ending nodes with a minimum cumulative cost. This search is an uninformed search algorithm since it operates in a brute-force manner, i.e. it does not take the state of the node or search space into consideration. It is used to find the path with the lowest cumulative cost in a weighted graph where nodes are expanded according to their cost of traversal from the root node. This is implemented using a priority queue where lower the cost higher is its priority.

Algorithm of Uniform Cost Search

- Insert RootNode into the queue.
- Repeat till queue is not empty:
- Remove the next element with the highest priority from the queue.
- If the node is a destination node, then print the cost and the path and exit

else insert all the children of removed elements into the queue with their cumulative cost as their priorities.

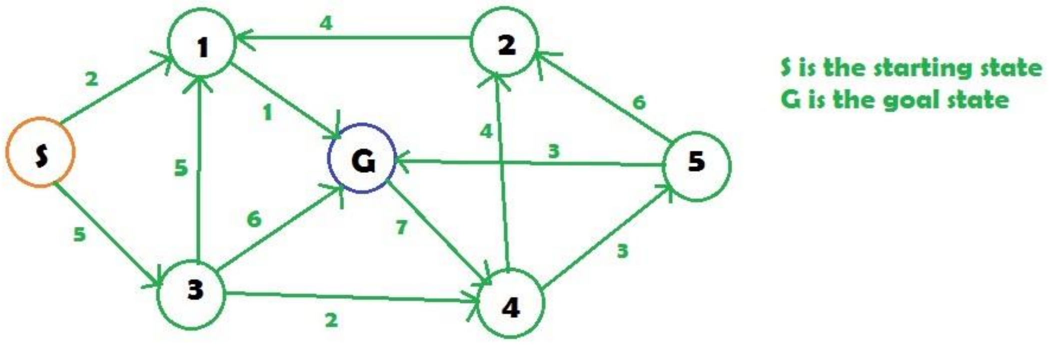
Pseudocode:



```
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
```

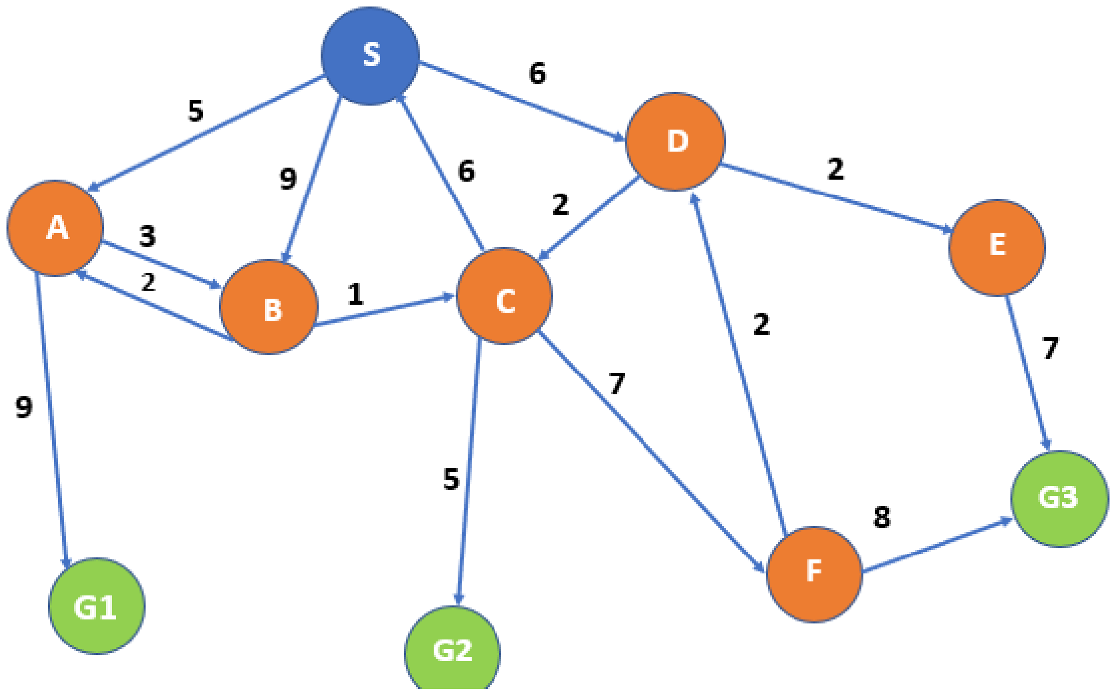
Lab Exercises:

Q.1) Write a python program to find the best path between node *s* and *g* from the given graph using Uniform Cost Search algorithm.

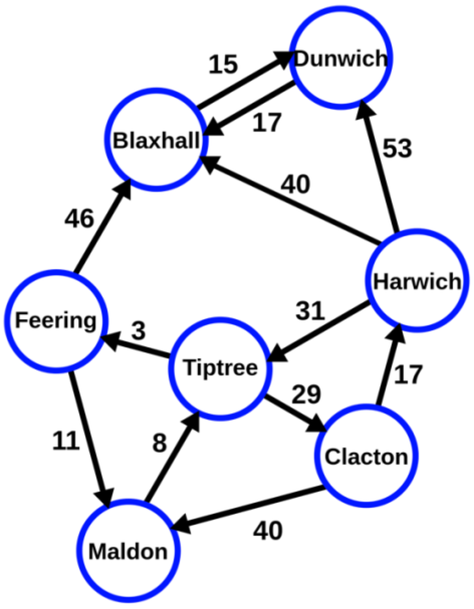


Output :
Minimum cost from S to G is =3

Q.2) Implement Uniform Cost Search algorithm for the following graph to find the goal (G1, G2 or G3) with the least cumulative cost from the source (S).

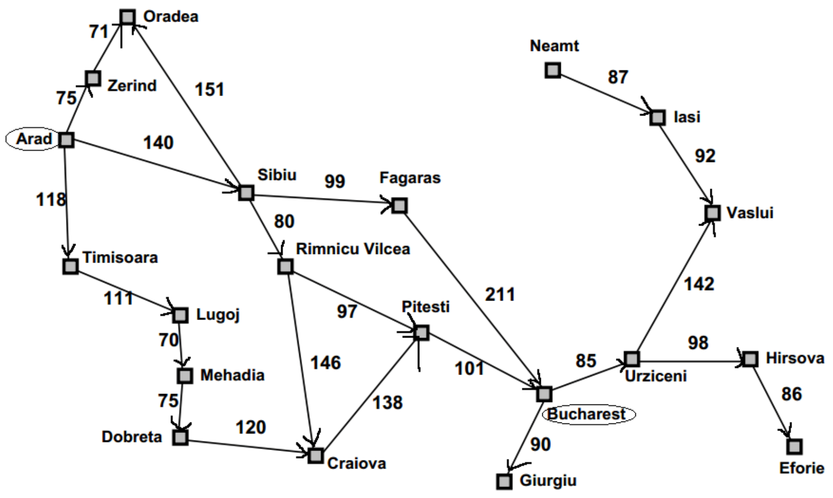


Q. 3) Find the shortest path between "Maldon" and "Dunwich" from the following graph using Uniform Cost search algorithm.



Additional Questions:

Q.1) Write a python program to find the best route between any 2 cites of the given road map using Uniform Cost Search algorithm.



Lab 06 – Implementation of Hill climbing search

HILL CLIMBING PROCEDURE:

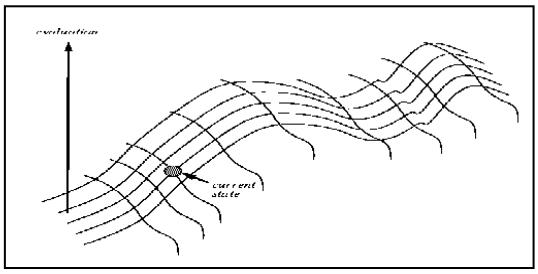
Hill Climbing Algorithm

We will assume we are trying to maximize a function. That is, we are trying to find a point in the search space that is better than all the others. And by "better" we mean that the evaluation is higher. We might also say that the solution is of better quality than all the others.

The idea behind hill climbing is as follows.

1. Pick a random point in the search space.
2. Consider all the neighbors of the current state.
3. Choose the neighbor with the best quality and move to that state.
4. Repeat 2 thru 4 until all the neighboring states are of lower quality.
5. Return the current state as the solution state.

We can also present this algorithm as follows (it is taken from the AIMA book (Russell, 1995) and follows the conventions we have been using on this course



when looking at blind and heuristic searches).

Algorithm:

```
Function HILL-CLIMBING(Problem)
        returns a solution stateInputs:
                            Problem,  problem
        Local variables:Current, a node
                Next, a node
Current = MAKE-NODE(INITIAL-STATE[Problem])
Loop do
    Next = a highest-valued successor of
                Current
        If VALUE[Next] < VALUE[Current]
        then returnCurrentCurrent = Next
End
```

Write a single python program to solve the Hill climbing search problem.

a. Let A, B to M represent a state in solution space.
   State space moves are given below. For Example A5 means A is node and 5 is its heuristics values.
   A5 to T11 , B13 and C21
   B13 to D27 and E3
   C21 to F25 and G4
   D27 to H101 and I99
   F25 to J67
   G4 to K99 and L3
   H101 ,I99,J67 to M17

   Start from {A} and Goal Node is {H}.

b. Sample out put: Initial Point=[‘A’,5]
   Start =[T,11]
   Sorted Child List=[[D,27][B,13]]

N=[D,27]
Child List=[[H,101],[I,99]]
Sorted Child List=[[H,101],[I,99]]
Closed=[[T,11],[D,27]]

N=[H,101]
Child List=[M,17]
Sorted Child List=[M,17]
Closed=[[T,11],[D,27],[H,101]

# Python4 program for the above approach

```
SuccList ={ 'A':[['B',3],['C',2]], 'B':[['D',2],['E',3]], 'C':[['F',2],['G',4]],
'D':[['H',1],['I',99]],'F': [['J',1]]
,'G':[['K',99],['L',3]]}
Start='A'

Closed = list()
SUCCESS=True
FAILURE=False


def MOVEGEN(N):
        New_list=list()
        if N in SuccList.keys():
                New_list=SuccList[N]

        return New_list

def SORT(L):
        L.sort(key = lambda x: x[1])
        return L

def heu(Node): #Node = ['B',2]--> [Node[0],Node[1]]
        return Node[1]
```

```
def APPEND(L1,L2):
    New_list=list(L1)+list(L2)
    return New_list

def Hill_Climbing(Start):
    global Closed
    N=Start
    CHILD = MOVEGEN(N)
    SORT(CHILD)
    N=[Start,5]
    print("\nStart=",N)
    print("Sorted Child List=",CHILD)
    newNode=CHILD[0]
    CLOSED=[N]

    while (heu(newNode) < heu(N)) and (len(CHILD) !=0):
        print("\n-------------------------")
        N= newNode
        print("N=",N)
        CLOSED = APPEND(CLOSED,[N])
        CHILD = MOVEGEN(N[0])
        SORT(CHILD)
        print("Sorted Child List=",CHILD)
        print("CLOSED=",CLOSED)
        newNode=CHILD[0]

    Closed=CLOSED

#Driver Code
Hill_Climbing(Start) #call search algorithm
```

Lab Exercise 1: Write python code to find maximum value of f(x) where -10 <= x <= 10) using Hill Climbing method.

Lab Exercise 2: Write a Python program to Hill Climbing Search to solve the 8-Queens problem:

Lab 07 – Implementation of A* Algorithm

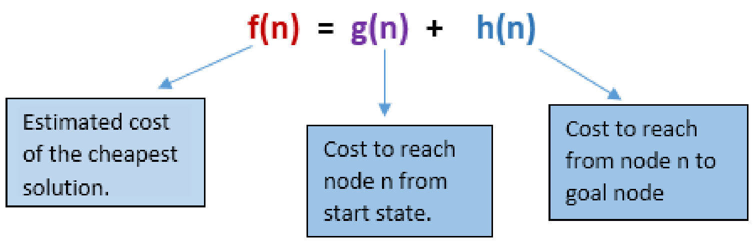
What is an A* Algorithm?
It is a searching algorithm that is used to find the shortest path between an initial and a final point.

It is a handy algorithm that is often used for map traversal to find the shortest path to be taken. A* was initially designed as a graph traversal problem, to help build a robot that can find its own course. It still remains a widely popular algorithm for graph traversal.

It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem.

Another aspect that makes A* so powerful is the use of weighted graphs in its implementation. A weighted graph uses numbers to represent the cost of taking each path or course of action. This means that the algorithms can take the path with the least cost, and find the best route in terms of distance and time.
Why A* Search Algorithm?

A* Search Algorithm is a simple and efficient search algorithm that can be used to find the optimal path between two nodes in a graph. It will be used for the shortest path finding. It is an extension of Dijkstra's shortest path algorithm (Dijkstra's Algorithm). The extension here is that, instead of using a priority queue to store all the elements, we use heaps (binary trees) to store them. The A* Search Algorithm also uses a heuristic function that provides additional information regarding how far away from the goal node we are. This function is used in conjunction with the f-heap data structure in order to make searching more efficient.


f(n) = g(n) + h(n)
Estimated cost of the cheapest solution.
Cost to reach node n from start state.
Cost to reach from node n to goal node

Algorithm of A* search:

Step1: Place the starting node in the OPEN list.

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

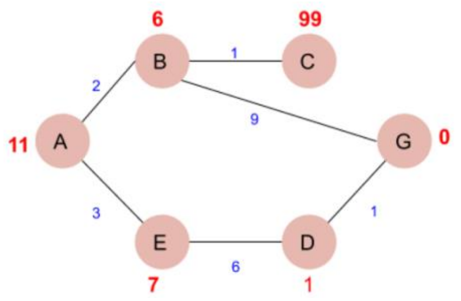
Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise

Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.

Step 6: Return to **Step 2**.

Sample Input:



```
def heuristic(n):
    H_dist = {
        'A': 11,
        'B': 6,
        'C': 99,
        'D': 1,
        'E': 7,
        'G': 0,
    }
    return H_dist[n]
```

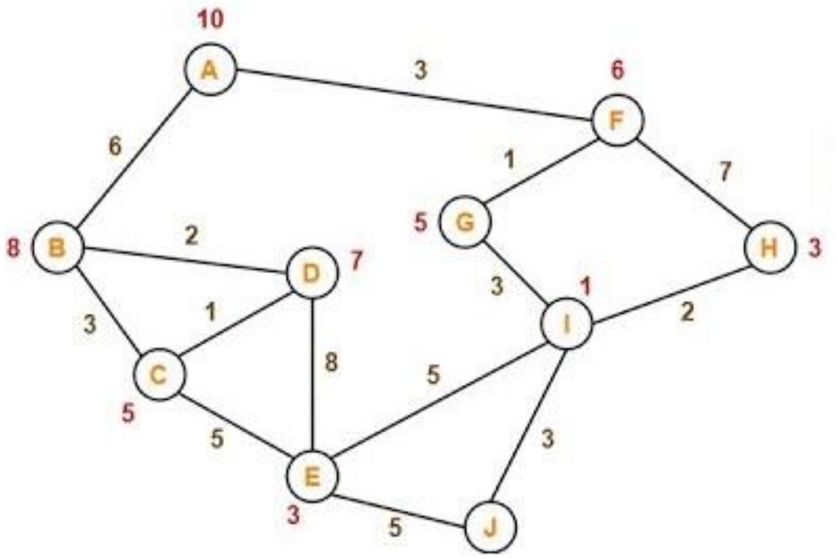
```
#Describe your graph here
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('A', 2), ('C', 1), ('G', 9)],
    'C': [('B', 1)],
    'D': [('E', 6), ('G', 1)],
    'E': [('A', 3), ('D', 6)],
    'G': [('B', 9), ('D', 1)]
}

aStarAlgo('A', 'G')
```

Output:

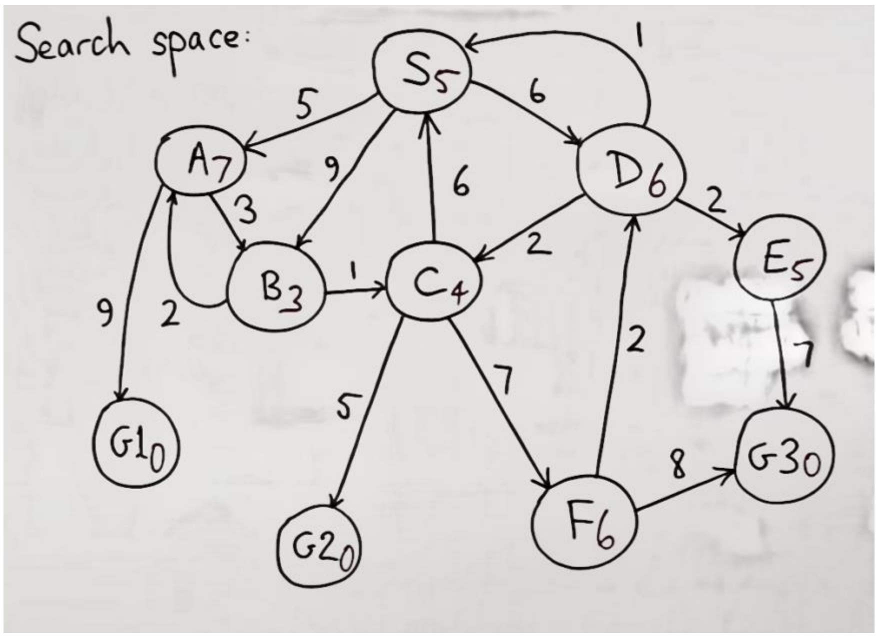
```
Path found: ['A', 'E', 'D', 'G']
```

Lab Exercises:

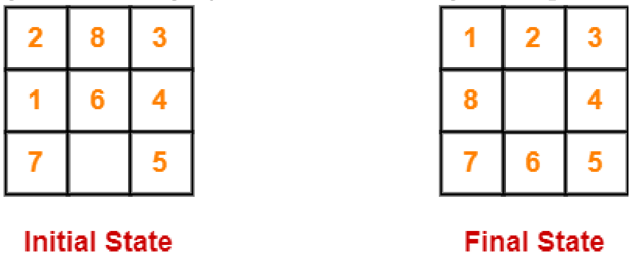
Q.1) Write a Python program to implement A* algorithm. Consider the following graph to find the path between **A** and **J**.



Q. 2) Write a Python program to implement A* algorithm. Consider the following graph to find the path between **S** and one of the goal node which has minimum cost.



Q. 2) Implement A* algorithm using Python to solve the given 8 puzzle problem.



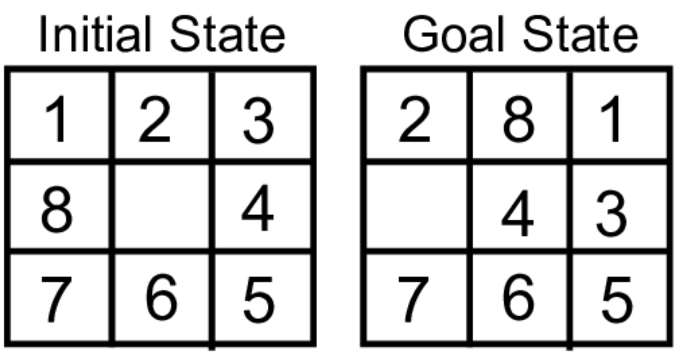
2	8	3
1	6	4
7		5

Initial State

1	2	3
8		4
7	6	5

Final State

Sample Solution:



Initial State

1	2	3
8		4
7	6	5

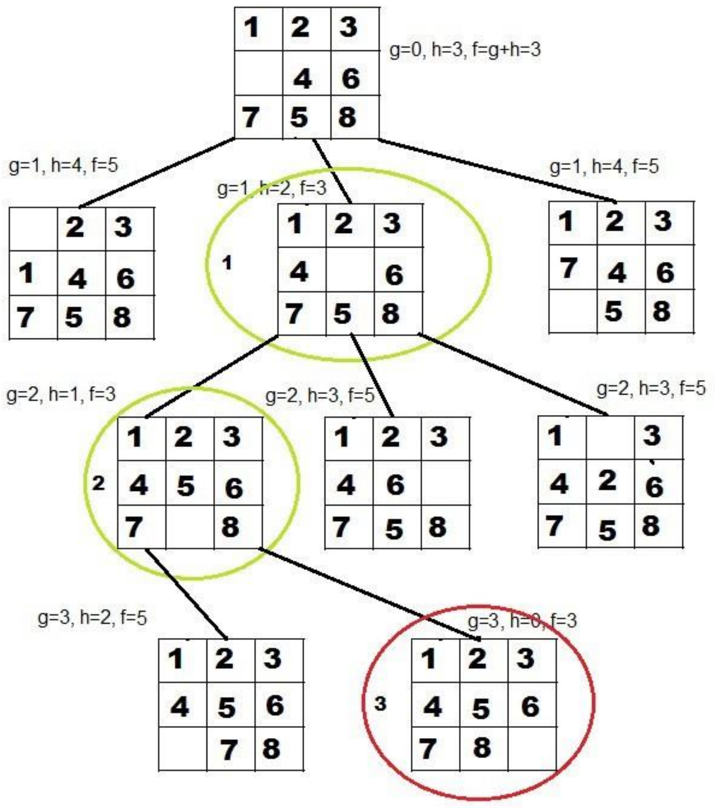
Goal State

2	8	1
	4	3
7	6	5

f(n) = h(n) + g(n)

g(n) – number of nodes traversed from start node to get to the current node.

h(n) – number of misplaced tiles



Additional Questions:

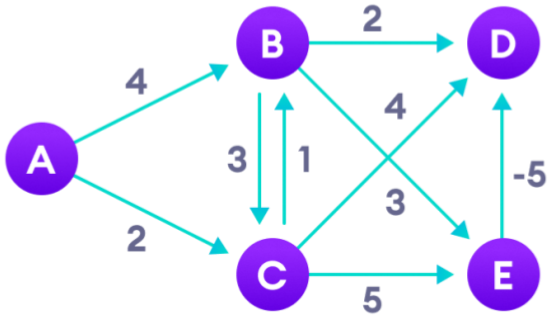
Q.1) Implement Bellman Ford algorithm using Python to solve the given problem.

Note: Bellman Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph. Bellman Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices. Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths.

Pseudocode:

```
function bellmanFord(G, S)
  for each vertex V in G
    distance[V] <- infinite
      previous[V] <- NULL
```

```
distance[S] <- 0

for each vertex V in G
  for each edge (U,V) in G
    tempDistance <- distance[U] + edge_weight(U, V)
    if tempDistance < distance[V]
      distance[V] <- tempDistance
      previous[V] <- U

for each edge (U,V) in G
  If distance[U] + edge_weight(U, V) < distance[V}
    Error: Negative Cycle Exists

return distance[], previous[]
```



Lab 08 – Implementation of Crypt Arithmetic

Input: arr[][] = {“SEND”, “MORE”}, S = “MONEY”
Output: Yes
***Explanation**:*
One of the possible ways is:

1. *Map the characters as the following, ‘S’→ 9, ‘E’→5, ‘N’→6, ‘D’→7, ‘M’→1, ‘O’→0, ‘R’→8, ‘Y’→2.*
2. *Now, after encoding the strings “SEND”, “MORE”, and “MONEY”, modifies to 9567, 1085 and 10652 respectively.*
3. *Thus, the sum of the values of “SEND” and “MORE” is equal to (9567+1085 = 10652), which is equal to the value of the string “MONEY”.*

Write a single python program to below problem.

a. Given an array of strings, **arr[]** of size **N** and a string **S**, the task is to find if it is possible to map integers value in the range **[0, 9]** to every alphabet that occurs in the strings, such that the sum obtained after summing the numbers formed by encoding all strings in the array is equal to the number formed by the string **S**.
b. **Input:** arr[][] = {“SEND”, “MORE”}, S = “MONEY”
Output: Yes
c. Input: **arr[][] = {“SIX”, “SEVEN”, “SEVEN”}, S = “TWENTY”**
Output: **Yes**

```
# Python6 program for the above approach

# Function to check if the
# assignment of digits to
# characters is possible
def isSolvable(words, result):
        # Stores the value
        # assigned to alphabets
        mp = [-1]*(26)
```

```
# Stores if a number
# is assigned to any
# character or not
used = [0]*(10)

# Stores the sum of position
# value of a character
# in every string
Hash = [0]*(26)

# Stores if a character
# is at index 0 of any
# string
CharAtfront = [0]*(26)

# Stores the string formed
# by concatenating every
# occurred character only
# once
uniq = ""

# Iterator over the array,
# words
for word in range(len(words)):
        # Iterate over the string,
        # word
        for i in range(len(words[word])):
                # Stores the character
                # at ith position
                ch = words[word][i]

                # Update Hash[ch-'A]
                Hash[ord(ch) - ord('A')] += pow(10, len(words[word]) - i - 1

                # If mp[ch-'A'] is -1
                if mp[ord(ch) - ord('A')] == -1:
                        mp[ord(ch) - ord('A')] = 0
                uniq += str(ch)

            # If i is 0 and word
            # length is greater
            # than 1
            if i == 0 and len(words[word]) > 1:
                CharAtfront[ord(ch) - ord('A')] = 1

    # Iterate over the string result
    for i in range(len(result)):
        ch = result[i]

        Hash[ord(ch) - ord('A')] -= pow(10, len(result) - i - 1)

        # If mp[ch-'A] is -1
        if mp[ord(ch) - ord('A')] == -1:
            mp[ord(ch) - ord('A')] = 0
            uniq += str(ch)

        # If i is 0 and length of
        # result is greater than 1
        if i == 0 and len(result) > 1:
            CharAtfront[ord(ch) - ord('A')] = 1

    mp = [-1]*(26)

    # Recursive call of the function
    return True

# Auxiliary Recursive function
# to perform backtracking
def solve(words, i, S, mp, used, Hash, CharAtfront):
    # If i is word.length
    if i == len(words):
        # Return true if S is 0
        return S == 0
        # Stores the character at
        # index i
        ch = words[i]

        # Stores the mapped value
        # of ch
        val = mp[ord(words[i]) - ord('A')]

        # If val is -1
        if val != -1:
                # Recursion
                return solve(words, i + 1, S + val * Hash[ord(ch) - ord('A')], mp, used,
Hash, CharAtfront)

        # Stores if there is any
        # possible solution
        x = False

        # Iterate over the range
        for l in range(10):
                # If CharAtfront[ch-'A']
                # is true and l is 0
                if CharAtfront[ord(ch) - ord('A')] == 1 and l == 0:
                        continue

                # If used[l] is true
                if used[l] == 1:
                        continue

                # Assign l to ch
                mp[ord(ch) - ord('A')] = l

                # Marked l as used
                used[l] = 1

                # Recursive function call
                x |= solve(words, i + 1, S + l * Hash[ord(ch) - ord('A')], mp, used,
Hash, CharAtfront)

                # Backtrack
                mp[ord(ch) - ord('A')] = -1

                # Unset used[l]
                used[l] = 0

        # Return the value of x;
        return x

arr = [ "SIX", "SEVEN", "SEVEN" ]
S = "TWENTY"

# Function Call
if isSolvable(arr, S):
        print("Yes")
else:
        print("No")
```

Lab Exercise 1:
Write python code for arithmetic problem CROSS + ROADS = DANGER

Lab Exercise 2:

Write python code for arithmetic problem DONALD+GERALD=ROBERT

Lab Exercise 3:
Write python code for arithmetic problem MIT + MANIPAL = MITMAHE

Lab 9 – Implementation of Water jug problem

Water Jug Problem:

Problem: You are given two jugs, a 4-gallon one and a 3-gallon one.Neither has any measuring mark on it.There is a pump that can be used to fill the jugs with water.How can you get exactly 2 gallons of water into the 4-gallon jug.

Solution:
The state space for this problem can be described as the set of ordered pairs of integers **(x,y)**
Where,
X represents the quantity of water in the 4-gallon jug **X= 0,1,2,3,4**
Y represents the quantity of water in 3-gallon jug **Y=0,1,2,3**
Start State: (0,0)
Goal State: (2,0)
Generate production rules for the water jug problem
Production Rules:

Rule	State	Process
1	(X,Y | X<4)	(4,Y) {Fill 4-gallon jug}
2	(X,Y |Y<3)	(X,3) {Fill 3-gallon jug}
3	(X,Y |X>0)	(0,Y) {Empty 4-gallon jug}
4	(X,Y | Y>0)	(X,0) {Empty 3-gallon jug}
5	(X,Y | X+Y>=4 ^ Y>0)	(4,Y-(4-X)) {Pour water from 3-gallon jug into 4-gallon jug until 4-gallon jug is full}
6	(X,Y | X+Y>=3 ^X>0)	(X-(3-Y),3) {Pour water from 4-gallon jug into 3-gallon jug until 3-gallon jug is full}

7	(X,Y | X+Y<=4 ^Y>0)	(X+Y,0) {Pour all water from 3-gallon jug into 4-gallon jug}
8	(X,Y | X+Y <=3^ X>0)	(0,X+Y) {Pour all water from 4-gallon jug into 3-gallon jug}
9	(0,2)	(2,0) {Pour 2 gallon water from 3 gallon jug into 4 gallon jug}

Initialization:
Start State: **(0,0)**
Apply Rule 2:
(X,Y | Y<3) ->
(X,3)
{Fill 3-gallon jug}
Now the state is **(X,3)**

Iteration 1:
Current State: **(X,3)**
Apply Rule 7:
(X,Y | X+Y<=4 ^Y>0)
(X+Y,0)
{Pour all water from 3-gallon jug into 4-gallon jug}
Now the state is **(3,0)**

Iteration 2:
Current State **: (3,0)**
Apply Rule 2:
(X,Y | Y<3) ->
(3,3)
{Fill 3-gallon jug}
Now the state is **(3,3)**

Iteration 3:
Current State**:(3,3)**
Apply Rule 5:
(X,Y | X+Y>=4 ^ Y>0)
(4,Y-(4-X))
{Pour water from 3-gallon jug into 4-gallon jug until 4-gallon jug is full}

Now the state is **(4,2)**

Iteration 4:
Current State : **(4,2)**
Apply Rule 3:
(X,Y | X>0)
(0,Y)
{Empty 4-gallon jug}
Now state is **(0,2)**

Iteration 5:
Current State : **(0,2)**
Apply Rule 9:
(0,2)
(2,0)
{Pour 2 gallon water from 3 gallon jug into 4 gallon jug}
Now the state is **(2,0)**

Goal Achieved.

Lab Exercises:

Q.1) Write a Python program to solve the water jug problem using Breadth First Search algorithm.

Q.2) Write a Python program to solve the water jug problem using Depth First Search algorithm.

Additional Questions:
Q.1) Write a Python program to solve the water jug problem using Memoization algorithm.

Note: Approach: Using Recursion, visit all the six possible moves one by one until one of them returns True. Since there can be repetitions of same recursive calls, hence every return value is stored using memoization to avoid calling the recursive function again and returning the stored value.

Lab 10 – Implementation of Missionaries and Cannibals problem

Write a single python program to below problem

a. *On one bank of a river are* **3** *missionaries and* **3** *cannibals. There is* **1** *boat available that can carry at most* **2** *people and that they would like to use to cross the river. If the cannibals ever outnumber the missionaries on either of the river's banks or on the boat, the missionaries will get eaten. How can the boat be used to carry all the missionaries and cannibals across the river safely?*

b. The below constraint should be satisfied for above problem are

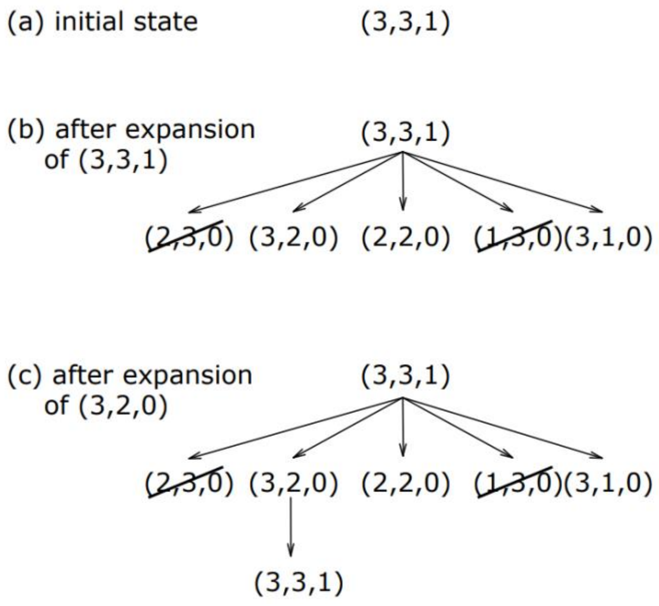
1. The boat can carry at most two people.

2. If cannibals numbers greater than missionaries then the cannibals would eat the missionaries.

3. The boat cannot cross the river by itself with no people on board.

Problem Formation:

State space: triple (x,y,z) with 0 x,y,z 3, wherex,y, and z represent the number of missionaries, cannibals and boats currently on the original bank.
Initial State: (3,3,1)
Successor function: From each state, either bring one missionary, one cannibal, two missionaries, two cannibals, or one of each type to the other bank.
Note: Not all states are attainable (e.g., (0,0,1)), andsome are illegal.
Goal State: (0,0,0)
Path Costs: 1 unit per crossing



#Python8 program to illustrate Missionaries & cannibals Problem

```
print("\n")
print("\tGame Start\nNow the task is to move all of them to right side of the river")
print("rules:\n1. The boat can carry at most two people\n2. If cannibals num greater
than missionaries then the cannibals would eat the missionaries\n3. The boat cannot
cross the river by itself with no people on board")
lM = 3          #lM = Left side Missionaries number
lC = 3          #lC = Laft side Cannibals number
rM=0            #rM = Right side Missionaries number
rC=0            #rC = Right side cannibals number
userM = 0       #userM = User input for number of missionaries for right to left side
travel
userC = 0       #userC = User input for number of cannibals for right to left travel
k = 0
print("\nM M M C C C |      --- | \n")
try:
        while(True):
                while(True):
                        print("Left side -> right side river travel")
                        #uM = user input for number of missionaries for left to right
```

```
                    travel
                #uC = user input for number of cannibals for left to right travel
                uM = int(input("Enter number of Missionaries travel => "))
                uC = int(input("Enter number of Cannibals travel => "))

                if((uM==0)and(uC==0)):
                    print("Empty travel not possible")
                    print("Re-enter : ")
                elif(((uM+uC) <= 2)and((lM-uM)>=0)and((lC-uC)>=0)):
                    break
                else:
                    print("Wrong input re-enter : ")
            lM = (lM-uM)
            lC = (lC-uC)
            rM += uM
            rC += uC

            print("\n")
            for i in range(0,lM):
                print("M ",end="")
            for i in range(0,lC):
                print("C ",end="")
            print("| --> | ",end="")
            for i in range(0,rM):
                print("M ",end="")
            for i in range(0,rC):
                print("C ",end="")
            print("\n")

            k +=1

            if(((lC==3)and (lM ==
1))or((lC==3)and(lM==2))or((lC==2)and(lM==1))or((rC==3)and (rM ==
1))or((rC==3)and(rM==2))or((rC==2)and(rM==1))):
                print("Cannibals eat missionaries:\nYou lost the game")

                break
```

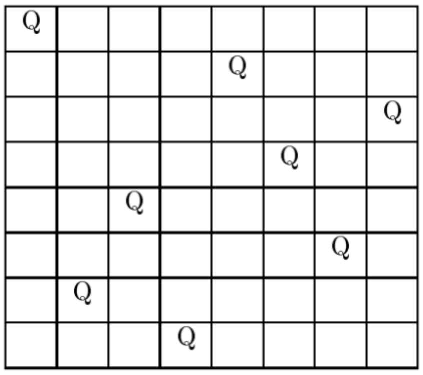
```
if((rM+rC) == 6):
    print("You won the game : \n\tCongrats")
    print("Total attempt")
    print(k)
    break
while(True):
    print("Right side -> Left side river travel")
    userM = int(input("Enter number of Missionaries travel => "))
    userC = int(input("Enter number of Cannibals travel => "))

    if((userM==0)and(userC==0)):
            print("Empty travel not possible")
            print("Re-enter : ")
    elif(((userM+userC) <= 2)and((rM-userM)>=0)and((rC-
userC)>=0)):
        break
    else:
        print("Wrong input re-enter : ")
lM += userM
lC += userC
rM -= userM
rC -= userC

k +=1
print("\n")
for i in range(0,lM):
    print("M ",end="")
for i in range(0,lC):
    print("C ",end="")
print("| <-- | ",end="")
for i in range(0,rM):
    print("M ",end="")
for i in range(0,rC):
    print("C ",end="")
print("\n")
```

```
                    if(((lC==3)and (lM ==
        1))or((lC==3)and(lM==2))or((lC==2)and(lM==1))or((rC==3)and (rM ==
        1))or((rC==3)and(rM==2))or((rC==2)and(rM==1))):
                            print("Cannibals eat missionaries:\nYou lost the game")
                            break
        except EOFError as e:
                print("\nInvalid input please retry !!")
```

Lab Exercise 1: Write python code for Python code for a Wumpus World game.

Lab 11 – Implementation of 8 queen's problem

The eight queens problem is the problem of placing eight queens on an 8×8 chessboard such that none of them attack one another (no two are in the same row, column, or diagonal). More generally, the n queens problem places n queens on an n×n chessboard.



Sample Solution for 4 queens problem using Backtracking

```
global N
N = 4

def printSolution(board):
        for i in range(N):
                for j in range(N):
                        print (board[i][j],end=' ')
                print()

# A utility function to check if a queen can
# be placed on board[row][col]. Note that this
# function is called when "col" queens are
# already placed in columns from 0 to col -1.
# So we need to check only left side for
# attacking queens
def isSafe(board, row, col):

    # Check this row on left side
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check lower diagonal on left side
    for i, j in zip(range(row, N, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def solveNQUtil(board, col):
    # base case: If all queens are placed
    # then return true
    if col >= N:
        return True
    # Consider this column and try placing
    # this queen in all rows one by one
    for i in range(N):

        if isSafe(board, i, col):
            # Place this queen in board[i][col]
            board[i][col] = 1

            # recur to place rest of the queens
            if solveNQUtil(board, col + 1) == True:
                return True

            # If placing queen in board[i][col
            # doesn't lead to a solution, then
            # queen from board[i][col]
            board[i][col] = 0

    # if the queen can not be placed in any row in
    # this column col then return false
    return False

# This function solves the N Queen problem using
# Backtracking. It mainly uses solveNQUtil() to
# solve the problem. It returns false if queens
# cannot be placed, otherwise return true and
```

```
# placement of queens in the form of 1s.
# note that there may be more than one
# solutions, this function prints one of the
# feasible solutions.
def solveNQ():
        board = [ [0, 0, 0, 0],
                        [0, 0, 0, 0],
                        [0, 0, 0, 0],
                        [0, 0, 0, 0]
                        ]

        if solveNQUtil(board, 0) == False:
                print ("Solution does not exist")
                return False

        printSolution(board)
        return True

# driver program to test above function
solveNQ()
```

Lab Exercises:

Q.1) Write a python program to solve 8 queens problem using Hill Climbing algorithm.

Sample Input: N = 8

Output:

```
0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0
```

Q. 2) Write a Python program to find all possible solution for 8 queens problem using Breadth First Search algorithm.

Expected output:

```
solution: (0, 4, 7, 5, 2, 6, 1, 3)
solution: (0, 5, 7, 2, 6, 3, 1, 4)
solution: (0, 6, 3, 5, 7, 1, 4, 2)
…
…
solution: (7, 3, 0, 2, 5, 1, 6, 4)
```

Additional Questions:

Q.1) Write a Python program to solve the N queens problem using genetic algorithm.

A genetic algorithm is a search heuristic that is inspired by Charles Darwin’s theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation.

Pseudocode:

```
START
Generate the initial population
Compute fitness
REPEAT
   Selection
   Crossover
   Mutation
   Compute fitness
UNTIL population has converged
STOP
```

Lab 12 – Implementation of Best First Search

Best First Search:

- A combination of depth first and breadth first searches.
- Depth first is good because a solution can be found without computing all nodes and breadth first is good because it does not get trapped in dead ends.
- The best first search allows us to switch between paths thus gaining the benefit of both approaches. At each step the most promising node is chosen. If one of the nodes chosen generates nodes that are less promising it is possible to choose another at the same level and in effect the search changes from depth to breadth. If on analysis these are no better than this previously unexpanded node and branch is not forgotten and the search method reverts to the

OPEN is a priorityqueue of nodes that have been evaluated by the heuristic function but which have notyet been expanded into successors. The most promising nodes are at the front.

CLOSED are nodes that have already been generated and these nodes must be stored because a graph isbeing used in preference to a tree.

Algorithm:

1. Start with OPEN holding the initial state
2. Until a goal is found or there are no nodes left on open do.

    - Pick the best node on OPEN
    - Generate its successors

- For each successor Do
    - If it has not been generated before ,evaluate it ,add it to OPEN and record itsparent.If it has been generated before change the parent if this new path is better and in that case update the cost of getting to any successor nodes.

3. If a goal is found or no more nodes left in OPEN, quit, else return to 2.

Write a single python program to solve the Best First Search Problem

a. State space moves are given below.
   A to [B,3],[C,2]
   B to [[A,5],[C,2],[D,2],[E,3]]
   C to [[A,5],[B,3],[F,2],[G,4]]
   D to [[H,1],[I,99]]
   F to [J,99]
   G to [K,99],[L,3]

   Start from {A} and Goal Node is {E}.

b. Sample out put: N=[‘A’,5]
   Closed [‘A’,5]
   Child [B,3],[C,2]
   Unsorted open [B,3],[C,2]
   Sorted open   [C,2],[B,3]

   N=[‘C’,2]
   Closed [‘A’,5], [C,2]
   Child [‘A’,5],[B,3],[F,2],[G,4]
   Unsorted open [F,2],[G,4] [B,3],
   Sorted open [F,2],[B,3], [G,4]

   N=[‘F’,2]
   Closed [‘A’,5], [C,2], [‘F’,2]
   Child [‘J’,99]
   Unsorted open [J,99], [B,3], [G,4]

Sorted open [B,3], [G,4], [J,99],

#Python10 program

```
SuccList ={ 'A':[['B',3],['C',2]], 'B':[['A',5],['C',2],['D',2],['E',3]],
'C':[['A',5],['B',3],['F',2],['G',4]], 'D':[['H',1],['I',99]],'F':
[['J',99]],'G':[['K',99],['L',3]]}
Start='A'
Goal='E'
Closed = list()
SUCCESS=True
FAILURE=False
State=FAILURE

def MOVEGEN(N):
        New_list=list()
        if N in SuccList.keys():
                New_list=SuccList[N]

        return New_list

def GOALTEST(N):
        if N == Goal:
                return True
        else:
                return False

def APPEND(L1,L2):
        New_list=list(L1)+list(L2)
        return New_list

def SORT(L):
        L.sort(key = lambda x: x[1])
        return L

def BestFirstSearch():
```

```
          OPEN=[[Start,5]]
          CLOSED=list()
          global State
          global Closed
          while (len(OPEN) != 0) and (State != SUCCESS):
                    print("------------")
                    N= OPEN[0]
                    print("N=",N)
                    del OPEN[0] #delete the node we picked

                    if GOALTEST(N[0])==True:
                              State = SUCCESS
                              CLOSED = APPEND(CLOSED,[N])
                              print("CLOSED=",CLOSED)
                    else:
                              CLOSED = APPEND(CLOSED,[N])
                              print("CLOSED=",CLOSED)
                              CHILD = MOVEGEN(N[0])
                              print("CHILD=",CHILD)
                              for val in CLOSED:
                                        if val in CHILD:
                                                  CHILD.remove(val)
                              for val in OPEN:
                                        if val in CHILD:
                                                  CHILD.remove(val)
                              OPEN = APPEND(CHILD,OPEN) #append movegen
elements to OPEN
                              print("Unsorted OPEN=",OPEN)
                              SORT(OPEN)
                              print("Sorted OPEN=",OPEN)

          Closed=CLOSED
          return State

#Driver Code
result=BestFirstSearch() #call search algorithm
print(Closed,result)
```

Lab Exercises: AI game called "Connect Four" with two players take turns dropping discs into a vertical grid with the goal of getting four discs in a row vertically, horizontally, or diagonally.

REFERENCES:

1. Stuart Russell and Peter Norvig -,Artificial Intelligence A Modern Approach", Pearson Education, Third Edition, 2016.,ÄØ,ÄØ,ÄØ
2. Elaine Rich, Kevin Knight, Shivashankar B. Nair, Artificial Intelligence, Third, Edition, Tata McGraw Hill Edition, 2010.,ÄØ,ÄØ
3. Saroj Kaushik-,Artificial Intelligence, Cengage Learning Publications, First Edition, 2011.,ÄØ
4. Don W. Patterson -,Introduction to, Artificial Intelligence and Expert Systems, PHI Publication,2006.,ÄØ