

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

**ARTIFICIAL INTELLIGENCE LABORATORY–CSE 2263**

**Department of Computer Science and Engineering**

**Compiled by**

**Dr. ANDREW J**

**Assistant Professor – Senior Scale**

**Dr. Nagaraj Naik**

**Assistant Professor – Senior Scale**

January 2023

**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:
  + Programs should be interactive with appropriate prompt messages, error messages if any, and descriptive messages for outputs.
  + 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:
  + Lab exercises – to be completed during lab hours
  + 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**

**CONTENTS**

|  |  |  |
| --- | --- | --- |
| **Sl. No.** | **Experiments** | **Page No.** |
| 1 | Week 01 : Implementation of Depth First Search | 05 |
| 2 | Week 02: Implementation of Breadth First Search | 07 |
| 3 | Week 03: Implementation of Uniform cost search | 09 |
| 4 | Week 04: Implementation of Hill climbing search | 10 |
| 5 | Week 05: Implementation of A\* Algorithm | 11 |
| 6 | Week 06: Implementation of Crypt Arithmetic | 13 |
| 7 | Week 07: Implementation of Water jug problem | 14 |
| 8 | Week 08: Implementation of Missionaries and Cannibals problem | 15 |
| 9 | Week 09: Implementation of 8 queen’s problem | 16 |
| 10 | Week 10: Implementation of Best First Search | 17 |
| 11 | Week 11: Design and Development of expert system for the specified domain | 18 |
| 12 | Week 12: Design and Development of expert system for the specified domain | 18 |

**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:**

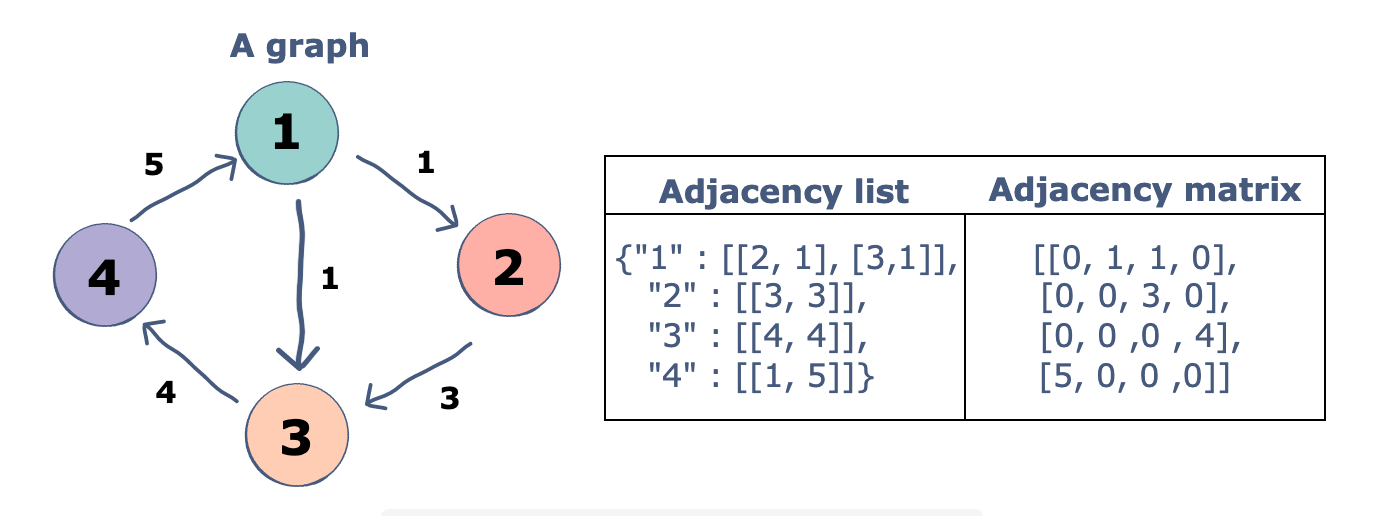
**Q1.** Implementation of a queue in Python using two stacks.

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.]

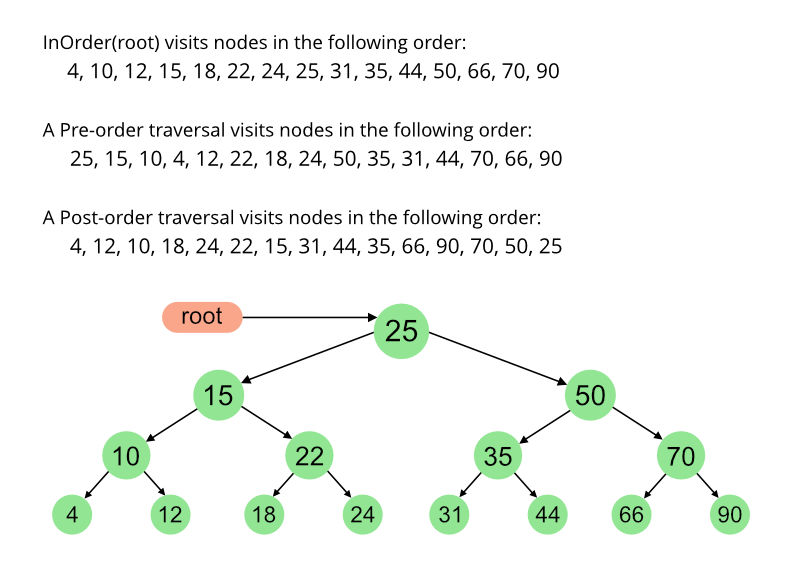


**Q3.** Create two list X and Y with some set of numerical values. Compute Euclidean distance for corresponding values in X and Y. Store the distance values in a separate list and sort them using Bubble sort algorithm.

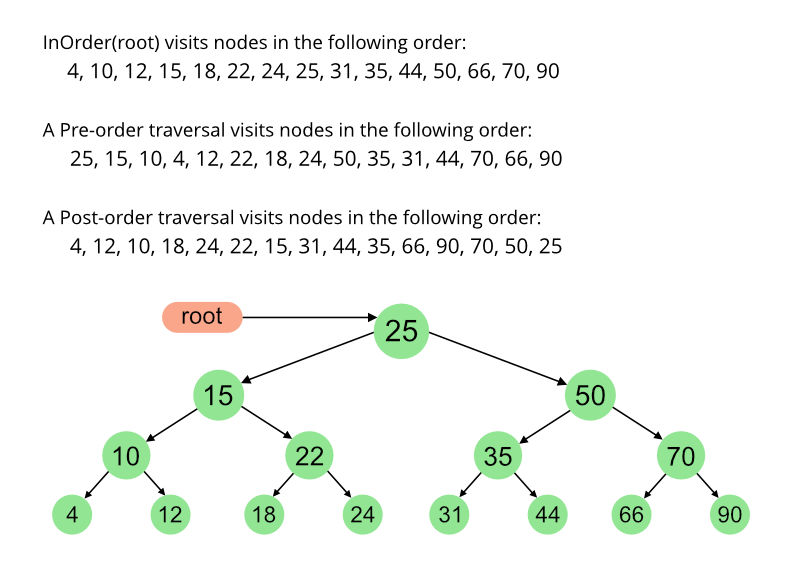
Text, application

Description automatically generated

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



**Expected Output:**



**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:

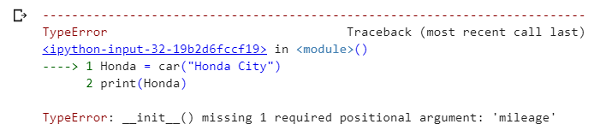
   def \_\_init\_\_(self, name, mileage):

      self.name = name

        self.mileage = mileage

Honda = Car("Honda City")

print(Honda)



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.

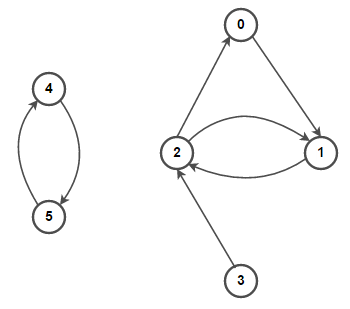
A set is unordered and declared in square brackets. It is not indexed and does not have duplicate entries as well.

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:**

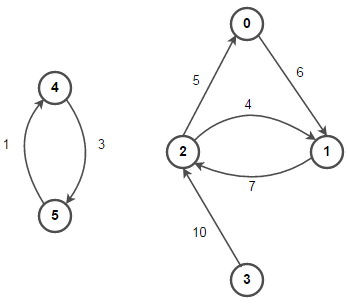
**Q1.** Implement the following directed unweighted graph using class, methods, and data structures of Python.



Expected output:

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

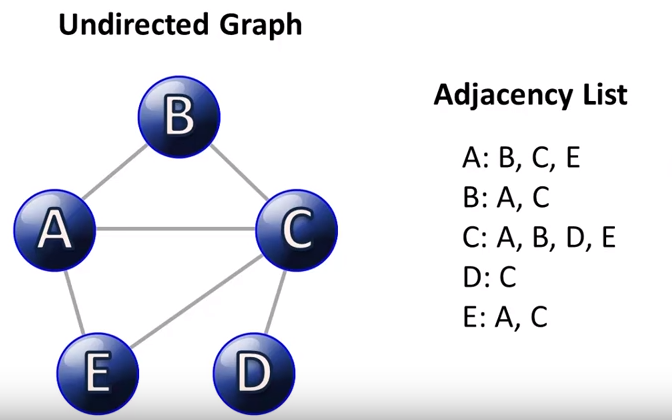
**Q2.** Implement the following directed weighted graph using class, methods, and data structures of Python.



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.

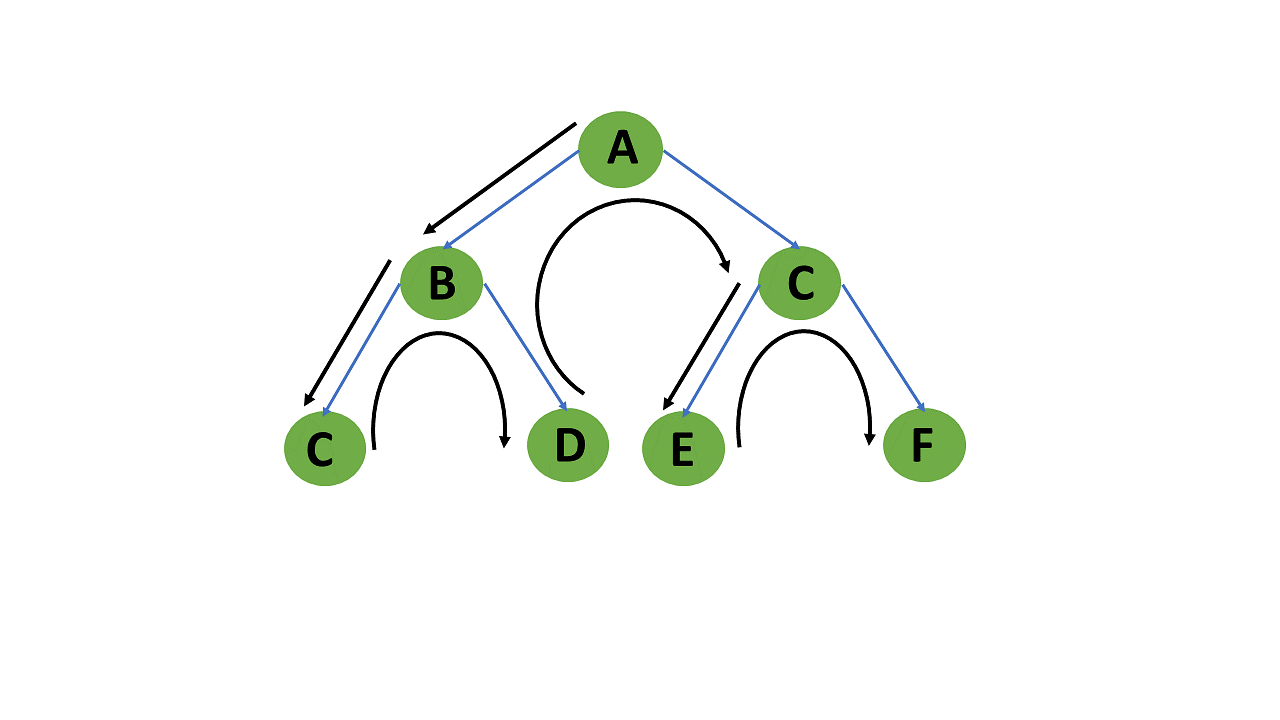
**Lab 3 - Implementation of Depth First Search**

**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.



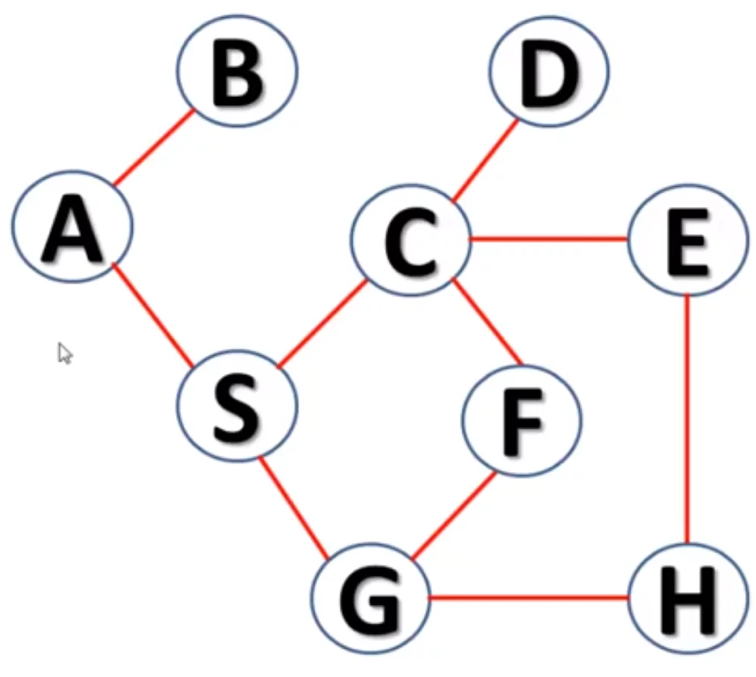
When a dead-end occurs in any iteration, the Depth First Search (DFS) method traverses a network in a deathward motion and uses a stack data structure to remember to acquire the next vertex to start a search.

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:**

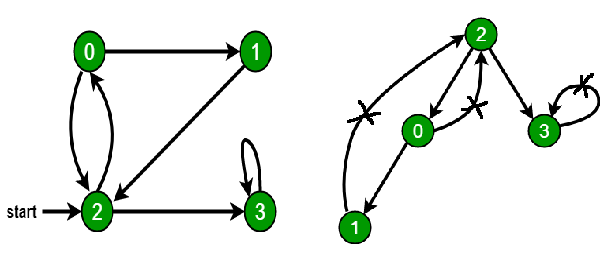
Q.1) Implement topological sorting using DFS algorithm for the following graph.



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.

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

**Pseudocode:**

*function DFS( node )*

*if you visited node:*

*continue*

*if node is ending:*

*return*

*mark node as visited*

*if node to west is valid:*

*DFS( node to west )*

*if node to north is valid:*

*DFS( node to north )*

*if node to east is valid:*

*DFS( node to east )*

*if node to south is valid:*

*DFS( node to south )*

*function valid( node )*

*if node.x < 0 or node.x >= width or node.y < 0 or node.y >= height or node is blocked:*

*return false*

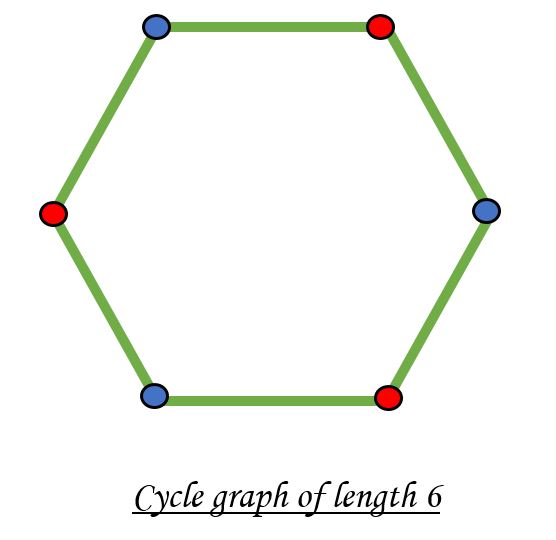
*else:*

*return true*

**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
   1. Remove the first element from NODE-LIST and call it E. If NODE-LIST was empty, quit
   2. For each way that each rule can match the state described in E do:
      1. Apply the rule to generate a new state
      2. If the new state is a goal state, quit and return this state
      3. 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.

Diagram, shape

Description automatically generated

##### 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.

A picture containing clock, watch

Description automatically generated

##### Figure 2

**FRINGE: B C**

A picture containing clock, watch

Description automatically generated**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.

A picture containing clock, watch

Description automatically generated

##### Figure 4

**FRINGE: D E D G**

A picture containing clock, watch

Description automatically generated**Step 5**: Node D is removed from fringe. Its children C and F are generated and added to the back of fringe.

##### Figure 5

**FRINGE: E D G C F**

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

A picture containing clock, watch

Description automatically generated

##### Figure 6

**FRINGE: D G C F**

A picture containing clock, watch

Description automatically generated**Step 7**: D is expanded; B and F are put in OPEN.

##### Figure 7

**FRINGE: G C F B F**

**Lab Exercise 1:**

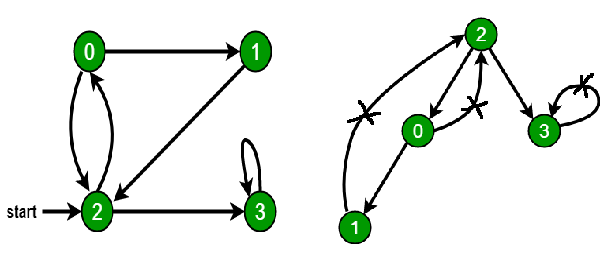
Q.1) Implement topological sorting using BFS algorithm for the following graph.



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**

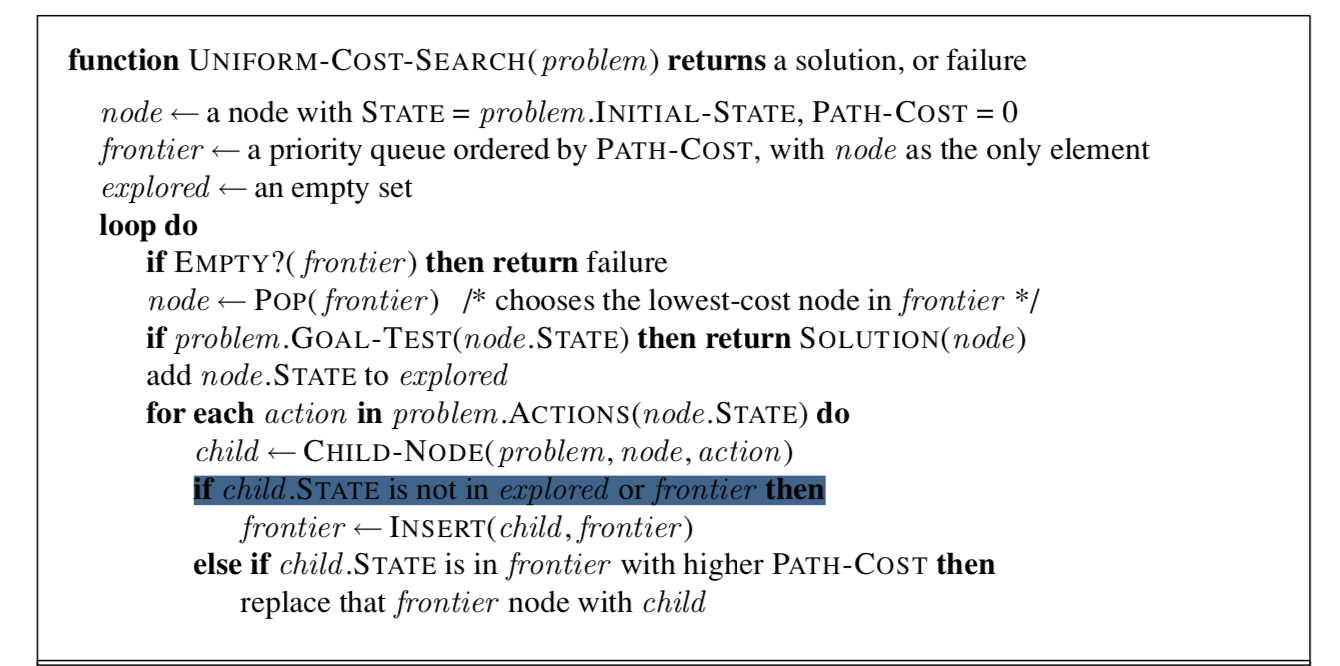
Uniform Cost Search is an algorithm used to move around a directed weighted search space to go from a start node to one of the ending nodes with a minimum cumulative cost. This search is an uninformed search algorithm since it operates in a brute-force manner, i.e. it does not take the state of the node 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

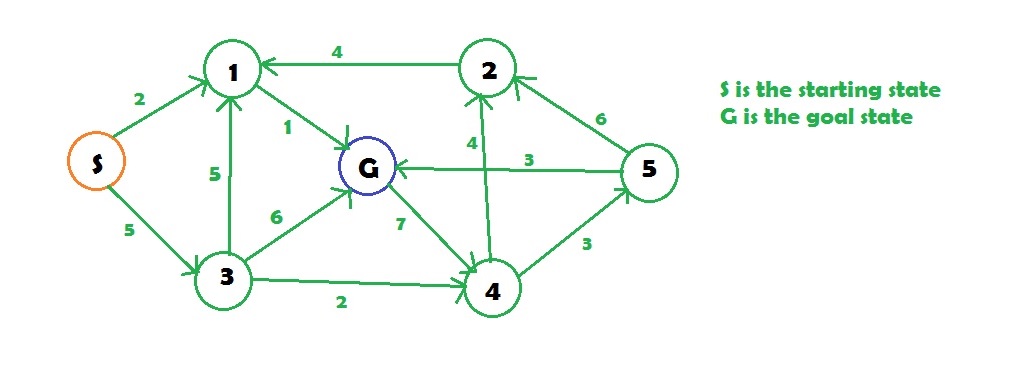
else insert all the children of removed elements into the queue with their cumulative cost as their priorities.

**Pseudocode:**



**Lab Exercises:**

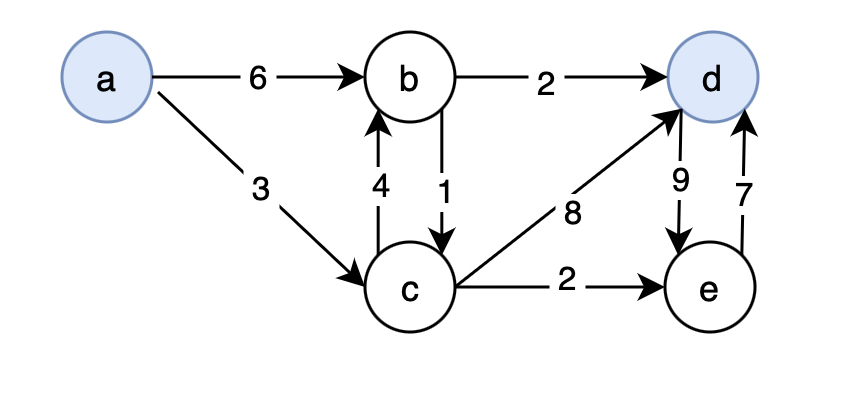
Q.1) Write a python program to find the best path between node *s* and *g* from the given graph using Uniform Cost Search algorithm.



**Output :**

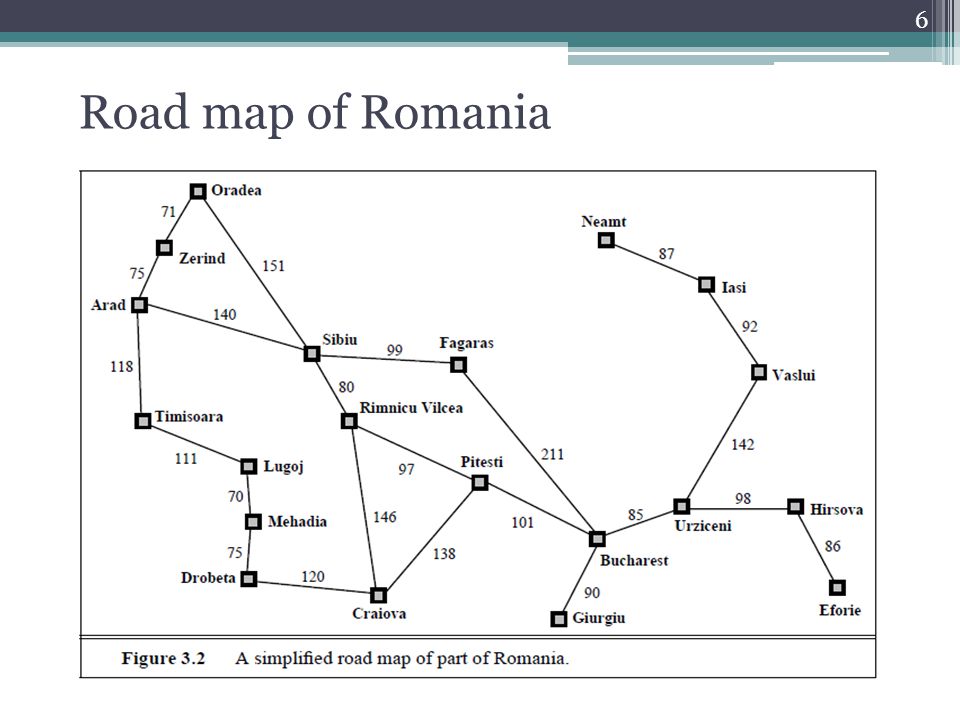
**Minimum cost from S to G is =3**

Q.2) Implement Uniform Cost Search algorithm for the following graph to find the cost of path between any two nodes using Python.



**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:**

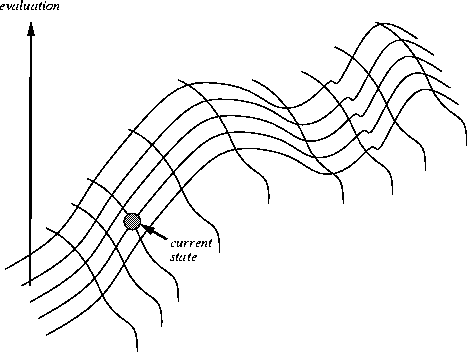
**Hill Climbing Algorithm**

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 state Inputs: ￼*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 return***Current Current* = *Next*

##### End

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

* 1. 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}.

+

* 1. Sample output: 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 and minimum 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: