DRONACHARYA
College of Engineering

# LABORATORY MANUAL

## B.Tech. Semester- VI

## ARTIFICIAL INTELLIGENCE LAB USING PYTHON
## Subject code: LC-RA-316G

| Prepared by: | Checked by: | Approved by: |
|---|---|---|
| Dr. Ritu Pahwa | Mrs. Dimple Saproo | Name : Prof. (Dr.) Isha Malhotra |
| Sign.: …………………….. | Sign.: …………………. | Sign.: …………………. |

DEPARTMENT OF ROBOTICS AND AUTOMATION
DRONACHARYA COLLEGE OF ENGINEERING
KHENTAWAS, FARRUKH NAGAR, GURUGRAM (HARYANA)

## Table of Contents

# Vision and Mission of the Institute

**Vision:**

"To impart Quality Education, to give an enviable growth to seekers of learning, to groom them as World Class Engineers and managers competent to match the expending expectations of the Corporate World has been ever enlarging vision extending to new horizons of Dronacharya College of Engineering"

**Mission:**

**M1:** To prepare students for full and ethical participation in a diverse society and encourage lifelong learning by following the principle of 'Shiksha evam Sahayata' i.e., Education & Help.

**M2:** To impart high-quality education, knowledge and technology through rigorous academic programs, cutting-edge research, & Industry collaborations, with a focus on producing engineers& managers who are socially responsible, globally aware, & equipped to address complex challenges.

**M3:** Educate students in the best practices of the field as well as integrate the latest research into the academics.

**M4:** Provide quality learning experiences through effective classroom practices, innovative teaching practices and opportunities for meaningful interactions between students and faculty.

**M5:** To devise and implement programmes of education in technology that are relevant to the changing needs of society, in terms of breadth of diversity and depth of specialization.

# Vision and Mission of the Department

**Vision:**

To be a globally recognized leader in robotics and automation education, research, and innovation, empowering students to excel in a technologically advanced world.

**Mission:**

**M1:** To provide high quality education and training in robotics and automation, equipping students with the knowledge, skills, and attitudes necessary for successful careers in the field.

**M2:** To foster a culture of innovation and entrepreneurship, encouraging student and faculty to develop and apply cutting-edge technologies in robotics and automation

**M3:** To conduct impactful research and development activities, addressing real-world challenges and advancing the field of robotics and automation

**M4:** To promote ethical practices, environmental sustainability and social responsibility in the deployment of technologies

**M5**. To collaborate with industry, academia and research organizations to create opportunities for industry-driven projects, internships, and placements ensuring the relevance of our programs and enhancing industry readiness or our graduates

## Programme Educational Objectives (PEOs)

**PEO1- ANALYTICAL SKILLS:**

Using a solid foundation in mathematical, scientific, engineering, and current computing principles, formulate, analyse, and resolve engineering issues in real-world domain.

**PEO2- TECHNICAL SKILLS:**

Apply artificial intelligence theory and concepts to analyse the requirements, realise technical specifications, and design engineering solutions.

**PEO3- SOFT SKILLS:**

Through inter-disciplinary projects and a variety of professional activities, demonstrate technical proficiency, AI competency, and foster collaborative learning and a sense of teamwork.

**PEO4- PROFESSIONAL ETHICS:**

Excel as socially responsible engineers or entrepreneurs with high moral and ethical standards, competence, and soft skills that will enable them to contribute to societal demands and achieve sustainable advancement in emerging computer technologies.

## PROGRAM OUTCOMES (POs)

**PO1: Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

**PO2: Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

**PO3: Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

**PO4: Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

**PO5: Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

**PO6: The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

**PO7: Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

**PO8: Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

**PO9: Individual and teamwork**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

**PO10: Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

**P11: Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

**P12: Life-long learning**: Recognize the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

## PROGRAM SPECIFIC OUTCOMES (PSOs)

**PSO1:** Identify the needs, analyze, design and develop simple robotic systems and programs for diverse applications in real time.

**PSO2:** Design, select and integrate appropriate automation and robotic subsystems for multi-domain engineering and integrate software applications tools.

**PSO3:** Develop impactful engineering solutions by using research-based knowledge and research methods in the fields of advanced robotics and other relevant fields.

**PSO4:** Evaluate existing engineering elements and processes, identifying areas for improvement. Propose innovative robotic and automation solutions to enhance the performance and efficiency of conventional systems.

**PSO5:** Identify suitable sensing, interfacing, control, actuation, and communication technologies to integrate various subsystems. Develop robots capable of analyzing data and implementing automated solutions through seamless connectivity between different components

## University Syllabus

Lab 1: Implementation of toy problems
Lab 2: Developing agent programs for real world problems
Lab 3: Implementation of constraint satisfaction problems
Lab 4: Implementation and Analysis of DFS and BFS for an application
Lab 5: Developing Best first search and A* Algorithm for real world problems
Lab 6: Implementation of minimax algorithm for an application
Lab7: Implementation of unification and resolution for real world problems.
Lab 8: Implementation of knowledge representation schemes - use cases
Lab 9: Implementation of uncertain methods for an application
Lab 10: Implementation of block world problem
Lab 11: Implementation of learning algorithms for an application
Lab 12: Development of ensemble model for an application
Lab 13: Expert System case study
Lab 14: Implementation of NLP programs
Lab 15: Applying deep learning methods to solve an application.

## Course Outcomes (COs)

Upon successful completion of the course, the students will be able to:

CO1: Apply various AI search algorithms (uninformed, informed, heuristic, constraint satisfaction,).
CO2: Understand the fundamentals of knowledge representation, inference.
CO3: Understand the fundamentals of theorem proving using AI tools.
CO4: Demonstrate working knowledge of reasoning in the presence of incomplete and/or uncertain information.
CO5: Apply AI techniques and technologies to solve real world business problems.

## CO-PO Mapping

| CO | PO1 | PO2 | PO3 | PO4 | PO5 | PO6 | PO7 | PO8 | PO9 | PO10 | PO11 | PO12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CO1 | 3 | 2 | 1 | 1 | 2 |  |  |  | 3 | 2 |  |  |
| CO2 |  | 1 |  | 2 |  | 3 | 1 |  | 2 |  |  | 3 |
| CO3 | 3 | 3 |  |  | 3 |  | 2 | 3 |  | 2 |  |  |
| CO4 | 1 | 1 | 2 | 2 |  |  |  |  | 2 |  | 2 |  |
| CO5 | 2 | 3 | 3 | 3 | 3 | 2 | 3 | 2 |  | 2 | 2 | 3 |
| CO | 1.8 | 2 | 1.2 | 1.6 | 1.6 | 1 | 1.2 | 1 | 1.4 | 1.2 | 0.8 | 1.2 |

## CO-PSO Mapping

| CO | PSO1 | PSO2 | PSO3 | PSO4 |
|---|---|---|---|---|
| CO1 | 3 |  | 1 |  |
| CO2 | 1 | 2 |  | 2 |
| CO3 | 3 |  | 1 |  |
| CO4 | 2 | 2 | 1 | 1 |
| CO5 | 1 | 3 | 3 | 3 |
| CO | 2 | 1.4 | 1.2 | 1.2 |

*3-HIGH
*2-MEDIUM
*1-LOW

## Course Overview

Artificial Intelligence Lab Manual for is designed to meet the course and program requirements of university B.Tech III year students of CSE (AI&ML). The concept of the lab work is to give brief practical experience for basic lab skills to students. It provides the space and scope for self-study so that students can come up with new and creative ideas. The Lab manual is written on the basis of "teach yourself pattern" and expected that students who come with proper preparation should be able to perform the experiments without any difficulty.

The pre-requisite is having a basic working knowledge of Python. Python is a general purpose, high-level programming language; other high-level languages you might have heard of C++, PHP, Java and Python. Virtually all modern programming languages make us of an Integrated Development Environment (IDE), which allows the creation, editing, testing, and saving of programs and modules. Python uses both processes, but because of the way programmers interact with it, it is usually considered an interpreted language. Practical aspects are the key to understanding and conceptual visualization of Theoretical aspects covered in the books. Also, this course is designed to review the concepts of Data Structure, studied in previous semester and implement the various algorithms related to different data structures.

Students are expected to come thoroughly prepared for the lab. General disciplines, safety guidelines and report writing are also discussed. We hope that lab manual would be useful to students of CSE, IT, ECE and BSc branches and author requests the readers to kindly forward their suggestions / constructive criticism for further improvement of the work book.

# List of Experiments mapped with Cos

## Minimum System requirements:

- Processors: Intel Atom® processor or Intel® Core™ i3 processor.
- Disk space: 1 GB.
- Operating systems: Windows* 7 or later, macOS, and Linux.
- Python* versions: 2.7.X, 3.6.X.,3.8.X and Python (Jupyter)

| Sr. No. | Title of the Experiment | CO Covered |
|---|---|---|
| 1. | PROGRAM 1: Introduction of various python libraries used for machine learning. | CO1, CO3 |
| 2. | PROGRAM 2: Write a Program to implement Uninformed Search Technique: Breadth First Search | CO1, CO3 |
| 3. | PROGRAM 3: Write a Program to implement Uninformed Search Technique: Depth First Search | CO1, CO3 |
| 4. | PROGRAM 4: Write a Program to implement Informed Search Technique: A* Algorithm | CO1, CO3, CO4 |
| 5. | PROGRAM 5: Write a Program to implement Informed Search Technique: AO* Algorithm | CO1, CO3 |
| 6. | PROGRAM 6: Write a Program to implement Local Search Technique: Hill Climbing Algorithm | CO1, CO3, CO4 |
| 7. | PROGRAM 7: Write a Program to implement Game Playing Algorithms: Minimax and Alpha Beta Pruning | CO1, CO3, CO4 |
| 8. | PROGRAM 8: Chatbot in Python | CO2, CO4, CO5 |
| 9. | PROGRAM 9: Program to Implement N-Queens Problem using Python | CO2, CO4, CO5 |
| 10. | PROGRAM 10: Program to Implement Missionaries-Cannibals Problems using Python | CO2, CO4, CO5 |

## DOs and DON'Ts

### DOs

1. Login-on with your username and password.
2. Log off the Computer every time when you leave the Lab.
3. Arrange your chair properly when you are leaving the lab.
4. Put your bags in the designated area.
5. Ask permission to print.

### DON'Ts

1. Do not share your username and password.
2. Do not remove or disconnect cables or hardware parts.
3. Do not personalize the computer setting.
4. Do not run programs that continue to execute after you log off.
5. Do not download or install any programs, games or music on computer in Lab.
6. Personal Internet use chat room for Instant Messaging (IM) and Sites is strictly prohibited.
7. No Internet gaming activities allowed.
8. Tea, Coffee, Water & Eatables are not allowed in the Computer Lab.

# General Safety Precautions

## Precautions (In case of Injury or Electric Shock)

1. To break the victim with live electric source, use an insulator such as fire wood or plastic to break the contact. Do not touch the victim with bare hands to avoid the risk of electrifying yourself.
2. Unplug the risk of faulty equipment. If main circuit breaker is accessible, turn the circuit off.
3. If the victim is unconscious, start resuscitation immediately, use your hands to press the chest in and out to continue breathing function. Use mouth-to-mouth resuscitation if necessary.
4. Immediately call medical emergency and security. Remember! Time is critical; be best.

## Precautions (In case of Fire)

1. Turn the equipment off. If power switch is not immediately accessible, take plug off.
2. If fire continues, try to curb the fire, if possible, by using the fire extinguisher or by covering it with a heavy cloth if possible, isolate the burning equipment from the other surrounding equipment.
3. Sound the fire alarm by activating the nearest alarm switch located in the hallway.
4. Call security and emergency department immediately:

**Emergency: Reception**

**Security: Main Gate**

# Guidelines to students for report preparation

All students are required to maintain a record of the experiments conducted by them. Guidelines for its preparation are as follows: -

1) All files must contain a title page followed by an index page. ***The files will not be signed by the faculty without an entry in the index page.***
2) Student's Name, roll number and date of conduction of experiment must be written on all pages.
3) For each experiment, the record must contain the following
   (i) Aim/Objective of the experiment
   (ii) Pre-experiment work (as given by the faculty)
   (iii) Lab assignment questions and their solutions
   (iv) Test Cases (if applicable to the course)
   (v) Results/ output

**Note:**
1. Students must bring their lab record along with them whenever they come for the lab.
2. Students must ensure that their lab record is regularly evaluated.

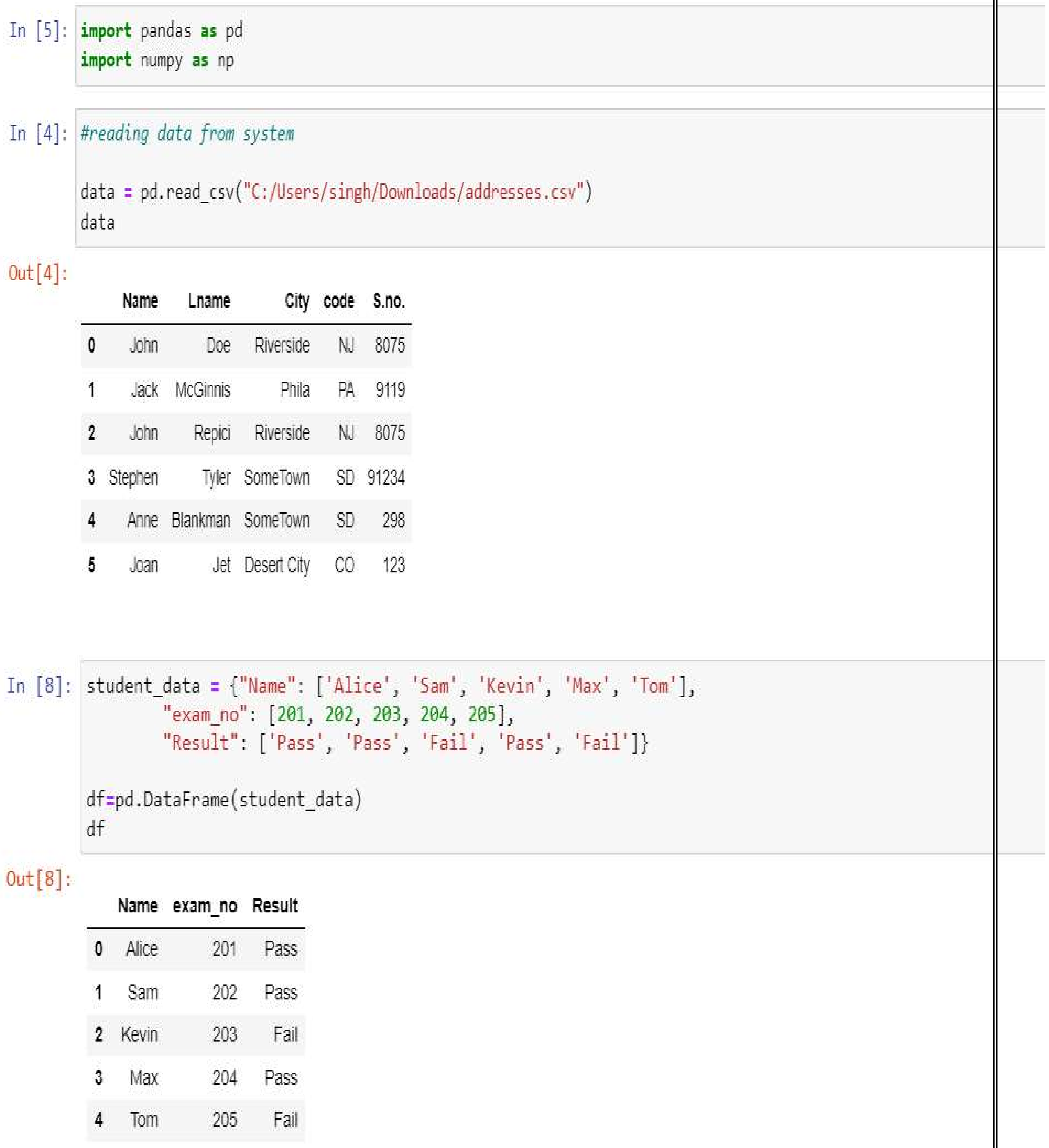
# Lab Assessment Criteria

An estimated 10 lab classes are conducted in a semester for each lab course. These lab classes are assessed continuously. Each lab experiment is evaluated based on 5 assessment criteria as shown in following table. Assessed performance in each experiment is used to compute CO attainment as well as internal marks in the lab course.

| **Grading Criteria** | **Exemplary (4)** | **Competent (3)** | **Needs Improvement (2)** | **Poor (1)** |
|---|---|---|---|---|
| **AC1:** **Pre-Lab written work (this may be assessed through viva)** | Complete procedure with underlined concept is properly written | Underlined concept is written but procedure is incomplete | Not able to write concept and procedure | Underlined concept is not clearly understood |
| **AC2:** **Program Writing/ Modeling** | Unable to understand the reason for errors/ bugs even after they are explicitly pointed out | Assigned problem is properly analyzed, correct solution designed, appropriate language constructs/ tools are applied | Assigned problem is properly analyzed & correct solution designed | Assigned problem is properly analyzed |
| **AC3:** **Identification & Removal of errors/ bugs** | Able to identify errors/ bugs and remove them | Able to identify errors/ bugs and remove them with little bit of guidance | Is dependent totally on someone for identification of errors/ bugs and their removal | Unable to understand the reason for errors/ bugs even after they are explicitly pointed out |
| **AC4:** **Execution & Demonstration** | All variants of input /output are tested, Solution is well demonstrated and implemented concept is clearly explained | All variants of input /output are not tested, However, solution is well demonstrated and implemented concept is clearly explained | Only few variants of input /output are tested, Solution is well demonstrated but implemented concept is not clearly explained | Solution is not well demonstrated and implemented concept is not clearly explained |
| **AC5:** **Lab Record Assessment** | All assigned problems are well recorded with objective, design constructs and solution along with Performance analysis using all variants of input and output | More than 70 % of the assigned problems are well recorded with objective, design contracts and solution along with Performance analysis is done with all variants of input and output | Less than 70 % of the assigned problems are well recorded with objective, design contracts and solution along with Performance analysis is done with all variants of input and output | |

# LAB EXPERIMENTS

**# PROGRAM 1: Introduction of various python libraries used for machine learning.**



In [5]:
```
import pandas as pd
import numpy as np
```

In [4]:
```
#reading data from system

data = pd.read_csv("C:/Users/singh/Downloads/addresses.csv")
data
```

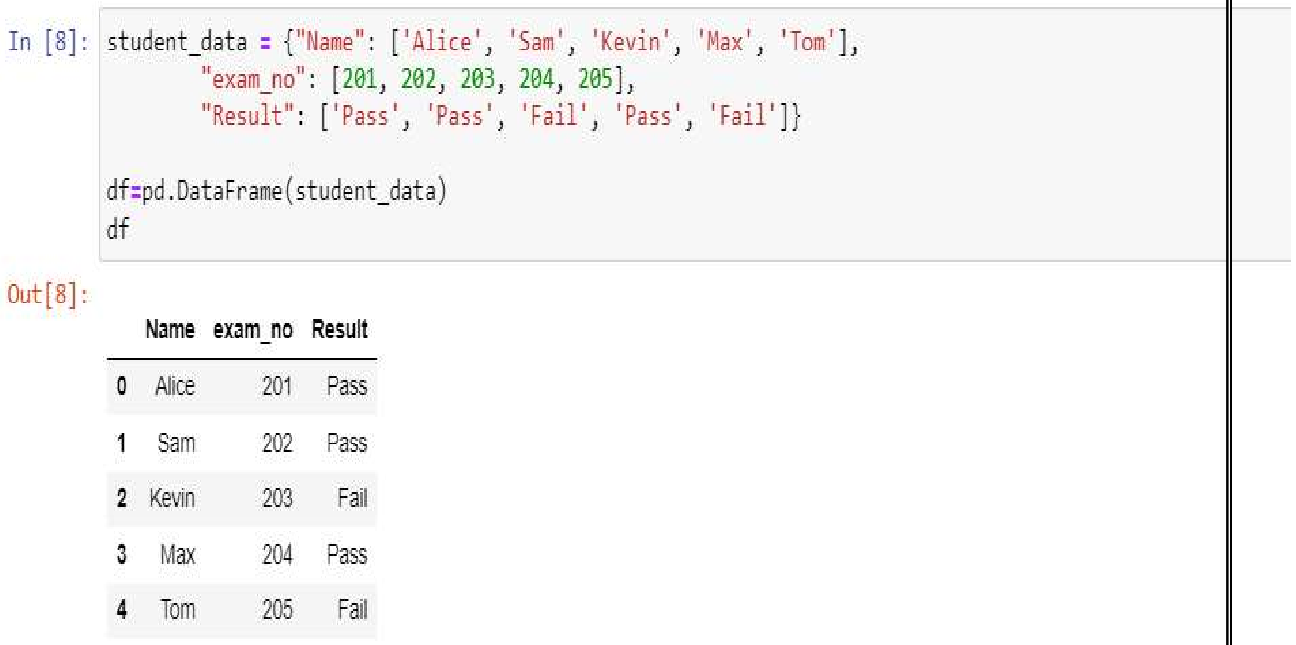
Out[4]:

| | Name | Lname | City | code | S.no. |
|---|---|---|---|---|---|
| 0 | John | Doe | Riverside | NJ | 8075 |
| 1 | Jack | McGinnis | Phila | PA | 9119 |
| 2 | John | Repici | Riverside | NJ | 8075 |
| 3 | Stephen | Tyler | SomeTown | SD | 91234 |
| 4 | Anne | Blankman | SomeTown | SD | 298 |
| 5 | Joan | Jet | Desert City | CO | 123 |

In [8]:
```
student_data = {"Name": ['Alice', 'Sam', 'Kevin', 'Max', 'Tom'],
        "exam_no": [201, 202, 203, 204, 205],
        "Result": ['Pass', 'Pass', 'Fail', 'Pass', 'Fail']}

df=pd.DataFrame(student_data)
df
```

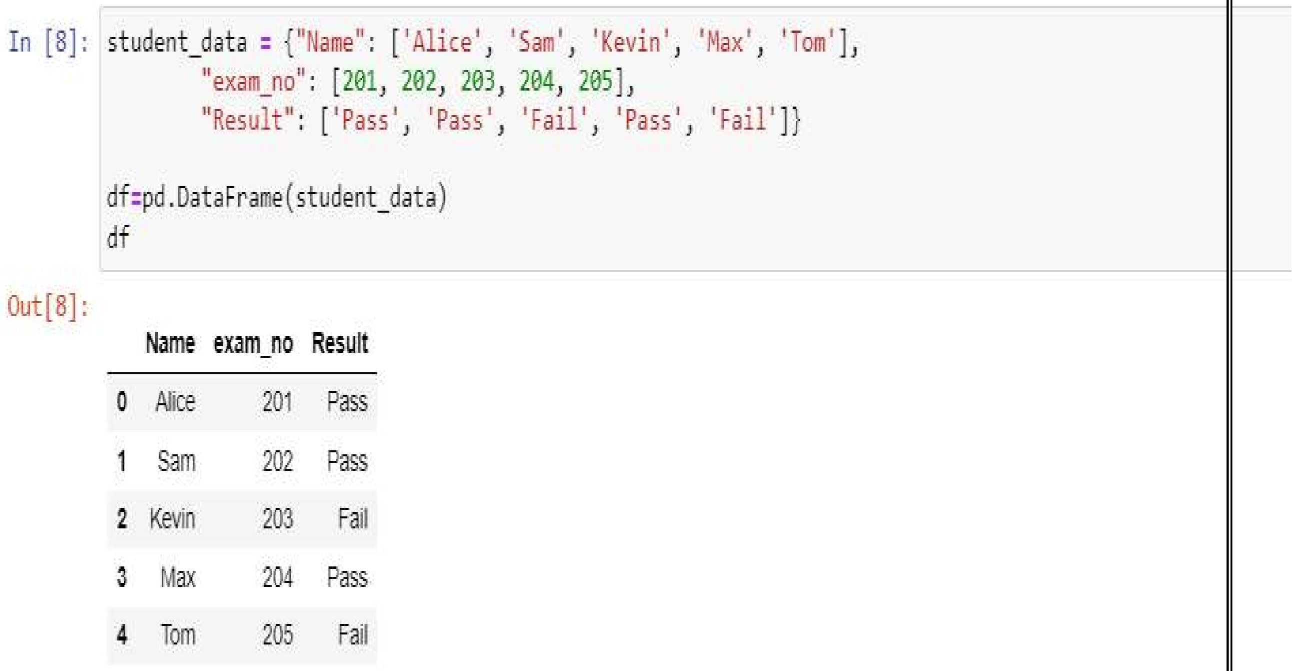
Out[8]:

| | Name | exam_no | Result |
|---|---|---|---|
| 0 | Alice | 201 | Pass |
| 1 | Sam | 202 | Pass |
| 2 | Kevin | 203 | Fail |
| 3 | Max | 204 | Pass |
| 4 | Tom | 205 | Fail |



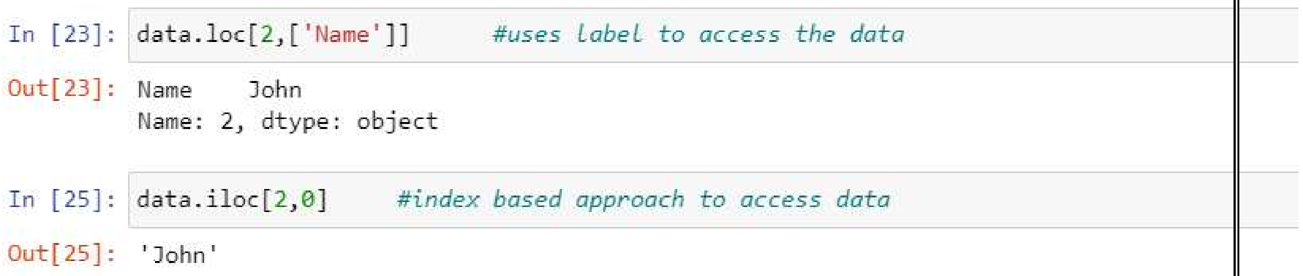
```
In [8]: student_data = {"Name": ['Alice', 'Sam', 'Kevin', 'Max', 'Tom'],
                "exam_no": [201, 202, 203, 204, 205],
                "Result": ['Pass', 'Pass', 'Fail', 'Pass', 'Fail']}

        df=pd.DataFrame(student_data)
        df
```

```
Out[8]:
     Name  exam_no  Result
  0  Alice     201    Pass
  1  Sam       202    Pass
  2  Kevin     203    Fail
  3  Max       204    Pass
  4  Tom       205    Fail
```



```
In [8]: student_data = {"Name": ['Alice', 'Sam', 'Kevin', 'Max', 'Tom'],
                "exam_no": [201, 202, 203, 204, 205],
                "Result": ['Pass', 'Pass', 'Fail', 'Pass', 'Fail']}

        df=pd.DataFrame(student_data)
        df
```

```
Out[8]:
     Name  exam_no  Result
  0  Alice     201    Pass
  1  Sam       202    Pass
  2  Kevin     203    Fail
  3  Max       204    Pass
  4  Tom       205    Fail
```



```
In [23]: data.loc[2,['Name']]      #uses label to access the data
Out[23]: Name     John
         Name: 2, dtype: object

In [25]: data.iloc[2,0]     #index based approach to access data
Out[25]: 'John'
```

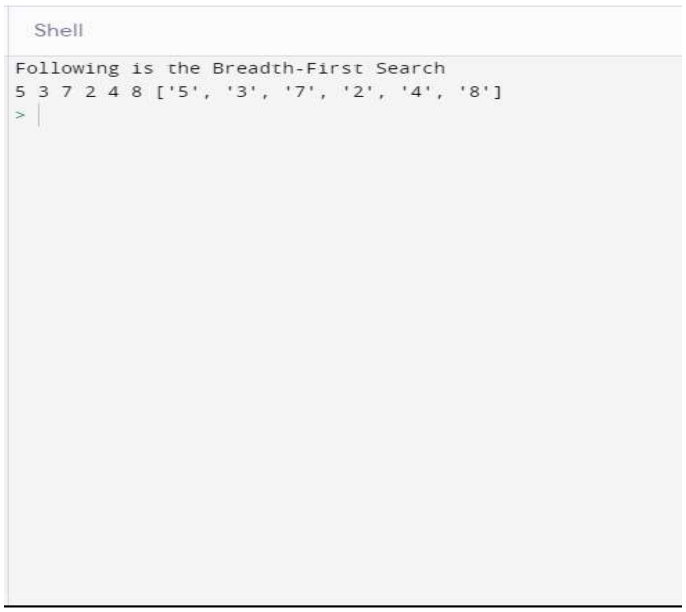
**# PROGRAM 2: Program to implement Uninformed Search Technique: Breadth First Search**

Source Code:

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
visited = [] # List for visited nodes.
queue = []     #Initialize a queue
def bfs(visited, graph, node): #function for BFS
  visited.append(node)
  queue.append(node)
  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    print (m, end = " ")
    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

# Driver Code
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')
```

Step-by-step algorithm for Breadth-First Search:

- Initialize a queue to keep track of nodes to visit.
- Enqueue the starting node into the queue.
- Initialize a set to keep track of visited nodes, and add the starting node to the set.
- While the queue is not empty, repeat steps 5-7.
- Dequeue the first node from the queue.
- For each neighbor of the dequeued node that has not been visited yet, add it to the visited set and enqueue it into the queue.
- If the goal node is found, return it. Otherwise, continue to step 4.



```
Following is the Breadth-First Search
5 3 7 2 4 8 ['5', '3', '7', '2', '4', '8']
>
```

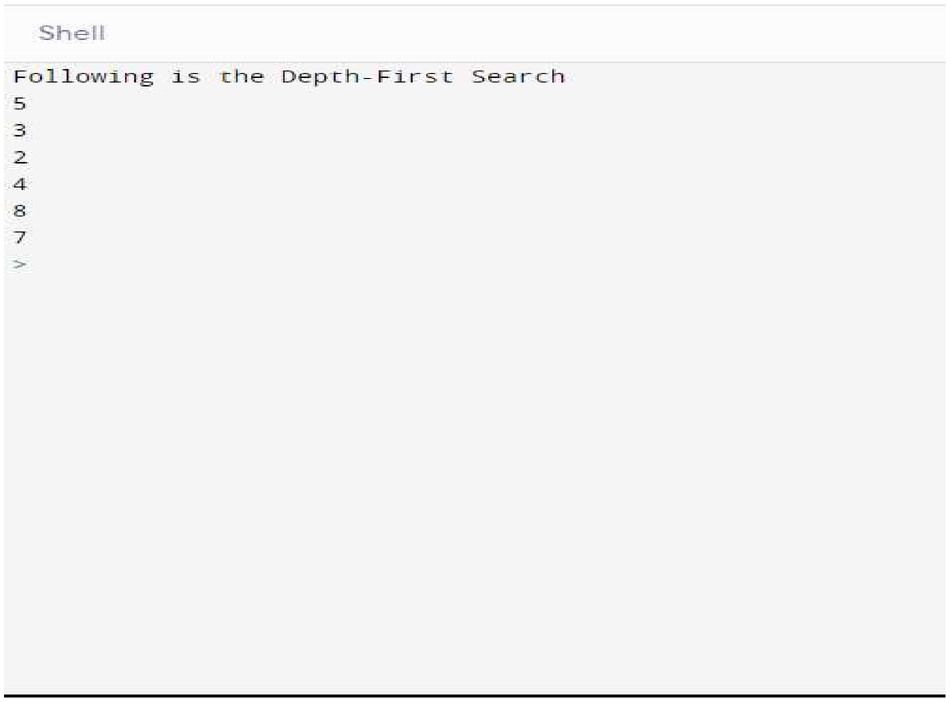
**# PROGRAM 3: Program to implement Uninformed Search Technique: Depth First Search**

Source Code:

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
visited = set() # Set to keep track of visited nodes of graph.
def dfs(visited, graph, node): #function for dfs
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)
# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

Step-by-step algorithm for Depth-First Search:

- Initialize a stack to keep track of nodes to visit.
- Push the starting node into the stack.
- Initialize a set to keep track of visited nodes, and add the starting node to the set.
- While the stack is not empty, repeat steps 5-7.
- Pop the top node from the stack.
- For each neighbor of the popped node that has not been visited yet, add it to the visited set and push it onto the stack.
- If the goal node is found, return it. Otherwise, continue to step 4.



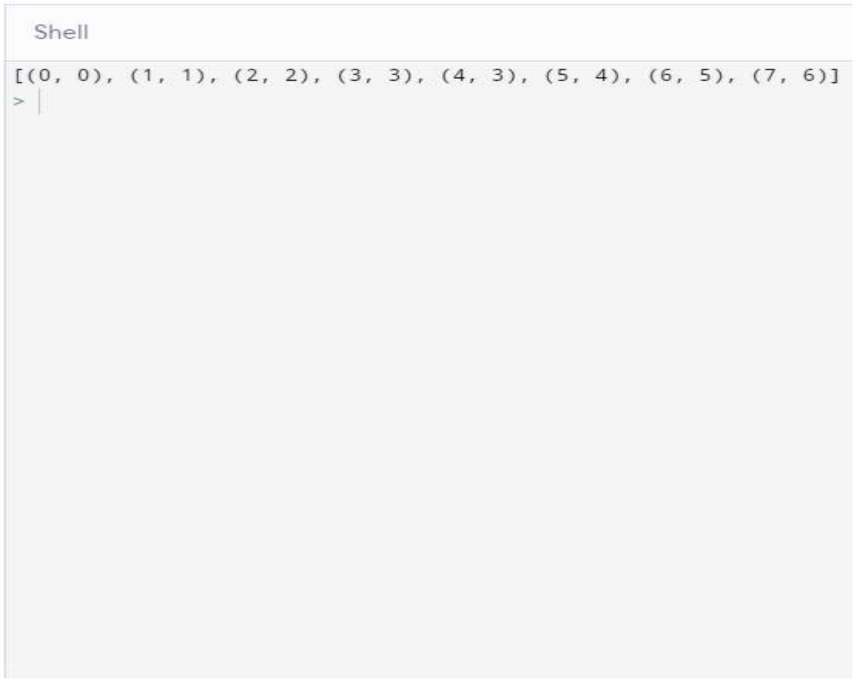
```
Following is the Depth-First Search
5
3
2
4
8
7
>
```

**# PROGRAM 4: Program to implement Informed Search Technique: A* Algorithm**

Source Code:

```
class Node():

    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position

        self.g = 0
        self.h = 0
        self.f = 0

    def __eq__(self, other):
        return self.position == other.position

def astar(maze, start, end):
    """Returns a list of tuples as a path from the given start to the given end in the given
maze"""

    # Create start and end node
    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end)
    end_node.g = end_node.h = end_node.f = 0

    # Initialize both open and closed list
    open_list = []
    closed_list = []

    # Add the start node
    open_list.append(start_node)

    # Loop until you find the end
    while len(open_list) > 0:

        # Get the current node
        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index
        # Pop current off open list, add to closed list
        open_list.pop(current_index)
        closed_list.append(current_node)

        # Found the goal
        if current_node == end_node:
            path = []
            current = current_node
            while current is not None:
                path.append(current.position)
                current = current.parent
            return path[::-1] # Return reversed path

        # Generate children
        children = []
        for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (1, 1)]: #
Adjacent squares

            # Get node position
            node_position = (current_node.position[0] + new_position[0],
current_node.position[1] + new_position[1])

            # Make sure within range
            if node_position[0] > (len(maze) - 1) or node_position[0] < 0 or
node_position[1] > (len(maze[len(maze)-1]) -1) or node_position[1] < 0:
                continue

            # Make sure walkable terrain
            if maze[node_position[0]][node_position[1]] != 0:
                continue

            # Create new node
            new_node = Node(current_node, node_position)

            # Append
            children.append(new_node)

        # Loop through children
        for child in children:

            # Child is on the closed list
            for closed_child in closed_list:
                if child == closed_child:
                    continue

            # Create the f, g, and h values
            child.g = current_node.g + 1
            child.h = ((child.position[0] - end_node.position[0]) ** 2) + ((child.position[1] -
end_node.position[1]) ** 2)
            child.f = child.g + child.h

            # Child is already in the open list
            for open_node in open_list:
                if child == open_node and child.g > open_node.g:
                    continue

            # Add the child to the open list
            open_list.append(child)

def main():

    maze = [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

    start = (0, 0)
    end = (7, 6)

    path = astar(maze, start, end)
    print(path)


if __name__ == '__main__':
    main()
```

```
// A* Search Algorithm

1. Initialize the open list

2. Initialize the closed list put the starting node on the open list (you can leave its f at zero)

3. while the open list is not empty

    a) find the node with the least f on the open list, call it "q"
    b) pop q off the open list
    c) generate q's 8 successors and set their parents to q
    d) for each successor
        i) if successor is the goal, stop search
        ii) else, compute both g and h for successor successor.g = q.g + distance
        between successor and q successor.h = distance from goal to successor (This
        can be done using many ways, we will discuss three heuristics- Manhattan,
        Diagonal and Euclidean Heuristics) successor.f = successor.g + successor.h
        iii)if a node with the same position as successor is in the OPEN list which has a
        lower f than successor, skip this successor
        iv) if a node with the same position as successor is in the CLOSED list which
        has a lower f than successor, skip this successor otherwise, add the node to the
        open list end (for loop)
   e) push q on the closed list end (while loop)
```



```
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 3), (5, 4), (6, 5), (7, 6)]
>
```

**# PROGRAM 5: Program to implement Informed Search Technique: AO* Algorithm**

Source Code:

```
def Cost(H, condition, weight = 1):
        cost = {}
        if 'AND' in condition:
                AND_nodes = condition['AND']
                Path_A = ' AND '.join(AND_nodes)
                PathA = sum(H[node]+weight for node in AND_nodes)
                cost[Path_A] = PathA

        if 'OR' in condition:
                OR_nodes = condition['OR']
                Path_B =' OR '.join(OR_nodes)
                PathB = min(H[node]+weight for node in OR_nodes)
                cost[Path_B] = PathB
        return cost

# Update the cost
def update_cost(H, Conditions, weight=1):
        Main_nodes = list(Conditions.keys())
        Main_nodes.reverse()
        least_cost= {}
        for key in Main_nodes:
                condition = Conditions[key]
                print(key,':', Conditions[key],'>>>', Cost(H, condition, weight))
                c = Cost(H, condition, weight)
                H[key] = min(c.values())
                least_cost[key] = Cost(H, condition, weight)
        return least_cost

# Print the shortest path
def shortest_path(Start,Updated_cost, H):
        Path = Start
        if Start in Updated_cost.keys():
                Min_cost = min(Updated_cost[Start].values())
                key = list(Updated_cost[Start].keys())
                values = list(Updated_cost[Start].values())
                Index = values.index(Min_cost)

                # FIND MINIMIMUM PATH KEY
                Next = key[Index].split()
                # ADD TO PATH FOR OR PATH
                if len(Next) == 1:

                    Start =Next[0]
                    Path += '<--' +shortest_path(Start, Updated_cost, H)
                # ADD TO PATH FOR AND PATH
                else:
                    Path +='<--('+key[Index]+') '

                    Start = Next[0]
                    Path += '[' +shortest_path(Start, Updated_cost, H) + ' + '

                    Start = Next[-1]
                    Path += shortest_path(Start, Updated_cost, H) + ']'

        return Path



H = {'A': -1, 'B': 5, 'C': 2, 'D': 4, 'E': 7, 'F': 9, 'G': 3, 'H': 0, 'I':0, 'J':0}

Conditions = {
'A': {'OR': ['B'], 'AND': ['C', 'D']},
'B': {'OR': ['E', 'F']},
'C': {'OR': ['G'], 'AND': ['H', 'I']},
'D': {'OR': ['J']}
}
# weight
weight = 1
# Updated cost
print('Updated Cost :')
Updated_cost = update_cost(H, Conditions, weight=1)
print('*'*75)
print('Shortest Path :\n',shortest_path('A', Updated_cost,H))
```

Working of AO algorithm:
The AO* algorithm works on the formula given below :
**f(n) = g(n) + h(n)**
where,

- g(n): The actual cost of traversal from initial state to the current state.
- h(n): The estimated cost of traversal from the current state to the goal state.
- f(n): The actual cost of traversal from the initial state to the goal state

Step-1: Create an initial graph with a single node (start node).

Step-2: Transverse the graph following the current path, accumulating node that has not yet been expanded or solved.

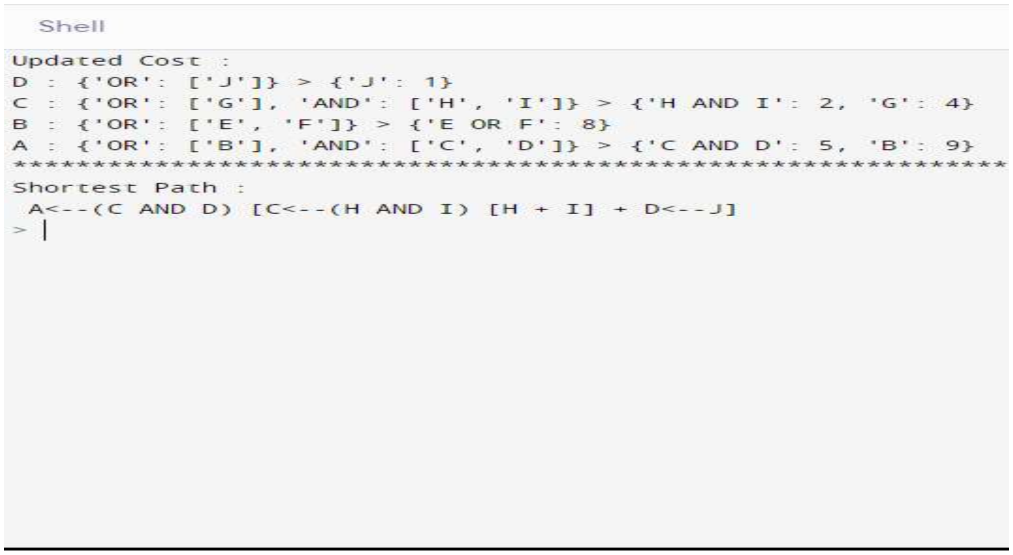
Step-3: Select any of these nodes and explore it. If it has no successors then call this value- FUTILITY else calculate f'(n) for each of the successors.

Step-4: If f'(n)=0, then mark the node as SOLVED.

Step-5: Change the value of f'(n) for the newly created node to reflect its successors by backpropagation.

Step-6: Whenever possible use the most promising routes, If a node is marked as SOLVED then mark the parent node as SOLVED.

Step-7: If the starting node is SOLVED or value is greater than FUTILITY then stop else repeat from Step-2.



```
Updated Cost :
D : {'OR': ['J']} > {'J': 1}
C : {'OR': ['G'], 'AND': ['H', 'I']} > {'H AND I': 2, 'G': 4}
B : {'OR': ['E', 'F']} > {'E OR F': 8}
A : {'OR': ['B'], 'AND': ['C', 'D']} > {'C AND D': 5, 'B': 9}
*****************************************************************
Shortest Path :
 A<--(C AND D) [C<--(H AND I) [H + I] + D<--J]
> |
```

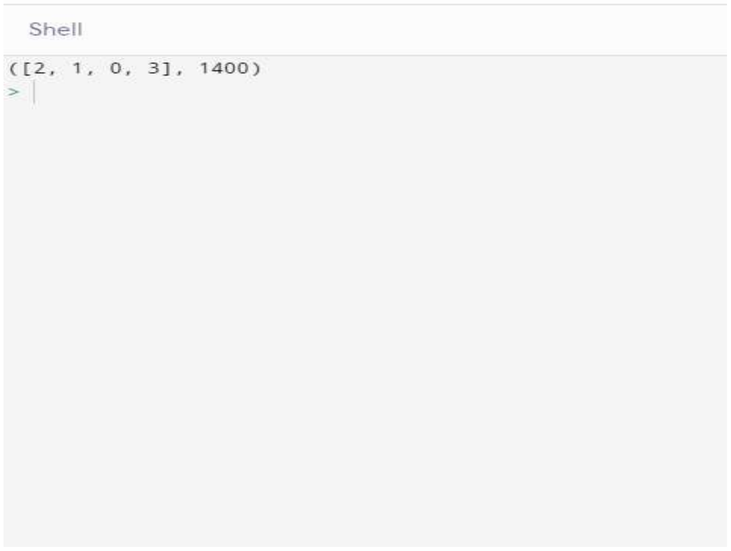
**# PROGRAM 6: Program to implement Local Search Technique: Hill Climbing Algorithm**

Source Code:

```
import random

def randomSolution(tsp):
    cities = list(range(len(tsp)))
    solution = []

    for i in range(len(tsp)):
        randomCity = cities[random.randint(0, len(cities) - 1)]
        solution.append(randomCity)
        cities.remove(randomCity)

    return solution

def routeLength(tsp, solution):
    routeLength = 0
    for i in range(len(solution)):
        routeLength += tsp[solution[i - 1]][solution[i]]
    return routeLength

def getNeighbours(solution):
    neighbours = []
    for i in range(len(solution)):
        for j in range(i + 1, len(solution)):
            neighbour = solution.copy()
            neighbour[i] = solution[j]
            neighbour[j] = solution[i]
            neighbours.append(neighbour)
    return neighbours

def getBestNeighbour(tsp, neighbours):
    bestRouteLength = routeLength(tsp, neighbours[0])
    bestNeighbour = neighbours[0]
    for neighbour in neighbours:
        currentRouteLength = routeLength(tsp, neighbour)
        if currentRouteLength < bestRouteLength:
            bestRouteLength = currentRouteLength
            bestNeighbour = neighbour
    return bestNeighbour, bestRouteLength
```

```
def hillClimbing(tsp):
    currentSolution = randomSolution(tsp)
    currentRouteLength = routeLength(tsp, currentSolution)
    neighbours = getNeighbours(currentSolution)
    bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(tsp, neighbours)

    while bestNeighbourRouteLength < currentRouteLength:
        currentSolution = bestNeighbour
        currentRouteLength = bestNeighbourRouteLength
        neighbours = getNeighbours(currentSolution)
        bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(tsp, neighbours)

    return currentSolution, currentRouteLength

def main():
    tsp = [
        [0, 400, 500, 300],
        [400, 0, 300, 500],
        [500, 300, 0, 400],
        [300, 500, 400, 0]
    ]

    print(hillClimbing(tsp))

if __name__ == "__main__":
    main()
```

Algorithm for Simple Hill Climbing:
Step 1: Evaluate the initial state, if it is goal state then return success and Stop.
Step 2: Loop Until a solution is found or there is no new operator left to apply.
Step 3: Select and apply an operator to the current state.
Step 4: Check new state:
If it is goal state, then return success and quit.
Else if it is better than the current state then assign new state as a current state.
Else if not better than the current state, then return to step2.
Step 5: Exit.



```
([2, 1, 0, 3], 1400)
>
```

**# PROGRAM 7: Program to implement Game Playing Algorithms: Minimax and Alpha Beta Pruning**

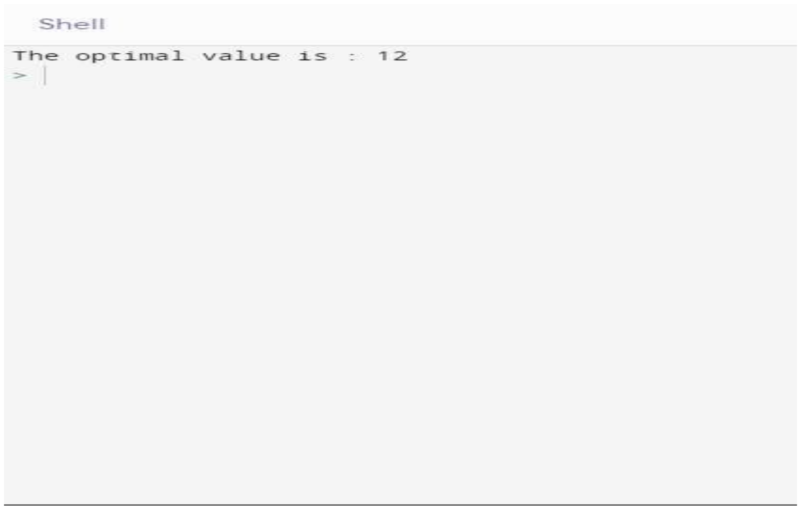
Source Code

a) MiniMAx Algorithm

```
import math

def minimax (curDepth, nodeIndex,
             maxTurn, scores,
             targetDepth):

    # base case : targetDepth reached
    if (curDepth == targetDepth):
        return scores[nodeIndex]

    if (maxTurn):
        return max(minimax(curDepth + 1, nodeIndex * 2,
                    False, scores, targetDepth),
                minimax(curDepth + 1, nodeIndex * 2 + 1,
                    False, scores, targetDepth))

    else:
        return min(minimax(curDepth + 1, nodeIndex * 2,
                    True, scores, targetDepth),
                minimax(curDepth + 1, nodeIndex * 2 + 1,
                    True, scores, targetDepth))

# Driver code
scores = [3, 5, 2, 9, 12, 5, 23, 23]

treeDepth = math.log(len(scores), 2)

print("The optimal value is : ", end = "")
print(minimax(0, 0, True, scores, treeDepth))
```

Algorithm:

1. Construct the complete game tree
2. Evaluate scores for leaves using the evaluation function
3. Back-up scores from leaves to root, considering the player type:
4. For max player, select the child with the maximum score
5. For min player, select the child with the minimum score
6. At the root node, choose the node with max value and perform the corresponding move



```
The optimal value is : 12
>
```

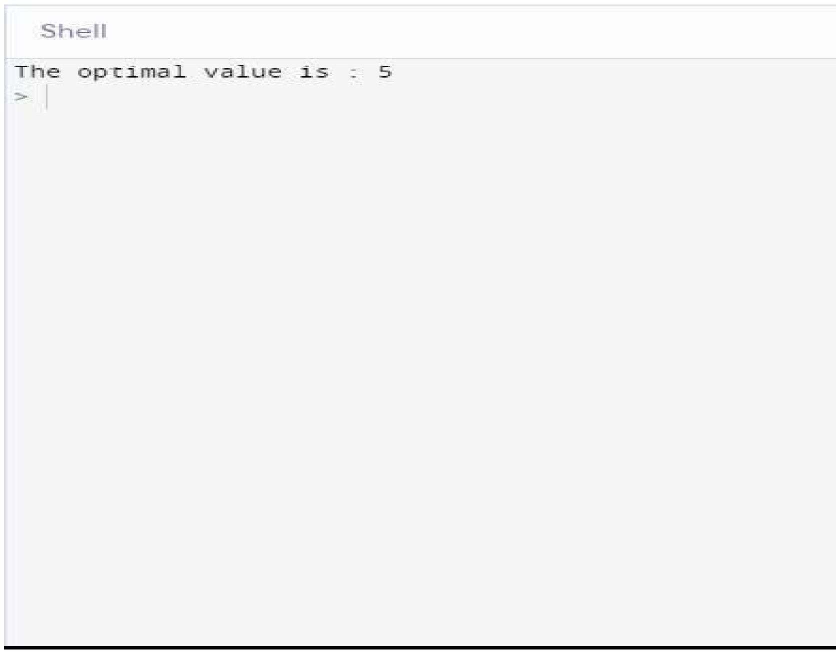
Source Code

b) Alpha Beta Pruning Algorithm

```
# Initial values of Alpha and Beta
MAX, MIN = 1000, -1000

# Returns optimal value for current player
#(Initially called for root and maximizer)
def minimax(depth, nodeIndex, maximizingPlayer,
            values, alpha, beta):

    # Terminating condition. i.e
    # leaf node is reached
    if depth == 3:
        return values[nodeIndex]

    if maximizingPlayer:

        best = MIN

        # Recur for left and right children
        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex * 2 + i,
                          False, values, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)

            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best

    else:
        best = MAX

        # Recur for left and
        # right children
        for i in range(0, 2):
            val = minimax(depth + 1, nodeIndex * 2 + i,
                                    True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)

            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best

# Driver Code
if __name__ == "__main__":

    values = [3, 5, 6, 9, 1, 2, 0, -1]
    print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

Algorithm:

1. Define the initial values for alpha and beta as negative and positive infinity, respectively.
2. Begin the recursive search through the tree, starting at the root node.
3. If the current node is a leaf node, evaluate its value and return it.
4. If the current node is a maximizing node, then set alpha to the maximum of alpha and the value returned from its child node.
5. If alpha is greater than or equal to beta, then prune the remaining child nodes and return alpha.
6. If the current node is a minimizing node, then set beta to the minimum of beta and the value returned from its child node.
7. If beta is less than or equal to alpha, then prune the remaining child nodes and return beta.
8. Recurse to the next level of the tree, continuing with steps 3 to 7 until the entire tree has been searched.
9. Return the final value of alpha or beta depending on whether the root node is a maximizing or minimizing node.



```
The optimal value is : 5
>
```

**# PROGRAM 8: Chatbot in Python**

```
$ pip install chatterbot
$ pip install chatterbot_corpus
$ pip install git+git://github.com/gunthercox/ChatterBot.git@master
$ pip install --upgrade chatterbot_corpus
$ pip install --upgrade chatterbot
# importing the required modules
from chatterbot import ChatBot
from chatterbot.trainers import ListTrainer
# creating a chatbot
myBot = ChatBot(
    name = 'Siya',
    read_only = True,
    logic_adapters = [
        'chatterbot.logic.MathematicalEvaluation',
        'chatterbot.logic.BestMatch'
        ]
        )
# training the chatbot
small_convo = [
    'Hi there!',
    'Hi',
    'How do you do?',
    'How are you?',
    'I\'m cool.',
    'Always cool.',
    'I\'m Okay',
    'Glad to hear that.',
    'I\'m fine',
    'I feel awesome',
    'Excellent, glad to hear that.',
    'Not so good',
    'Sorry to hear that.',
    'What\'s your name?',
    ' I\'m Sakura. Ask me a math question, please.'
    ]
math_convo_1 = [
    'Pythagorean theorem',
    'a squared plus b squared equals c squared.'
    ]

math_convo_2 = [
    'Law of Cosines',
    'c**2 = a**2 + b**2 - 2*a*b*cos(gamma)'
    ]
```

**File: my_chatbot.py**

```
# using the ListTrainer class
list_trainee = ListTrainer(myBot)
for i in (small_convo, math_convo_1, math_convo_2):
    list_trainee.train(i)
```

OUTPUT:

```
# starting a conversation
>>> print(myBot.get_response("Hi, there!"))
Hi
>>> print(myBot.get_response("What's your name?"))
I'm SIYA. Ask me a math question, please.
>>> print(myBot.get_response("Do you know Pythagorean theorem"))
a squared plus b squared equals c squared.
>>> print(myBot.get_response("Tell me the formula of law of cosines"))
c**2 = a**2 + b**2 - 2*a*b*cos(gamma)
```

**# PROGRAM 9: Program to Implement N-Queens Problem using Python**

```
# Python program to solve N Queen
# Problem using backtracking
global N
N = 4
def printSolution(board):
for i in range(N):
for j in range(N):
print board[i][j],
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]
# 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 colum 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()
```

**Output:**

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

## # PROGRAM 10: Program to Implement Missionaries-Cannibals Problems using Python

```
'' mclib.py '''
class MCState:
### MC is missionaries and cannibals
def __init__(self, state_vars, num_moves=0, parent=None):
self.state_vars = state_vars
self.num_moves = num_moves
self.parent = parent
### decorator
@classmethod
def root(cls):
return cls((3,3,1))
def get_possible_moves(self):
''' return all possible moves in the game as tuples
possible moves:
1 or 2 mis
1 or 2 cannibals
1 mis, 1 can
'''
moves = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]
return moves
def is_legal(self):
missionaries = self.state_vars[0]
cannibals = self.state_vars[1]
## could have done tuple unpacking too:
## missionaries, cannibals, boat = self.state_vars
if missionaries < 0 or missionaries > 3:
return False
elif cannibals < 0 or cannibals > 3:
return False
return True
## alternate
# if 0 <= missionaries <= 3 and 0 <= cannibals <= 3
# return True
###
def is_solution(self):
if self.state_vars == (0,0,0):
return True
return False
def is_failure(self):
missionaries = self.state_vars[0]
cannibals = self.state_vars[1]
boat = self.state_vars[2]
## could have done tuple unpacking too:
## missionaries, cannibals, boat = self.state_vars
### missionaries on right side AND more cannibals than missionaries
if missionaries > 0 and missionaries < cannibals:
return True
## to make this easier to understand, I will create temporary variables
## but we could just substitute the math and skip the variables
missionaries_on_left = 3 - missionaries
cannibals_on_left = 3 - cannibals
if missionaries_on_left > 0 and missionaries_on_left < cannibals_on_left:
return True
## if you replace the math in, you get:
#if 3 - missionaries > 0 and 3 - missionaires < 3 - cannaibals
# which leads to:
#if missionaries < 3 and cannibals < missionaries:
### if we make it here, we aren't in a failed state!
return False
def get_next_states(self):
## using possible move, get next states
moves = self.get_possible_moves()
all_states = list()
mis_right, can_right, raft_right = self.state_vars
## if raft is on right, subtract move from these numbers
## if raft is on left, add these move numbers to these numbers
for move in moves:
change_mis, change_can = move
if raft_right == 1: ## mis_right = 3; can_right = 3, raft_right = 1
new_state_vars = (mis_right-change_mis, can_right-change_can, 0)
else:
new_state_vars = (mis_right+change_mis, can_right+change_can, 1)
## notice the number of moves is increasing by 1
## also notice we are passing self to our child.
new_state = MCState(new_state_vars, self.num_moves+1, self)
if new_state.is_legal():
all_states.append(new_state)
return all_states
def __str__(self):
return "MCState[{}]".format(self.state_vars)
def __repr__(self):
return str(self)
def search(dfs=True):
### this is the stack/queue that we used before
from collections import deque
### create the root state
root = MCState.root()
### we use the stack/queue for keeping track of where to search next
to_search = deque()
### use a set to keep track fo where we've been
seen_states = set()
### use a list to keep track of the solutions that have been seen
solutions = list()
### start the search with the root
to_search.append(root)
### safety variable for infinite loops!
loop_count = 0
max_loop = 10000
```

```
### while the stack/queue still has items
while len(to_search) > 0:
loop_count += 1
if loop_count > max_loop:
print(len(to_search))
print("Escaping this super long loop!")
break
### get the next item
current_state = to_search.pop()
## look at the current state's children
## this uses the rule for actions and moves to create next states
## it is also removing all illegal states
next_states = current_state.get_next_states()
## next_states is a list, so iterate through it
for possible_next_state in next_states[::-1]:
## to see if we've been here before, we look at the state variables
possible_state_vars = possible_next_state.state_vars
## we use the set and the "not in" boolean comparison
if possible_state_vars not in seen_states:
if possible_next_state.is_failure():
#print("Failure!")
continue
elif possible_next_state.is_solution():
## Save it into our solutions list
solutions.append(possible_next_state)
#print("Solution!")
continue
# the state variables haven't been seen yet
# so we add the state itself into the searching stack/queue
#### IMPORTANT
## which side we append on changes how the search works
## why is this?
if dfs:
to_search.append(possible_next_state)
else:
to_search.appendleft(possible_next_state)
# now that we have "seen" the state, we add the state vars to the set.
# this means next time when we do the "not in", that will return False
# because it IS in
#seen_states.add(possible_state_vars)
seen_states.add(possible_state_vars)
## finally, we reach this line when the stack/queue is empty (len(to_searching==))
print("Found {} solutions".format(len(solutions)))
return solutions
sol_dfs = search(True)
sol_bfs = search(False)
current_state = sol_dfs[0]
while current_state:
print(current_state)
current_state = current_state.parent
```

```
print("--")
current_state = sol_dfs[1]
while current_state:
print(current_state)
current_state = current_state.parent
```

```
print("--")
current_state = sol_bfs[0]
while current_state:
print(current_state)
current_state = current_state.parent
print("--")
current_state = sol_bfs[1]
while current_state:
print(current_state)
current_state = current_state.parent
Found 2 solutions
Found 2 solutions
MCState[(0, 0, 0)]
MCState[(1, 1, 1)]
MCState[(0, 1, 0)]
MCState[(0, 3, 1)]
MCState[(0, 2, 0)]
MCState[(2, 2, 1)]
MCState[(1, 1, 0)]
MCState[(3, 1, 1)]
MCState[(3, 0, 0)]
MCState[(3, 2, 1)]
MCState[(3, 1, 0)]
MCState[(3, 3, 1)]
--
MCState[(0, 0, 0)]
MCState[(0, 2, 1)]
MCState[(0, 1, 0)]
MCState[(0, 3, 1)]
MCState[(0, 2, 0)]
MCState[(2, 2, 1)]
MCState[(1, 1, 0)]
MCState[(3, 1, 1)]
MCState[(3, 0, 0)]
MCState[(3, 2, 1)]
MCState[(3, 1, 0)]
MCState[(3, 3, 1)]
--
MCState[(0, 0, 0)]
MCState[(0, 2, 1)]
MCState[(0, 1, 0)]
MCState[(0, 3, 1)]
MCState[(0, 2, 0)]
MCState[(2, 2, 1)]
MCState[(1, 1, 0)]
MCState[(3, 1, 1)]
MCState[(3, 0, 0)]
```

```
MCState[(3, 2, 1)]
MCState[(2, 2, 0)]
MCState[(3, 3, 1)]
--
MCState[(0, 0, 0)]
MCState[(1, 1, 1)]
MCState[(0, 1, 0)]
MCState[(0, 3, 1)]
MCState[(0, 2, 0)]
MCState[(2, 2, 1)]
MCState[(1, 1, 0)]
MCState[(3, 1, 1)]
MCState[(3, 0, 0)]
MCState[(3, 2, 1)]
MCState[(2, 2, 0)]
MCState[(3, 3, 1)]
```

This lab manual has been updated by

Dr. Ritu Pahwa

(ritu.pahwa@ggnindia.dronacharya.info)

**Cross checked by**

HoD / EEE & ECE