



SARASWATI Education Society's
SARASWATI College of Engineering

Learn Live Achieve and Contribute

Kharghar, Navi Mumbai - 410 210.

NAAC Accredited

Department of Computer Engineering

Name: Shaikh Adnan Shaukat Ali

Roll No: 76

Subject: Artificial Intelligence (AI)

Class/Sem: T.E. / SEM-VI



Department of Computer Engineering

Subject: AI

Class/Sem: TE/VI

Name of the Laboratory: Artificial Intelligence Lab

Year: 2021-2022

LIST OF EXPERIMENTS

Expt. No.	Date	Name of the Experiment
1.	17-01	One case study on AI applications published in IEEE/ACM/Springer or any prominent journal
2.	31-01	To implement BFS (Breadth First Search)
3.	08-02	To implement Depth First Search (DFS) algorithm
4.	15-02	To implement uniform cost search
5.	22-02	To implement Greedy Best First Search
6.	08-03	To implement A* search algorithm
7.	15-03	To implement basic code in Prolog
8.	15-03	To find max of two number using Prolog
9	16-03	To implement family tree using recursion in Prolog
10	24-03	To implement graph coloring problem in Prolog
11	4-04	To calculate factorial of a number using Prolog

Prof. Neha Mahajan

Prof. Sujata Bhairnallykar

Subject In-charge

HOD



SARASWATI Education Society's
SARASWATI College of Engineering

Learn Live Achieve and Contribute

Kharghar, Navi Mumbai - 410 210.

NAAC Accredited

Department of Computer Engineering

Subject: AI

Class/Sem: TE/VI

Name of the Laboratory: Artificial Intelligence Lab

Year: 2021-2022

ASSIGNMENT INDEX

SR No.	Date	Name of Assignment
1	11-02	Assignment-1
2	24-03	Assignment-2

Prof. Neha Mahajan

Prof. Sujata Bhairnallykar

Subject In-charge

HOD

Experiment No. 1

Case Study on Early and Remote Detection of Possible Heartbeat Problems

Topic: Early and Remote Detection of Possible Heartbeat Problems With Convolutional Neural Networks and Multipart Interactive Training.

Journal: IEEE

Publication Date: May 27, 2019

Author: Krzysztof Wołk (kwolk@pja.edu.pl)

Introduction

In the past decades, heart attack has been a significant health issue worldwide. Records from the World Health Organization (WHO) show that the number of deaths due to heart disease was higher in the twenty-first century. Any measures that can help contain the negative effects of cardiovascular disease should be seriously considered. Krzysztof created the Convolution Neural Network Model which produces accurate assessment of heart function by analysing heart beat sound.

Challenges

1. Heart beat sound data used for training the model contained lots of background noises.
2. The dataset A had more murmur class data than normal and extra sound.
3. In both dataset A and B Artificial audio file doesn't contained sound at all.
4. Some data were shorter than 3 second.
5. CSV file wasn't contained correct name as WAV file.
6. Data set size were small to train neural network.

Proposed Solution

1. Data were down sampled with very aggressive low pass band filtering (around 300Hz), all the background noise frequency were removed.
2. Data Augmentation applied on normal and extra sound data of data set A.
3. Artificats data totally removed from both set A and B.
4. Data shorter than 4 seconds were removed.
5. Data iteratively fetched from directory instead of using CSV File.
6. Pre-trained model such as ResNet and Mobile V Net were used.

Experiments

Most of the Image processing model can be used for Sound Classification, data were pre-processed and then Mel-Spectrogram image were created for each data and each image was reduced to 224*224 pixel(to fit in a 64-size batch). Models input layer changes to accept 224*224*3 (height*width*channels) and output layer were changed and contained only 4 neurons. Optimizer used for Resnet was Cyclic Stochastic Gradient Descent and data were trained using 2-3 Epochs.

Conclusion

Model were able to achieved overall accuracy of 99.96% and made only one mistake on the official testing set. With better quality of data set and more training iteration of model can achieve greater accuracy.

Experiment No. 2

Aim: To write a program of BFS (Breadth First Search).

Requirements: Windows/MAC/Linux O.S, Compatible version of Python.

Theory:

BFS is an algorithm to traverse each node of **Graph** at least once. It goes level by level that means it will only explore nodes of next level if all the nodes of previous level are explored or it first traverse all neighbouring nodes and then move forwards. This algorithm implements **Queue Data Structure** to store intermediate nodes and terminates when Queue is exhausted or empty (depends on how it is used) and visited array is used to keep tracks of already visited nodes so they don't get visited again. Normal **Set Data Structure** can also be used for maintaining visited nodes if number of nodes are unknown, below code uses **Set** to store visited nodes.

Algorithm:

Step 1: Take an Empty Queue.

Step 2: Select a starting node (visiting a node) and insert it into the Queue.

Step 3: Provided that the Queue is not empty, extract the node from the Queue and insert its child nodes (exploring a node) into the Queue.

Step 4: Print the extracted node.

Breadth-First Search Algorithm Pseudocode

1. Input: s as the source node

2. BFS (G, s)

3. Let Q be queue.
4. Q.enqueue(s)
5. Mark s as visited
6. While(Q is not empty)
7. $v = Q.dequeue()$
8. For all neighbours w of v in Graph G
9. If w is not visited
10. Q.enqueue(w)
11. Mark w as visited

In the above code, the following steps are executed:

1. (G, s) is input, here G is the graph and s is the root node
2. A queue Q is created and initialized with the source node s
3. All child nodes of s are marked.
4. Extract s from queue and visit the child nodes
5. Process all the child nodes of v
6. Stores w (child nodes) in Q to further visit its child nodes
7. Continue till Q is empty

Code:

#For Creating Nodes

class Node:

"""Creating a new node and adding all its neighbour"""

def __init__(self,value,Node_map,neighbours = set()):

self.value = value

self.neighbours = set()

self.add_neighbours(Node_map,neighbours)

 """Adding Neighbours to a node(bidirectional edges) by checking neighbour node exist or not,
 if doesn't creating one

"""

def add_neighbours(self,Node_map,neighbours=set()):

for neighbour in neighbours:

if neighbour not in Node_map:

temp = Node(neighbour,Node_map)

Node_map[neighbour] = temp

temp.neighbours.add(self.value)

self.neighbours.add(temp.value)

else:

self.neighbours.add(neighbour)

Node_map[neighbour].neighbours.add(self.value)

class Graph:

def __init__(self):

 self.Node_map = { } #Keeping accounting of nodes added till now with their value and
references

"""Simply create a Node if doesn't exist using Node class

if Node exist only add edges

"""

def create_node(self,value,neighbours = set()):

if value in self.Node_map:

self.add_neighbours(value,neighbours)

else:

node = Node(value,self.Node_map,neighbours)

self.Node_map[value] = node

def add_neighbours(self,value, neighbours):

#Before adding neighbour to node first checking if node exist or node, if not raise error


```

assert value in self.Node_map, "Given node doesn't exist"
node = self.Node_map[value]
node.add_neighbours(self.Node_map,neighbours)

```

```

def bfs(self,value =None):
    visited = set()
    que = []
    if value == None:
        #picking up random node since no Node is passed
        que.append(list(self.Node_map.keys())[0])
    else:
        #Cheking if passed Node exist or not, if not raise error
        assert value in self.Node_map, "Node doesn't exist"
        que.append(value)

    visited.add(que[0])
    for value in(que):
        for neighbour in self.Node_map[value].neighbours:
            if neighbour not in visited:
                que.append(neighbour)
                visited.add(neighbour)
    print(value,end=" ")

```

Output:

```

g1 = Graph()
g1.create_node("A",{ "B", "E", "F" })
g1.create_node("B",{ "C", "D" })
g1.create_node("C",{ "H", "J" })
g1.add_neighbours("C",{ "D", "I" })

```

```

g1.bfs() #it pickup random node when no parameter is passed

```

```

E A F B C D I H J

```

```

g1.bfs("A")

```

```

A F B E C D I H J

```

Conclusion: We have successfully implemented BFS program using Python.

Experiment no. 3

Aim: To implement Depth First Search (DFS) algorithm.

Requirement: Compatible version of python.

Theory:

Depth-first search (DFS) is an algorithm for searching a graph or tree data structure. The algorithm starts at the root (top) node of a tree and goes as far as it can down a given branch (path), then backtracks until it finds an unexplored path, and then explores it. The algorithm does this until the entire graph has been explored. Many problems in computer science can be thought of in terms of graphs. For example, analysing networks, mapping routes, scheduling, and finding spanning trees are graph problems. To analyse these problems, graph-search algorithms like depth-first search are useful. Depth-first searches are often used as subroutines in other more complex algorithms. For example, the matching algorithm, Hopcroft–Karp, uses a DFS as part of its algorithm to help to find a matching in a graph. DFS is also used in tree-traversal algorithms, also known as tree searches, which have applications in the traveling-salesman problem and the Ford-Fulkerson algorithm.

Algorithm:

Iterative Algorithm:

1. Initialize empty stack and visited set.
2. Pick up random a node from Graph or Tree and push it in to the stack.
3. Pop element of stack mark it as visited and print it.
4. Push all the non-visited neighbours of pop element in step 3 in to the stack.
5. Repeat steps 3 and 4 until stack is empty.

Recursive Algorithm:

1. Pass empty visited set and a random node as parameters to algorithm.
2. Mark the randomly selected node as visited and print it.
3. Call recursive DFS on all the non-visited neighbours of node with neighbour node and visited set as parameters.

Since, DFS is not unique and output depends on which neighbour is selected first result of Iterative and Recursive algorithm may vary.

Implementation:

#For Creating Nodes
class Node:

```
    """Creating a new node and adding all its neighbour"""
    def __init__(self,value,Node_map,neighbours = set()):
        self.value = value
        self.neighbours = set()
        self.add_neighbours(Node_map,neighbours)
```

```
    """Adding Neighbours to a node(bidirectional edges) by checking neighbour node exist or
    not,
    if doesn't creating one
    """
```

```
    def add_neighbours(self,Node_map,neighbours=set()):
        for neighbour in neighbours:
            if neighbour not in Node_map:
                temp = Node(neighbour,Node_map)
                Node_map[neighbour] = temp
                temp.neighbours.add(self.value)
                self.neighbours.add(temp.value)

            else:
                self.neighbours.add(neighbour)
                Node_map[neighbour].neighbours.add(self.value)
```

class Graph:

```
    def __init__(self):
        self.Node_map = { } #Keeping accounting of nodes added till now with their value and
        references
```

```

"""Simply create a Node if doesn't exist using Node class
   if Node exist only add edges
"""

```

```

def create_node(self,value,neighbours = set()):

```

```

    if value in self.Node_map:
        self.add_neighbours(value,neighbours)
    else:
        node = Node(value,self.Node_map,neighbours)
        self.Node_map[value] = node

```

```

def add_neighbours(self,value, neighbours):

```

```

    #Before adding neighbour to node first checking if node exist or node, if not raise error
    assert value in self.Node_map, "Given node doesn't exist"
    node = self.Node_map[value]
    node.add_neighbours(self.Node_map,neighbours)

```

```

def dfs(self,value = None,visited = set()):

```

```

    if not value:
        value = list(self.Node_map.keys())[0]
        print(f"Randomly pick up: {value}")
    else:
        assert value in self.Node_map, "Node is not present"

```

```

    visited.add(value)
    print(value, end = " ")
    for neighbour in self.Node_map[value].neighbours:
        if neighbour not in visited:
            self.dfs(neighbour,visited)

```

```

def iterative_dfs(self, value = None):

```

```

    if not value:
        value = list(self.Node_map.keys())[0]
        print(f"Randomly pick up: {value}")
    else:
        assert value in self.Node_map, "Node is not present"
    stack = [value]
    visited = set()
    while(stack):
        value = stack.pop()
        visited.add(value)
        for neighbour in self.Node_map[value].neighbours:
            if neighbour not in visited:
                stack.append(neighbour)
                visited.add(neighbour)
        print(value, end=" ")

```

Output:

```
In [18]: g1.dfs("A", visited = set())
```

```
A B D C J H I E F
```

```
In [19]: g1.iterative_dfs("A")
```

```
A F E B C I H J D
```

```
In [20]: g1.dfs(visited=set())
```

```
Randomly pick up: B  
B D C J H I A E F
```

```
In [21]: g1.iterative_dfs()
```

```
Randomly pick up: B  
B A F E C I H J D
```

Conclusion: We have successfully implemented recursive and iterative DFS in python.

Experiment No. 4

Aim: To implement uniform cost search.

Requirements: Compatible version of python.

Theory:

Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search expands nodes according to their path costs from the root node. It can be used to solve any graph/tree where the optimal cost is in demand. A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost. Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

Uniform cost search algorithm:

1. Initialize visited, parent and Priority Queue data structure.
2. Enqueue source node into Priority Queue and add it in parent and set parent as Null.
3. While goal is not found repeat below steps.
4. Dequeue a node from Priority Queue and mark it as visited
5. Visit its neighbours and check if it is visited or not if it is visited continue else if it is already has some other parent check if current node has lower cost than current parent node if yes change the current parent node with current node.
6. If goal is found traverse parent data structure and reverse it.

Advantages:

- o Uniform cost search is optimal because at every state the path with the least cost is chosen.

Disadvantages:

- o It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Implementation:

```
from queue import PriorityQueue
```

```
class Node:
```

```
    def __init__(self,Node_map,value = None, neighbours = {}):
        self.add_node(value = value, neighbours = neighbours,Node_map = Node_map)
```

```
    def add_node(self, Node_map, value = None ,neighbours = {}):
        assert value!=None, "Value cannot be none"
        if value in Node_map:
            Node_map[value].add_neighbours(Node_map = Node_map, neighbours =
neighbours)
        return
```

```
        self.value = value
        Node_map[value] = self
        self.neighbours = {}
        self.add_neighbours(neighbours = neighbours,Node_map = Node_map)
```

```
    def add_neighbours(self, Node_map, neighbours = {}):
        for neighbour, weight in neighbours.items():
            if neighbour not in Node_map:
                n = Node(value = neighbour, Node_map= Node_map)

            self.neighbours[neighbour] = weight
```

```
class Graph:
```

```
    def __init__(self):
        self.Node_map = {}
```

```
    def add_node(self, value = None, neighbours = {}):
        node = Node(value = value,neighbours = neighbours,Node_map = self.Node_map)
```

```
    def add_neighbours(self,value = None, neighbours = {}):
        assert value in self.Node_map, "Node doesn't exist"
```

```

self.Node_map[value].add_neighbours(neighbours = neighbours, Node_map =
self.Node_map)

def print_graph(self):
    for key, value in self.Node_map.items():
        print(f"key: {key}, value: {value.neighbours}")

def uniform_search(self, source = None, goal = None):
    assert source != None and goal != None, "Source and goal node cannot be None"
    assert source in self.Node_map and goal in self.Node_map, "Source or Goal node
doesn't exist"

    parent = {source: (None, 0)}
    visited = set()
    min_queue = PriorityQueue()
    min_queue.put(source)

    while(True):
        node = min_queue.get()
        if node == goal:
            break
        visited.add(min_queue)

        for neighbour, weight in self.Node_map[node].neighbours.items():
            if neighbour in visited:
                continue
            min_queue.put(neighbour)
            if neighbour in parent:
                if weight < self.Node_map[parent[neighbour][0]].neighbours[neighbour]:
                    parent[neighbour] = (node, weight)
            else:
                parent[neighbour] = (node, weight)

    node = goal
    ls = []

    total = 0
    while(parent[node][0]):
        total += parent[node][1]
        ls.append((node, parent[node][1]))
        node = parent[node][0]
    ls.append((source, 0))
    ls.reverse()
    for x in ls:
        print(f"Node={x[0]}, Weight={x[1]}-->", end="")
    print("None")
    print(f"Total weight= {total}")

g = Graph()
g.add_node("S", {"A": 1, "G": 12})

```



```
g.add_node("A",{ "B":3, "C":1})
g.add_node("B",{ "D":3})
g.add_node("C",{ "D":1,"G":2})
g.add_node("D",{ "G":3})
g.print_graph()
g.uniform_search(source="S", goal="G")
```

Output:

```
Node: S, neighbours: {'A': 1, 'G': 12}
Node: A, neighbours: {'B': 3, 'C': 1}
Node: G, neighbours: {}
Node: B, neighbours: {'D': 3}
Node: C, neighbours: {'D': 1, 'G': 2}
Node: D, neighbours: {'G': 3}
Node=S, Weight=0-->Node=A, Weight=1-->Node=C, Weight=1-->Node=G, Weight=2-->None
Total weight= 4
```

Conclusion: We have successfully implemented uniform cost search in python.

EXPERIMENT NO. 5

Aim: Write a program to implement Greedy Best First Search.

Requirements: Compatible version of python.

Theory:

The informed search algorithm is also called heuristic search or directed search. In contrast to uninformed search algorithms, informed search algorithms require details such as distance to reach the goal, steps to reach the goal, cost of the paths which makes this algorithm more efficient.

Here, the goal state can be achieved by using the heuristic function.

The heuristic function is used to achieve the goal state with the lowest cost possible. This function estimates how close a state is to the goal.

Algorithm:

Greedy best-first search uses the properties of both depth-first search and breadth-first search. Greedy best-first search traverses the node by selecting the path which appears best at the moment. The closest path is selected by using the heuristic function.

Implementation:

```
from queue import PriorityQueue
```

```
def create_path(parent,dest):  
    temp = []
```

```
    while(dest):  
        temp.append(dest)  
        dest = parent[dest]
```

```
    return list(reversed(temp))
```

```
def gbfs(graph,source,dest,heu):
```

```

parent, visited = {}, set()
que = PriorityQueue()
que.put((heu[source],source))
parent[source] = None
while(not que.empty()):

    current = que.get()[1]

    if current == dest:
        return create_path(parent,dest)

    visited.add(current)

    for neighbour in graph[current]:

        if neighbour in visited:
            continue

        que.put((heu[neighbour],neighbour))
        parent[neighbour] = current

    return "path doesn't exist"

graph = {
    "Arad":["Sibiu", "Timisoara","Zerind"],
    "Sibiu": ["Arad","Fagaras","Oradea", "RimnicuVilcea"],
    "Timisoara": ["RimnicuVilcea"],
    "Zerind": [],
    "Fagaras": ["Sibiu","Bucharest"],
    "Oradea":[],
    "RimnicuVilcea": ["Arad"],
    "Bucharest": ["Zerind"],
}

heu = {
    "Arad": 366, "Bucharest":0, "Fagaras":176,
    "Sibiu":359, "Timisoara": 253, "Zerind":350,
    "Oradea":170, "RimnicuVilcea": 100,
    "Bucharest":0
}

inputs = [
    (graph,"Arad","Bucharest",heu),
    (graph,"Arad","RimnicuVilcea",heu),
    (graph,"Arad","Sinaia",heu)
]

for x in inputs:
    print(f"path from {x[1]} to {x[2]}: {gbfs(*x)}\n")

```

Output:

```
path from Arad to Bucharest: ['Arad', 'Sibiu', 'Fagaras', 'Bucharest']  
path from Arad to RimnicuVilcea: ['Arad', 'Timisoara', 'RimnicuVilcea']  
path from Arad to Sinaia: path doesn't exist
```

Conclusion: We have successfully implemented Greedy Best First Search in python.

EXPERTIMENT NO. 6

Aim: To implement A* search algorithm.

Requirements: Compatible version of python.

Theory:

The most widely known form of best-first search is called A* A search (pronounced “A-star * SEARCH search”). It evaluates nodes by combining $g(n)$, the cost to reach the node, and $h(n)$, the cost to get from the node to the goal: $f(n) = g(n) + h(n)$. Since $g(n)$ gives the path cost from the start node to node n , and $h(n)$ is the estimated cost of the cheapest path from n to the goal, we have $f(n)$ = estimated cost of the cheapest solution through n . Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of $g(n) + h(n)$. It turns out that this strategy is more than just reasonable: provided that the heuristic function $h(n)$ satisfies certain conditions, A* search is both complete and optimal. The algorithm is identical to UNIFORM-COST-SEARCH except that A* uses $g + h$ instead of g .

Algorithm:

1. make an openlist containing only the starting node
2. make an empty closed list
3. while (the destination node has not been reached):
4. consider the node with the lowest f score in the open list
5. if (this node is our destination node) :
6. we are finished
7. if not:
8. put the current node in the closed list and look at all of its neighbours

9. for (each neighbour of the current node):
 - a. if (neighbour has lower g value than current and is in the closed list) :
 - i. replace the neighbour with the new, lower, g value
 - ii. current node is now the neighbour's parent
 - b. else if (current g value is lower and this neighbour is in the open list) :
 - i. replace the neighbour with the new, lower, g value
 - ii. change the neighbour's parent to our current node
 - c. else if this neighbour is not in both lists:
 - i. add it to the open list and set its g

Implementation:

```
from queue import PriorityQueue
```

```
def create_path(parent,dest):
```

```
    temp = []
```

```
    while(dest):
```

```
        temp.append((dest,parent[dest][0]))
```

```
        dest = parent[dest][1]
```

```
    return list(reversed(temp))
```

```
def gbfs(graph,source,dest,heu):
```

```
    parent, close_ls = { }, set()
```

```
    open_ls = PriorityQueue()
```

```
    open_ls.put((heu[source],source))
```

```
    parent[source] = (heu[source], None)
```

```
    total_cost = { }
```

```
    while(not open_ls.empty()):
```

```
        current_cost, current = open_ls.get()
```

```
        if current in close_ls:
```

```
            continue
```

```
        if current == dest:
```

```
            return create_path(parent,dest),current_cost
```

```

    close_ls.add(current)

    for cost,neighbour in graph[current]:

        if neighbour in close_ls:
            continue

        temp = cost+heu[neighbour]

        if neighbour not in total_cost:
            total_cost[neighbour] = temp
            open_ls.put((temp,neighbour))
            parent[neighbour] = (temp,current)

        elif temp < total_cost[neighbour]:
            total_cost[neighbour] = temp
            open_ls.put((temp,neighbour))
            parent[neighbour] = (temp,current)

    return "path doesn't exist"

graph = {
    "Arad":[(140,"Sibiu"), (118,"Timisoara"),(75,"Zerind")],
    "Sibiu": [(280,"Arad"),(239,"Fagaras"),(291,"Oradea"), (220,"RimnicuVilcea")],
    "Timisoara": [(200,"RimnicuVilcea")],
    "Zerind": [],
    "Fagaras": [(338,"Sibiu"),(450,"Bucharest")],
    "Oradea":[],
    "RimnicuVilcea": [(366,"Craiova"),(317,"Pitesti"),(300,"Sibiu")],
    "Bucharest": [(100,"Zerind")],
    "Craiova":[],
    "Pitesti":[(418,"Bucharest"),(455,"Craiova"),(414,"RimnicuVilcea")]
}

heu = {
    "Arad": 366, "Bucharest":0, "Fagaras":176,
    "Sibiu":253, "Timisoara": 329, "Zerind":374,
    "Oradea":380, "RimnicuVilcea": 193,
    "Craiova":160, "Pitesti":100,
}

inputs = [
    (graph,"Arad","Bucharest",heu),
    (graph,"Arad","RimnicuVilcea",heu),
    (graph,"Arad","Sinaia",heu)
]

for x in inputs:
    result = gbfs(*x)

```

```
print(f"path from {x[1]} to {x[2]}: {result[0]}\ncost: {result[1]}\n")\nif len(result) <= 2 else print(f"path from {x[1]} to {x[2]}: {result}\n")
```

Output:

```
path from Arad to Bucharest: [('Arad', 366), ('Sibiu', 393), ('RimnicuVilcea', 413), ('Pitesti', 417), ('Bucharest', 418)]\ncost: 418
```

```
path from Arad to RimnicuVilcea: [('Arad', 366), ('Sibiu', 393), ('RimnicuVilcea', 413)]\ncost: 413
```

```
path from Arad to Sinaia: path doesn't exist
```

Conclusion: We have successfully implemented A* search algorithm in python.

Experiment no. 7

Aim: To implement basic code in Prolog.

Requirements: Compatible version of SWI-Prolog.

Theory:

Prolog as the name itself suggests, is the short form of LOGical PROgramming. It is a logical and declarative programming language. Before diving deep into the concepts of Prolog, let us first understand what exactly logical programming is.

Logic Programming is one of the Computer Programming Paradigm, in which the program statements express the facts and rules about different problems within a system of formal logic. Here, the rules are written in the form of logical clauses, where head and body are present. For example, H is head and B1, B2, B3 are the elements of the body. Now if we state that “H is true, when B1, B2, B3 all are true”, this is a rule. On the other hand, facts are like the rules, but without any body. So, an example of fact is “H is true”.

Some logic programming languages like Datalog or ASP (Answer Set Programming) are known as purely declarative languages. These languages allow statements about what the program should accomplish. There is no such step-by-step instruction on how to perform the task. However, other languages like Prolog, have declarative and also imperative properties. This may also include procedural statements like “To solve the problem H, perform B1, B2 and B3”.

Prolog or **PRO**gramming in **LOG**ics is a logical and declarative programming language. It is one major example of the fourth generation language that supports the declarative programming paradigm. This is particularly suitable for programs that involve **symbolic** or **non-numeric computation**. This is the main reason to use Prolog as the programming

language in **Artificial Intelligence**, where **symbol manipulation** and **inference manipulation** are the fundamental tasks.

In Prolog, we need not mention the way how one problem can be solved, we just need to mention what the problem is, so that Prolog automatically solves it. However, in Prolog we are supposed to give clues as the **solution method**.

Prolog language basically has three different elements –

Facts – The fact is predicate that is true, for example, if we say, “Tom is the son of Jack”, then this is a fact.

Rules – Rules are extensions of facts that contain conditional clauses. To satisfy a rule these conditions should be met. For example, if we define a rule as –

grandfather(X, Y) :- father(X, Z), parent(Z, Y)

This implies that for X to be the grandfather of Y, Z should be a parent of Y and X should be father of Z.

Facts

We can define fact as an explicit relationship between objects, and properties these objects might have. So facts are unconditionally true in nature. Suppose we have some facts as given below –

- Tom is a cat
- Kunal loves to eat Pasta
- Hair is black
- Nawaz loves to play games
- Pratyusha is lazy.

So these are some facts that are unconditionally true. These are actually statements, that we have to consider as true.

Following are some guidelines to write facts –

- Names of properties/relationships begin with lower case letters. • The relationship name appears as the first term.
- Objects appear as comma-separated arguments within parentheses. • A period "." must end a fact.
- Objects also begin with lower case letters. They also can begin with digits (like 1234), and can be strings of characters enclosed in quotes e.g. color('pink', 'red').
- phoneno(agnibha, 1122334455). Is also called a predicate or clause. **Syntax**

The syntax for facts is as follows –

relation (object1,object2...).

Example

Following is an example of the above concept –

cat(tom).

loves_to_eat(kunal,pasta).

of_color(hair,black).

loves_to_play_games(nawaz).

lazy(pratyusha).

Rules

We can define rule as an implicit relationship between objects. So facts are conditionally true.

So when one associated condition is true, then the predicate is also true. Suppose we have some rules as given below –

- Lili is happy if she dances.
- Tom is hungry if he is searching for food.
- Jack and Bili are friends if both of them love to play cricket.
- will go to play if school is closed, and he is free.

So these are some rules that are **conditionally** true, so when the right hand side is true, then the left hand side is also true.

Here the symbol (:-) will be pronounced as “If”, or “is implied by”. This is also known as neck symbol, the LHS of this symbol is called the Head, and right hand side is called Body.

Here we can use comma (,) which is known as conjunction, and we can also use semicolon, that is known as disjunction.

Syntax

rule_name(object1, object2, ...) :- fact/rule(object1, object2, ...)

Suppose a clause is like :

P :- Q;R.

This can also be written as

P :- Q.

P :- R.

If one clause is like :

$P :- Q,R;S,T,U.$

Is understood as

$P :- (Q,R);(S,T,U).$

Or can also be written as:

$P :- Q,R.$

$P :- S,T,U.$

Example

$happy(lili) :- dances(lili).$

$hungry(tom) :- search_for_food(tom).$

$friends(jack, bili) :- lovesCricket(jack), lovesCricket(bili).$
 $goToPlay(ryan) :- isClosed(school),$
 $free(ryan).$

Queries

Queries are some questions on the relationships between objects and object properties. So question can be anything, as given below –

- Is tom a cat?
- Does Kunal love to eat pasta?
- Is Lili happy?
- Will Ryan go to play?

So according to these queries, Logic programming language can find the answer and return them.

Code:

mouse(jerry).

cat(tom).

search_for_food(tom).

hungry(X):- search_for_food(X).

will_eat(X,Y):- cat(X),hungry(X),mouse(Y).

play(jack,cricket).

play(billi,cricket).

friends(X,Y):- play(X,Z),play(Y,Z).

eat(kunal,pasta).

loves_to_eat(X,Y):- eat(X,Y).

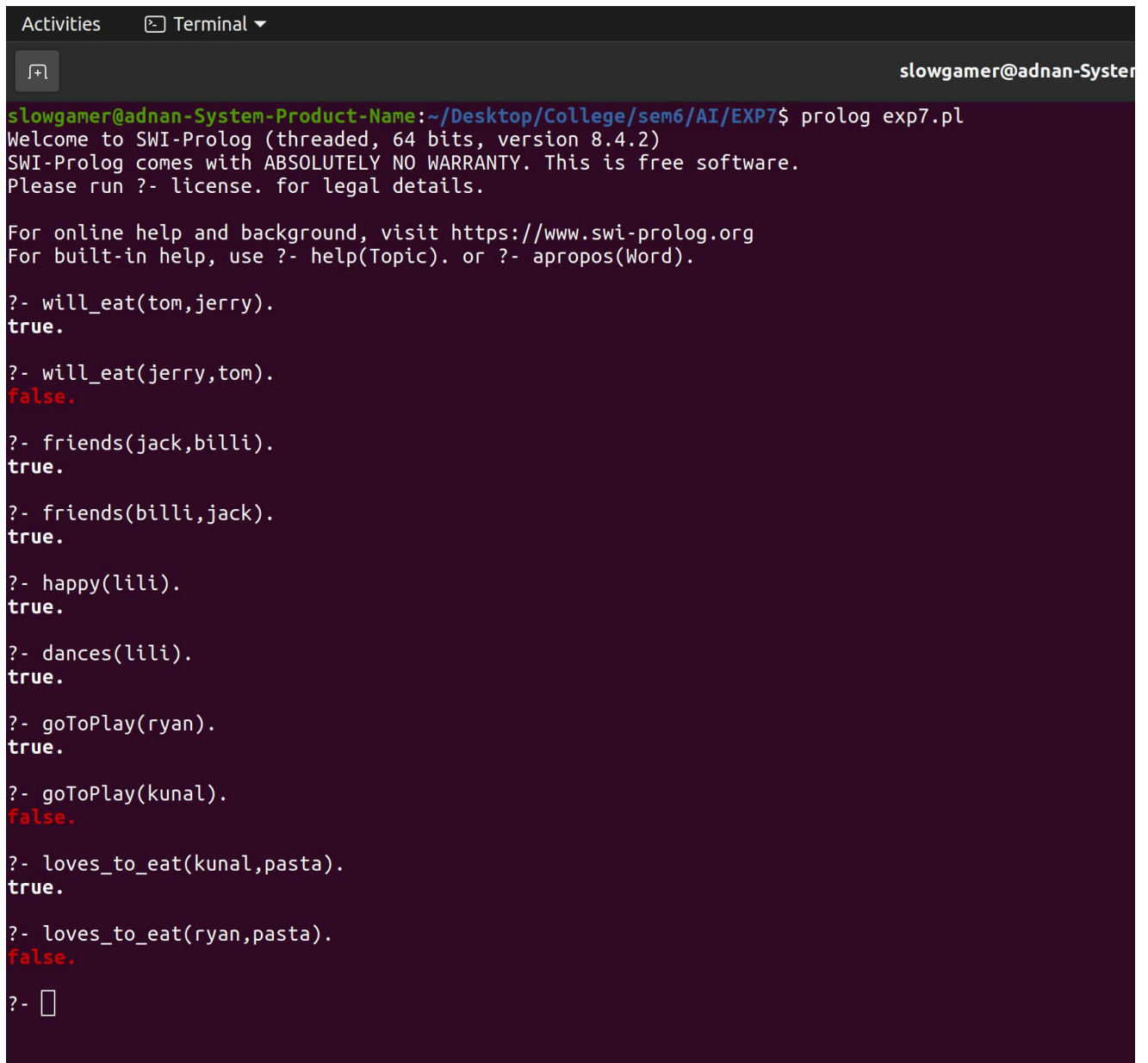
dances(lili).

happy(X):- dances(X).

is_Closed(school).

free(ryan).

goToPlay(X):- is_Closed(school),free(X).

Queries:A terminal window titled 'Terminal' with a dark background. The prompt is 'slowgamer@adnan-System-Product-Name:~/Desktop/College/sem6/AI/EXP7\$'. The user has run 'prolog exp7.pl'. The terminal shows the Prolog welcome message and several queries with their results. The queries and results are: 'will_eat(tom,jerry). true.', 'will_eat(jerry,tom). false.', 'friends(jack,billi). true.', 'friends(billi,jack). true.', 'happy(lili). true.', 'dances(lili). true.', 'goToPlay(ryan). true.', 'goToPlay(kunal). false.', 'loves_to_eat(kunal,pasta). true.', 'loves_to_eat(ryan,pasta). false.', and a final prompt '?- ' with a cursor.

```
slowgamer@adnan-System-Product-Name:~/Desktop/College/sem6/AI/EXP7$ prolog exp7.pl
Welcome to SWI-Prolog (threaded, 64 bits, version 8.4.2)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- will_eat(tom,jerry).
true.

?- will_eat(jerry,tom).
false.

?- friends(jack,billi).
true.

?- friends(billi,jack).
true.

?- happy(lili).
true.

?- dances(lili).
true.

?- goToPlay(ryan).
true.

?- goToPlay(kunal).
false.

?- loves_to_eat(kunal,pasta).
true.

?- loves_to_eat(ryan,pasta).
false.

?- 
```

Conclusion: We have successfully implemented basic code of Prolog.

Experiment no. 8

Aim: To find max of two number using Prolog.

Requirements: Compatible version of SWI-Prolog.

Theory:

There is a built-in predicate construction in Prolog which allows you to express exactly such conditions: the if-then-else construct. In Prolog, *if A then B else C* is written as (A -> B ; C). To Prolog this means: try A. If you can prove it, go on to prove B and ignore C. If A fails, however, go on to prove C ignoring B. The max predicate using the if-then-else construct looks as follows:

max(X,Y,Z) :-

```
( X =< Y
  -> Z = Y
  ; Z = X
  ).
```

Prolog's Persistence

- When a subgoal fails, Prolog will backtrack to the most recent successful goal and try to find another solution.
- Once there are no more solutions for this subgoal it will backtrack again; retrying every subgoal before failing the parent goal.
- A call can match any clause head.
- A redo ignores old matches.

Cut !

The cut, in Prolog, is a goal, written as `!`, which always succeeds, but cannot be backtracked past. It is used to prevent unwanted backtracking, for example, to prevent extra solutions being found by Prolog.

Code:

```
/*Without Cut*/
```

```
maximum(X,Y,Z):- (X>=Y-> Z=X; Z=Y).
```

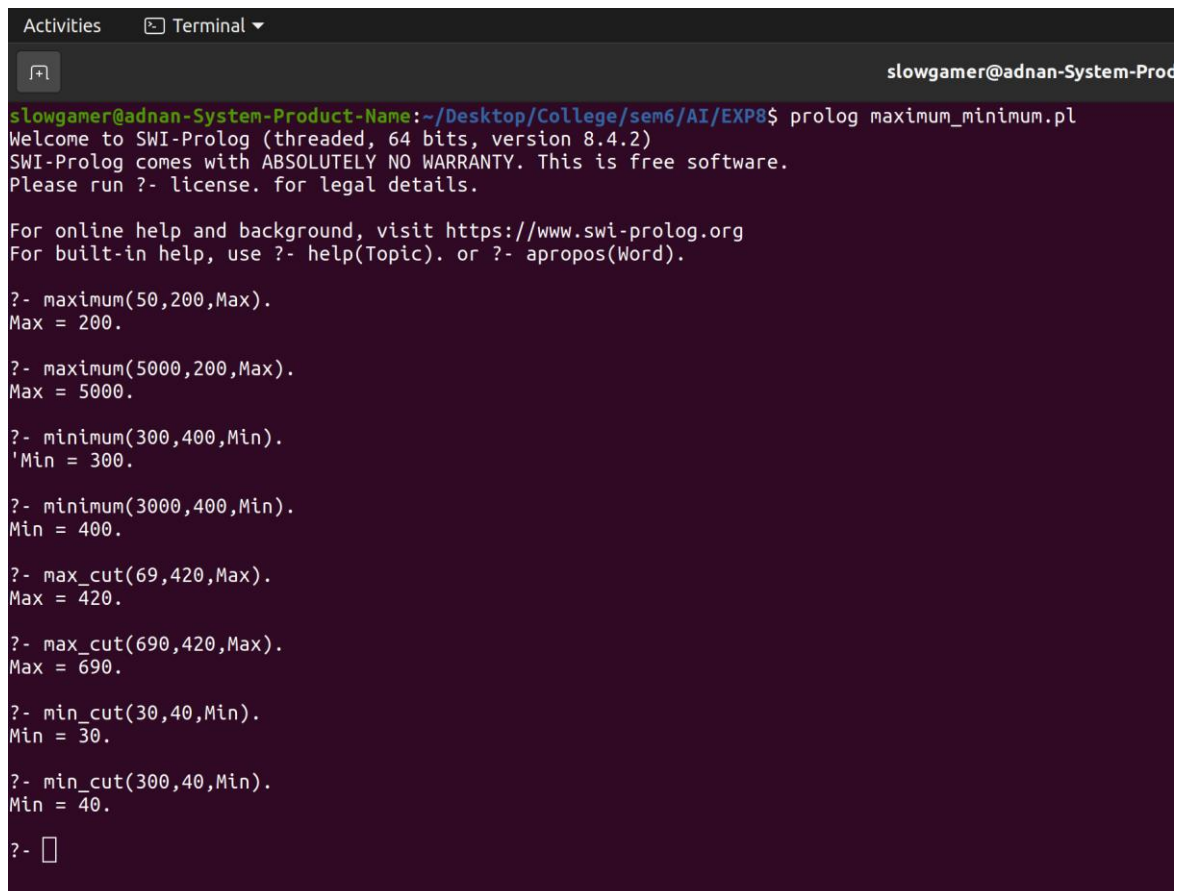
```
minimum(X,Y,Z):- (X>=Y-> Z=Y; Z=X).
```

```
/*With Cut*/
```

```
max_cut(X,Y,Max):- X>=Y,!,Max=X; Max=Y.
```

```
min_cut(X,Y,Min):- X>=Y,!,Min=Y; Min=X.
```

Output:



```
slowgamer@adnan-System-Product-Name:~/Desktop/College/sem6/AI/EXP8$ prolog maximum_minimum.pl
Welcome to SWI-Prolog (threaded, 64 bits, version 8.4.2)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- maximum(50,200,Max).
Max = 200.

?- maximum(5000,200,Max).
Max = 5000.

?- minimum(300,400,Min).
Min = 300.

?- minimum(3000,400,Min).
Min = 400.

?- max_cut(69,420,Max).
Max = 420.

?- max_cut(690,420,Max).
Max = 690.

?- min_cut(30,40,Min).
Min = 30.

?- min_cut(300,40,Min).
Min = 40.

?- 
```

Conclusion: We have successfully implemented Maximum number finding code in Prolog.

Experiment No. 9

Aim: To implement family tree using recursion in Prolog.

Requirements: Compatible version of Prolog.

Theory:

Recursion

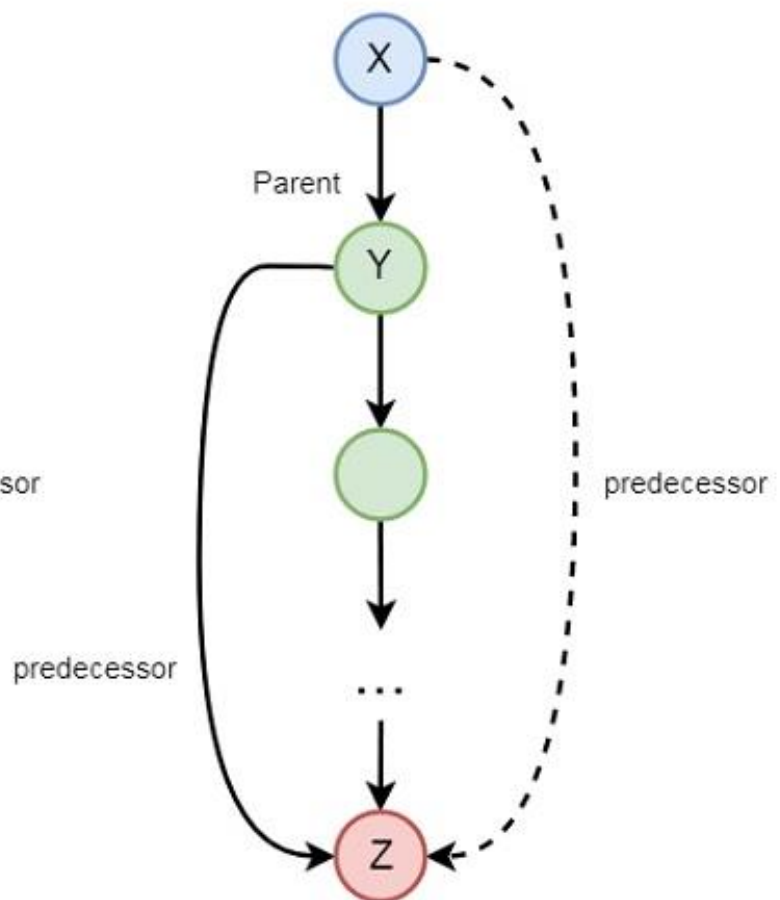
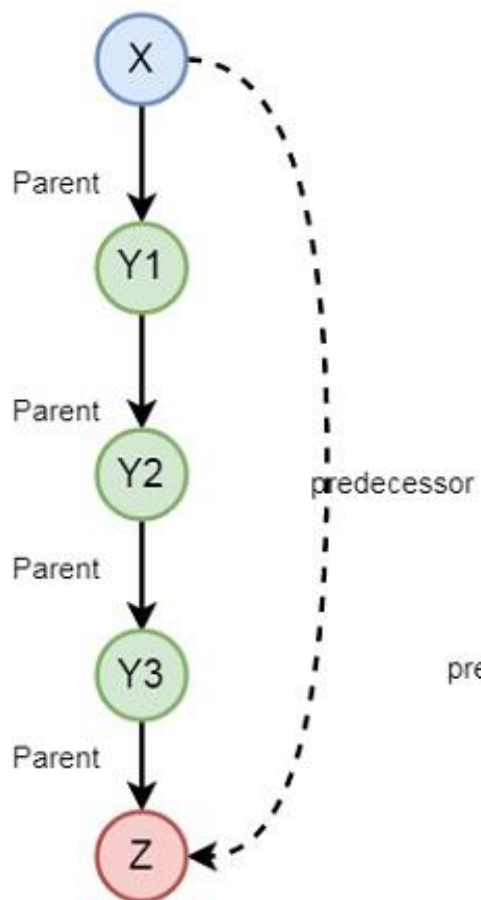
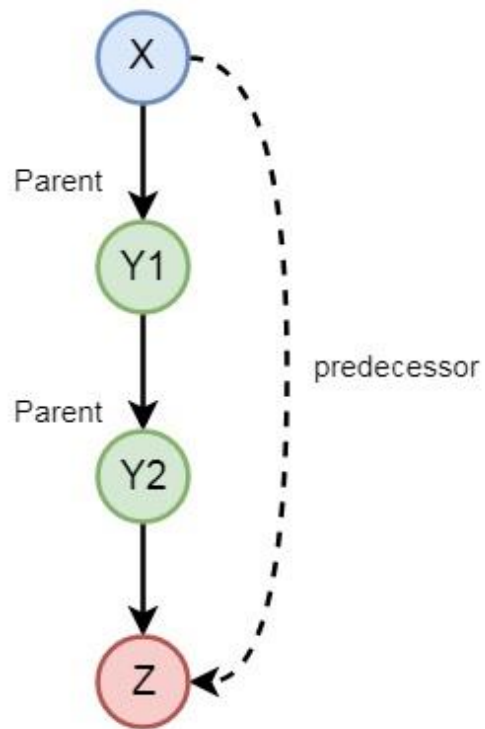
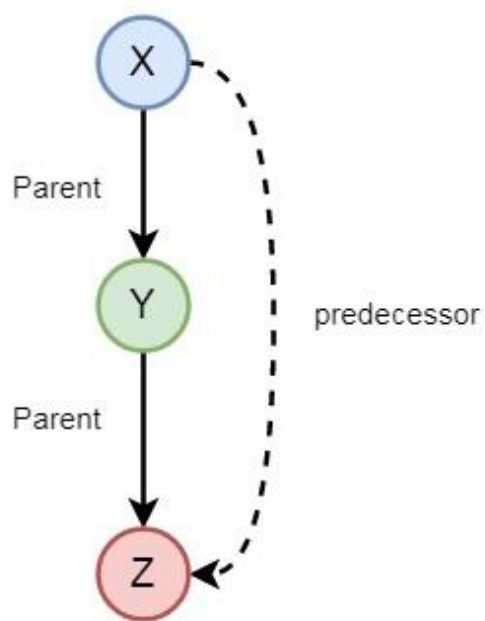
Recursion is a technique in which one predicate uses itself (may be with some other predicates) to find the truth value.

Let us understand this definition with the help of an example –

- $\text{is_digesting}(X,Y) :- \text{just_ate}(X,Y).$
- $\text{is_digesting}(X,Y) :- \text{just_ate}(X,Z), \text{is_digesting}(Z,Y).$

So this predicate is recursive in nature. Suppose we say that $\text{just_ate}(\text{deer}, \text{grass})$, it means $\text{is_digesting}(\text{deer}, \text{grass})$ is true. Now if we say $\text{is_digesting}(\text{tiger}, \text{grass})$, this will be true if $\text{is_digesting}(\text{tiger}, \text{grass}) :- \text{just_ate}(\text{tiger}, \text{deer}), \text{is_digesting}(\text{deer}, \text{grass})$, then the statement $\text{is_digesting}(\text{tiger}, \text{grass})$ is also true.

There may be some other examples also, so let us see one family example. So if we want to express the predecessor logic, that can be expressed using the following diagram –



So we can understand the predecessor relationship is recursive. We can express this relationship using the following syntax –

- predecessor(X, Z) :- parent(X, Z).
- predecessor(X, Z) :- parent(X, Y),predecessor(Y, Z).

Code:

```
male(james).
male(charles).
male(george).
male(james).
```

```
female(catherine).
female(elizabeth).
female(sophia).
```

```
mother(catherine,charles).
mother(catherine,george).
mother(catherine,elizabeth).
```

```
sister(catherine,sophia).
```

```
partner(james,catherine).
```

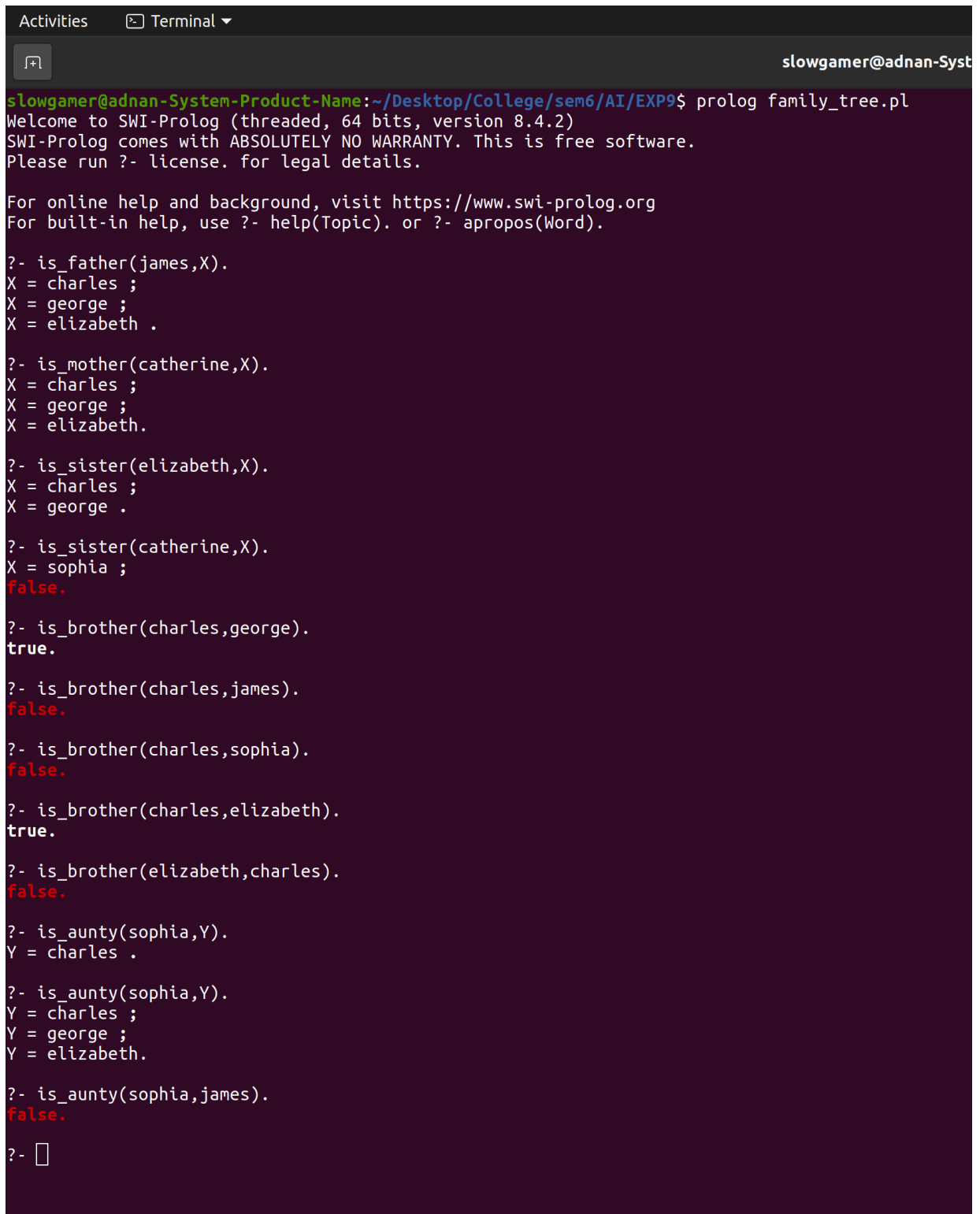
```
is_father(X,Y):- male(X),partner(X,Z),mother(Z,Y).
```

```
is_mother(X,Y):- mother(X,Y).
```

```
is_sister(X,Y):- sister(X,Y); (female(X), mother(Z,X),mother(Z,Y)).
```

```
is_brother(X,Y):- male(X), mother(Z,X), mother(Z,Y).
```

```
is_aunty(X,Y):- female(X), sister(Z,X), mother(Z,Y).
```

Output:


```

Activities  Terminal ▾
slowgamer@adnan-Syst
slowgamer@adnan-System-Product-Name:~/Desktop/College/sem6/AI/EXP9$ prolog family_tree.pl
Welcome to SWI-Prolog (threaded, 64 bits, version 8.4.2)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- is_father(james,X).
X = charles ;
X = george ;
X = elizabeth .

?- is_mother(catherine,X).
X = charles ;
X = george ;
X = elizabeth.

?- is_sister(elizabeth,X).
X = charles ;
X = george .

?- is_sister(catherine,X).
X = sophia ;
false.

?- is_brother(charles,george).
true.

?- is_brother(charles,james).
false.

?- is_brother(charles,sophia).
false.

?- is_brother(charles,elizabeth).
true.

?- is_brother(elizabeth,charles).
false.

?- is_aunty(sophia,Y).
Y = charles .

?- is_aunty(sophia,Y).
Y = charles ;
Y = george ;
Y = elizabeth.

?- is_aunty(sophia,james).
false.

?- 

```

Conclusion: We have successfully implemented family tree using recursion in Prolog.

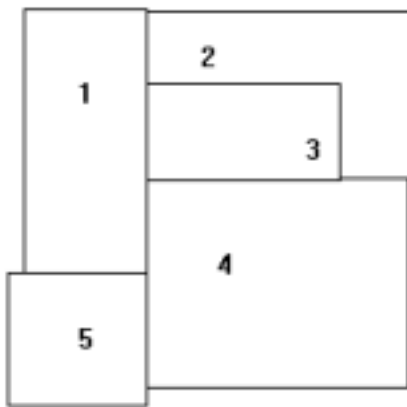
Experiment No. 10

Aim: To implement graph coloring problem in Prolog.

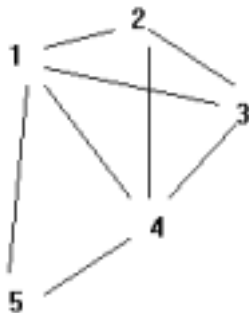
Requirements: Compatible version of SWI-Prolog.

Theory:

A famous problem in mathematics concerns coloring adjacent planar regions. Like cartographic maps, it is required that, whatever colors are actually used, no two adjacent regions may not have the same color. Two regions are considered adjacent provided they share some boundary line segment. Consider the following map.



We have given numerical names to the regions. To represent which regions are adjacent, consider also the following graph.



Here we have erased the original boundaries and have instead drawn an arc between the names of two regions, provided they were adjacent in the original drawing. In fact, the adjacency graph will convey all of the original adjacency information.

Implementation:

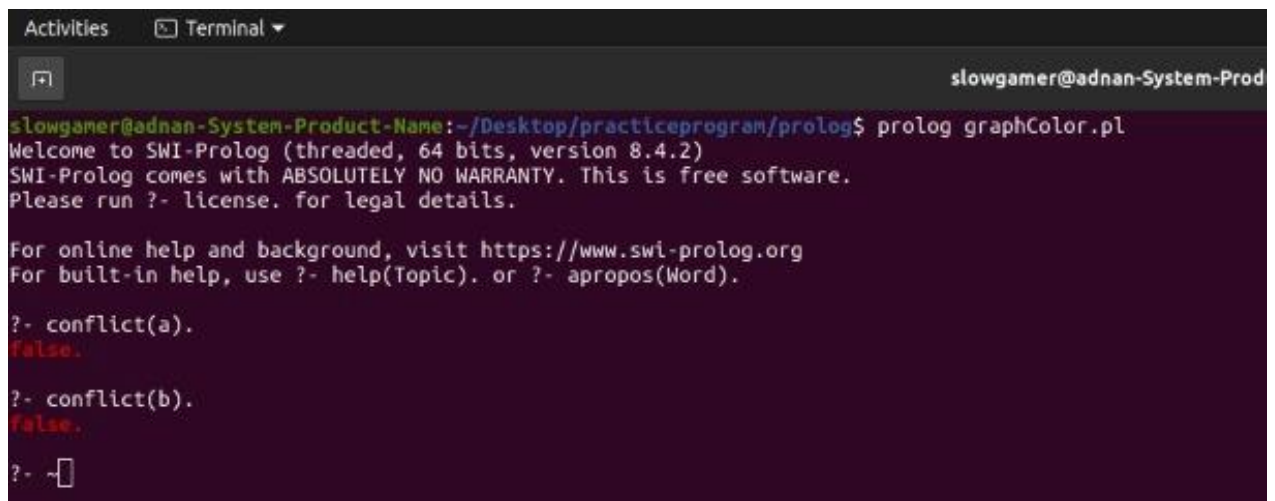
```
color(1,red,a).
color(2,blue,a).
color(3,green,a).
color(4,yellow,a).
color(5,blue,a).
color(1,red,b).
color(2,blue,b).
color(3,green,b).
color(4,yellow,b).
color(5,blue,b).
```

```
adj(1,2).
adj(1,3).
adj(1,4).
adj(1,5).
adj(2,3).
adj(2,4).
adj(3,5).
adj(4,5).
```

```
adjacent(X,Y) :- adj(X,Y);adj(Y,X).
```

```
conflict(Z):- adjacent(X,Y),color(X,K,Z),color(Y,K,Z), write(X),write('->'),write(Y),writeln('
conflict').
```

Output:

A screenshot of a Linux terminal window. The title bar shows 'Activities' and 'Terminal'. The user is 'slowgamer@adnan-System-Prod'. The prompt is 'slowgamer@adnan-System-Product-Name:~/Desktop/practiceprogram/prolog\$'. The command 'prolog graphColor.pl' has been executed. The output shows the SWI-Prolog welcome message, version 8.4.2, and a license notice. The user has entered two queries: '?- conflict(a).', which returned 'false.', and '?- conflict(b).', which also returned 'false.'. The prompt is now '?- ~' with a cursor.

```
slowgamer@adnan-System-Product-Name:~/Desktop/practiceprogram/prolog$ prolog graphColor.pl
Welcome to SWI-Prolog (threaded, 64 bits, version 8.4.2)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- conflict(a).
false.

?- conflict(b).
false.

?- ~
```

Conclusion: We have successfully implemented graph coloring problem in Prolog.

Experiment No. 11

Aim: To calculate factorial of a number using Prolog.

Requirements: Compatible version of SWI-Prolog.

Theory:

Formula:

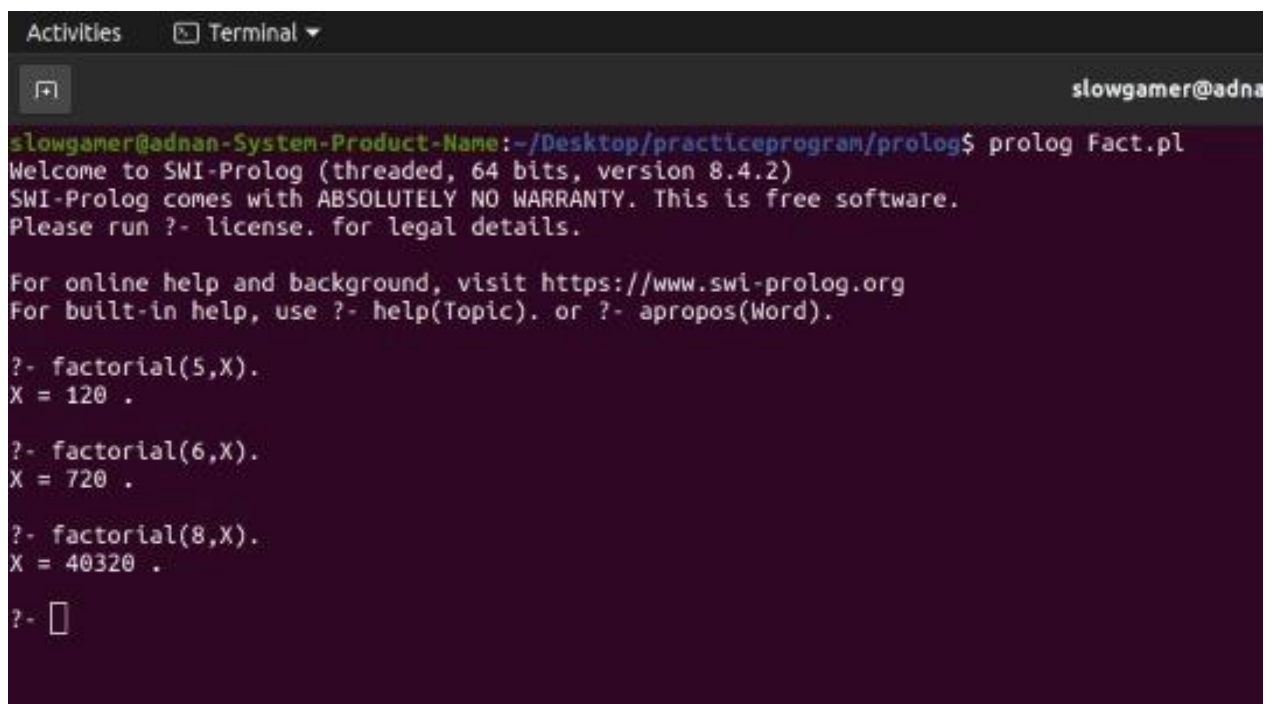
$$F(x) = \begin{cases} 1 & x = 1 \\ x * F(x - 1) & x > 1 \end{cases}$$

Code:

factorial(1,X):- X is 1.

factorial(N,X) :- N1 is N-1, factorial(N1,X1),X is N*X1.

Output:



```

slowgamer@adnan-System-Product-Name:~/Desktop/practiceprogram/prolog$ prolog Fact.pl
Welcome to SWI-Prolog (threaded, 64 bits, version 8.4.2)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- factorial(5,X).
X = 120 .

?- factorial(6,X).
X = 720 .

?- factorial(8,X).
X = 40320 .

?- 

```

Conclusion: We have successfully calculated factorial of two number using Prolog.