SISTER NIVEDITA UNIVERSITY

School of Engineering

Subject Name:	Artificial Intelligence Lab
Subject Code:	
Department:	Computer Science & Engineering

Name:	Shuvom Dhar
Enrolment Number:	22112000010 43
Registration Number:	220011314862
Semester:	5
Academic Year:	3

Index Page

SN	Assignment Name	Page No.	Remarks
1.	Practical - 1		12.09.24
2.	Practical - 2		
3.			
4.			
5.			
6.			
7.			
8.			
9.			
10.			

DATE 9/9/2024 | PAGE NO. 1 | EXPT. NO. 1.1

Family Tree

Objective: This practical will help you to get know the relationship between a family members.

Apparatus: Here we are using SWI-Prolog software to solve this problem.

Procedure: After opening the software we make a new file and there we write the code then we run it. Then we run a query and get an output according to our query.

Program: Fact –

```
parent(sudip, piyus).
parent(sudip, raj).
male(piyus).
male(raj).
```

Rule –

```
brother(X,Y):-parent(Z,X), parent(Z,Y), male(X), male(Y).
```

Input/Output: Output is below –

Query 1: parent(X, piyus)
Output: X = sudip

Query 2: parent (sudip, Y)
Output : Y = piyus

Result : We put two queries to test our code and get the correct output according to our queries.

Conclusion : By doing this practical we can easily find out the relationship between two people in a family.

DATE 9/9/2024 | PAGE NO. 3 | EXPT. NO. 1-2

Family Tree

Objective: This practical will help you to get know the relationship between a family members.

Apparatus: Here we are using SWI-Prolog software to solve this problem.

Procedure: After openning the software we make a new file and there we write the code then we ruen it. Then we ruen a query and get an output accordingly to our query.

Program: Fact –

```
female(pam).
female(liz).
female(pat).
female(ann).
male(jim).
male(bob).
male(tom).
male(peter).
parent(pam, bob).
parent(tom, bob).
parent(tom, liz).
```

DATE 9/9/2024 | PAGE NO. 4 | EXPT. NO. 1.2

```
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).
parent(bob, peter).
parent(peter, jim).
```

Rule —

```
mother(X,Y):- parent(X,Y), female(X).
father(X,Y):- parent(X,Y), male(X).
haschild(X):- parent(X,_)
sister(X,Y):- parent(Z,X), parent(Z,Y), female(X), X\==Y.
brother(X,Y):- parent(Z,X), parent(Z,Y), male(X), X\==Y.
```

Input/Output : Output is below —

Query 1: parent(X, jim)
Output: X = pat

Query 2: mother(X, Y)
Output: X = pam
Y = bob

Query 3: sister(X, Y)
Output: X = liz
Y = bob

DATE 9/9/2024 | PAGE NO. 5 | EXPT. NO. 1.2

Query 4: haschild (X)
Output: X = pam

Result: We put two queries to test our code and get the correct output according to our queries.

Conclusion: By doing this practical we can easily find out the relationship between two people in a family.

DATE 17/9/2024 | PAGE NO. 1 | EXPT. NO. 2.1

Tower of Hanoi

Objective : In this practical we are going to write the python code for 'Tower of Hanoi'.

Apparatus : Here we are using SWI-Prolog software to solve this problem. We can also use VS Code or Jupyter notebook to do this problem.

Program : The program is below —

```
def tower_of_hanoi(n, source, auxiliary, target):
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
        return
    tower_of_hanoi(n-1, source, target, auxiliary)
    print(f"Move disk {n} from {source} to {target}")
    tower_of_hanoi(n-1, auxiliary, source, target)

n = int(input("Enter the number of disks: "))
tower_of_hanoi(n, 'A', 'B', 'C')
```

Output : For n = 3, the output would be:

```
Move disk 1 from A to C
Move disk 2 from A to B
```

Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C

Conclusion : By doing this practical, we find the optimal way to solve this type of problem using recursion.

DATE 17/9/2024 | PAGE NO. 3 | EXPT. NO. 2.2

Breath First Search (BFS)

Objective: In this practical we are going to write the python code for 'Breath First Search (BFS)'.

Apparature: Here we are using SWI-Prolog software to solve this problem. We can also use VS Code or Jypyter notebook to do this problem.

Program: The program is below —

```
from collections import deque
def bfs(graph, start_node):
    visited = set()
    queue = deque([start_node])
    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)
```

DATE 17/9/2024 | PAGE NO. 4 | EXPT. NO. 2.2

```
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

bfs(graph, 'A')
```

Output: The output of the above code is below:

A B C D E F

Conclusion: By doing this practical, we get to know how to traverse a graph breathwise in order to search our desired node.

DATE	PAGE NO.	EXPT. NO.
18/9/2024	5	2.3

Depth First Search (DFS)

Objective: In this practical we are going to write the python code for 'Depth First Search (DFS)'.

Apparatus: Here we are using SWI-Prolog software to solve this problem. We can also use VS Code or Jupyter notebook to do this problem.

Program: The program is below—

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=' ')
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
```

```
    'F':[]
}

dfs(graph, 'A')
```

Output : The output of the above code is below:

A B D E F C

Conclusion : By doing this practical, we get to know how to traverse a graph depthwise in order to search our desired node.

DATE 29/9/2024 | PAGE NO. 1 | EXPT. NO. 3.1

Hill Climbing Algorithm

Objective: In this practical we are going to write the algorithm of 'Hill Climbing' in python.

Apparatus: Here we are going to use VS Code to write the 'Hill Climbing Algorithm'. We can also use Jupyter Notebook.

Program: We are going to write the algorithm below —

```
def hill_climbing(initial_state, get_neighbours, objective_function):
    current_state = initial_state
    current_value = objective_function(current_state)
    while True:
        neighbors = get_neighbors(curren_state)
        next_state = None
        next_value = current_value
        for neighbor in neighbors:
            neighbor_value = objective_function(neighbor)
            if neighbor_value > next_value:
                next_state = neighbor
                next_value = neighbor_value
        if next_state is None or next_value <= current_value:
            return current_state, current_value
```

```
current_state = next_state
current_value = next_value
```

Conclusion: By doing this practical we get to know about the 'Hill Climbing Algorithm' and each and every steps of this practical.

DATE 29/9/2024 | PAGE NO. 3 | EXPT. NO. 3.2

A* Search Algorithm

Objective: In this practical we are going to write the algorithm of 'A* Search Technique' in python programming language.

Apparature: Here we are going to use VS Code to write the 'A* Search Algorithm'. We can also use Jupyter Notebook.

Program: We are going to write the algorithm below —

```
import heapq

def a_star_search(start_node, goal_node, heuristic_function, get_neighbors,
                  cost_function):
    open_list = []
    heapq.heappush(open_list, (0, start_node))
    g_costs = {start_node: 0}
    came_from = {start_node: None}
    while open_list:
        current_f_cost, current_node = heapq.heappop(open_list)
        if current_node == goal_node:
            path = []
            while current_node is not None:
                path.append(current_node)
```

```
            current_node = came_from[current_node]
        return path[::-1]
    for neighbor in get_neighbors(current_node):
        tentative_g_cost = g_costs[current_node] +
                           cost_function(current_node,
                                         neighbor)
        if neighbor not in g_costs or tentative_g_cost
        < g_costs[neighbor]:
            g_costs[neighbor] = tentative_g_cost
            f_cost = tentative_g_cost + heuristic_
            function(neighbor, goal_node)
            heapq.heappush(open_list, (f_cost, neighbor))
            came_from[neighbor] = current_node
return None
```

Conclusion: By doing this practical we get to know about the 'A* Search Algorithm' and each and every steps of this practical.