ARTIFICIAL INTELLIGENCE

PRACTICAL 1

Name: Kartik Jolapara
SAP: 60004200107
Division/Batch: B/B1
Branch: Computer Engineering

Experiment 1

Aim: Identify PEAS for different Applications/Agents

Theory:

1. **Application: Chess playing with a clock**
   Description: Chess is a board game with two players. One player takes white pieces and the other player takes black pieces. The goal of the game is to checkmate the opponent's king, which means to place it under an attack from which it cannot escape or be rescued. Chess clocks are used in chess tournaments to limit the time for each player during a game, making it more difficult for a player to think without worrying about time running out.
   PEAS:
   Performance Measures: Win/lose percentage, time taken
   Environment: Chessboard, clock, chess pieces
   Actuators: Human arm, motors, chess board & pieces
   Sensors: Camera for tracking pieces on board or reed/mechanical switches
   Environment: Fully observable, strategic, sequential, semi-dynamic, discrete and has multiple agents

2. **Application: Learning disability detection**
   Description: By using the many different forms of inputs from the user, we can predict the chances of dyslexia or any other disability for a particular person. The use of AI has been a great help to educators, parents and other professionals who work with children with learning disabilities. It can be used to identify dyslexia, which is a common disability that hinders reading fluency and comprehension. It can also be used for other disabilities like ADHD, autism spectrum disorder, etc.
   PEAS:
   Performance Measures: Low % of the estimation of the certain learning disability
   Environment: Children (usually below the age of 12)
   Actuators: Computer, Written material
   Sensors: Camera, microphones, movement of the eyes

Environment: Fully observable, deterministic, single agent, dynamic, continuous, sequential

3. **Application: Corona guidelines follower**
   Description: An agent to identify whether the person going through is following the corona guidelines. This agent can be assigned to a person who needs assistance, and it will follow them around and mimic their actions as they navigate through an environment. The idea behind this is that it gives the person some sense of independence, and also helps them get used to new surroundings.
   PEAS:
   Performance Measures: Wearing a mask, obeying social distance (of eg. 1m)
   Environment: Normal People, Traffic signals, Crosswalks, the entrance of the public places
   Actuators: Movement/walking of the people, signals, entry points, sanitizer
   Sensors: Cameras(multiple can be used in order of different angles), clothes, mask on the face analysis
   Environment: Fully observable, stochastic, multi-agent, static, discrete activity, episodic in nature

4. **Application: Conduction of an auction**
   Description: When conducting an auction on a large scale basis or locally, there are various participants taking part like bidders and auctioneers. The auction is a process of buying and selling goods, which is done by calling out bids to the highest bidder. Auctions are typically conducted by an auctioneer who may use a variety of methods to conduct the sale, such as shouting or hand gestures. Auctioneers can usually be found at auctions selling anything from antiques and art to cars and furniture. They can also be found at estate sales where they sell items from the deceased's estate.
   PEAS:
   Performance Measures: cost, value, quality, the necessity of the selling item
   Environment: Auctioneers, Bidders, BiddersItems (which are to be bid)
   Actuators: Speakers, microphones, display items, budget
   Sensors: Camera, price monitor, eyes, ears of attendees
   Environment: Partially observable, single agent, stochastic, sequential, dynamic, continuous activity

5. **Application: Satellite image analysis system**
   Description: An application that correctly analyses images from a satellite and classifies it. Satellite images have been used for many years to monitor and analyze the Earth's surface. In recent years, however, satellite imagery has been increasingly used for other purposes such as crop monitoring and fire detection. Satellite image analysis systems are automated systems that can detect objects in satellite images. The system can be operated either manually or automatically.
   PEAS:

Performance Measures: Coverage of land, quality of images, efficiency in classification
Environment: Satellite, space, land, weather
Actuators: Mechanism to position cameras, satellite thrusters
Sensors: RGB cameras and infrared cameras

Environment: Fully Observable, Deterministic, Episodic, Static, Continuous, single agent.

6. **Application: Medical diagnosis system**
   Description: A robot or application that diagnoses the patient's disease and recommends treatments by analyzing patient. AI has a lot of potential and can be used for a variety of tasks. It can be applied to help doctors diagnose patients and provide more accurate results.AI is also being used in the medical field for more than just diagnosis. There are systems that can predict when someone is going to have a heart attack or stroke. These AI-based systems are capable of predicting this before it happens, which could save someone's life.
   PEAS:
   Performance Measures: Healthy patient, minimize costs, lawsuits
   Environment: Patient, hospital, staff
   Actuators: Screen display (questions, tests, diagnoses, treatments, referrals)
   Sensors: Keyboard (entry of symptoms, findings, patient's answers)
   Environment: Partially observable, single-agent, stochastic, sequential, dynamic, continuous

7. **Application: Refinery controller**
   Description: An application that automatically manages the flow, quantity, temperature, and other parameters for the refining process
   PEAS:
   Performance Measures: Purity metric, yield, safety
   Environment: Refinery, valves, material
   Actuators: Valves to control flow, temperature control
   Sensors: Temperature, pressure, and material-specific physical readings
   Environment: Fully observable, stochastic, episodic, dynamic, continuous, single agent

8. **Application: Poker playing**
   Description: A robot that plays poker with humans and tries to win
   PEAS:
   Performance Measures: Win/Lose ratio, Amount earned
   Environment: Poker table, chips, money pool
   Actuators: Human/robotic mechanism to play cards
   Sensors: Visual sensors to see current cards on hand and table
   Environment: Partially observable, multi-agent, stochastic, sequential, static, discrete

9. **Application: Chatbot**

Description: An application that talks with humans to solve queries or hold sensible conversations

PEAS:

Performance Measures: Following up conversations, grammatically accuracy
Environment: Screen, User
Actuators: Writing on screen, replying to messages
Sensors: User input

Environment: Partially observable, multi-agent, stochastic, sequential, dynamic, continuous

10. Application: Soccer playing robot

Description: A robot that plays soccer with humans and tries to win

PEAS:

Performance Measures: Goals hit, fairness, speed, safety
Environment: Soccer field, ball, players
Actuators: Mechanism to hit ball (servos and actuators)
Sensors: RGB cameras, odometry, infrared/depth cameras

Environment: Partially observable, multi-agent, stochastic, sequential, dynamic, continuous

11. Application: Recommender system

Description: An application that tries to recommend similar items to a user that they have interacted with

PEAS:

Performance Measures: Accuracy in the recommendation, similarity score in recommendations
Environment: Users, products
Actuators: Showing recommendations to users
Sensors: User’s choices and favorite products

Environment: Partially observable, multi-agent, stochastic, sequential, dynamic, discrete

Name: Kartik Jolapara **SAPID:** 60004200107

DIV: B/B1

A.I.

Exp2

Aim: Perform uninformed searching techniques like BFS, DFS, etc.

Theory:

BFS:

Breadth-first search is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. BFS or Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighbouring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

There are many ways to traverse the graph, but among them, BFS is the most commonly used approach. It is a recursive algorithm to search all the vertices of a tree or graph data structure. BFS puts every vertex of the graph into two categories - visited and non-visited. It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node.

Algorithm:

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS =

1) and set their

STATUS = 2

(waiting state)

[END OF LOOP]

Step 6: EXIT

Code: graph

```
= {
  "A" : ["B","C"], "B" : ["D","E"],"C":["G"],"D":[" "], "E":["F"],"F":[" "],"G":[" "]
}

def bfs(visited, graph, src, goal):
  visited.append(src)
queue.append(src)

  while queue:
    m = queue.pop(0)
print (m, end = " ")
if(m==goal):
        break

    for i in graph[m]:
if(i==" " and i!=goal):
        continue       elif i
not in visited:
visited.append(i)
```

```
queue.append(i)
visited = []  queue = []
src=input("Enter
source node: ")
goal=input("Enter
goal node:")

bfs(visited, graph, src,goal)
```

Output:

```
Enter source node: A
Enter goal node:E
A B C D E
```

DFS:

Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node and explores as far as possible along each branch before backtracking

It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children. Because of the recursive nature, stack data structure can be used to implement the DFS algorithm. The process of implementing the DFS is similar to the BFS algorithm.

Algorithm:

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

Step 6: EXIT

Code: def

```
dfs(graph,visited,src,goal):
if src not in visited:
print(src , end = " ")
visited.append(src)
if(src==goal):
      print()
exit(dfs)      for i in
graph[src]:
         dfs(graph,visited,i,goal)

graph = {
   "A" : ["B","C"], "B" : ["D","E"],"C":["G"],"D":[], "E":["F"],"F":[],"G":[] }

visited = [] src=input("Enter
source node: ")
goal=input("Enter goal node: ")
dfs(graph,visited,src,goal)
```

Output:

```
Enter source node: A
Enter goal node: F
A B D E F
```

DFID:

The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found. This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.

This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency. The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Algorithm:

Step1: INPUT: START and GOAL states

Step2: LOCAL VARIABLE: Found

Step3: Initialise d = 1 and FOUND = False

Step4: while (FOUND = False) do
perform DFS from start to depth d.

Step5: if goal state is obtained then FOUND = True else
discard the nodes generated in the search of depth d.

Step6: d = d + 1

Step7: if FOUND = true, then return the depth.

Step8: Stop

Code: from collections import

defaultdict class Graph: def

__init__(self,vertices): self.V

```
= vertices        self.graph =
defaultdict(list)

   def addEdge(self,u,v):
self.graph[u].append(v)

   def DLS(self,src,target,maxDepth):

       if src == target : return True

       if maxDepth <= 0 : return False

       for i in self.graph[src]:
if(self.DLS(i,target,maxDepth-1)):
                   return True
return False

   def IDDFS(self,src, target, maxDepth):
for i in range(maxDepth):            if
(self.DLS(src, target, i)):
               return True
return False

g = Graph (7);
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 3)
g.addEdge(1, 4)
g.addEdge(2, 5)
```

```
g.addEdge(2, 6)

target = int(input("Enter target: ")); maxDepth =
int(input("Enter Maximum Depth: ")); src =
int(input("Enter source"));

if g.IDDFS(src, target, maxDepth) == True:
print ("Target is reachable from source " +
      "within max depth") else
:
   print ("Target is NOT reachable from source " +
      "within max depth")
```

Output:

```
Enter target: 5
Enter Maximum Depth: 2
Enter source 0
0
Target is NOT reachable from source within max depth
```

Conclusion:

Thus, we successfully studied Uninformed Searching Algorithms like BFS, DFS, DFID etc.

Name: Kartik Jolapara **SAPID:** 60004200107

DIV: B/B1

A.I.

Exp3

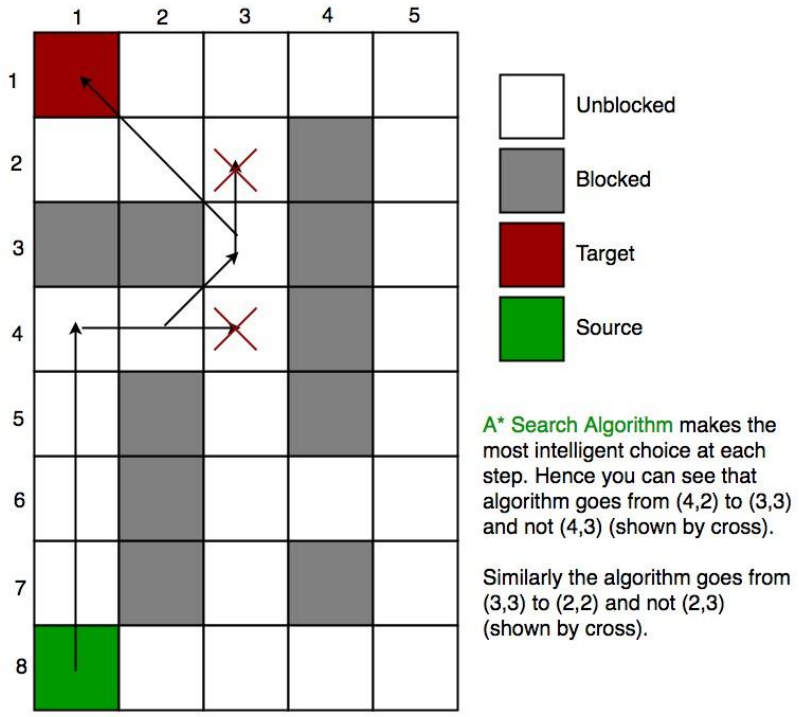
Aim: To study and implement A* Algorithm

Theory:

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

It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem. Another aspect that makes A* so powerful is the use of weighted graphs in its implementation. A weighted graph uses numbers to represent the cost of taking each path or course of action. This means that the algorithms can take the path with the least cost, and find the best route in terms of distance and time.

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



Algorithm:

The open list must be initialized.

Put the starting node on the open list (leave its f at zero). Initialize the closed list.

Follow the steps until the open list is non-empty:

Find the node with the least f on the open list and name it “q”.

Remove Q from the open list.

Produce q's eight descendants and set q as their parent.

For every descendant:

i) If finding a successor is the goal, cease looking ii)Else, calculate g and h

for the successor. successor.g = q.g + the calculated distance between the

successor and the q.

successor.h = the calculated distance between the successor and the goal. We will cover three heuristics to do this: the Diagonal, the Euclidean, and the Manhattan heuristics.

successor.f = successor.g plus successor.h

iii) Skip this successor if a node in the OPEN list with the same location as it but a lower f value than the successor is present.

iv) Skip the successor if there is a node in the CLOSED list with the same position as the successor but a lower f value; otherwise, add the node to the open list end (for loop).

Push Q into the closed list and end the while loop.

Code:

```
def aStarAlgo(start_node, stop_node):
        open_set = set(start_node)
closed_set = set()        g = {}
parents = {}        g[start_node]
= 0
        parents[start_node] = start_node
while len(open_set) > 0:
            n = None            for v in open_set:                if n == None
or g[v] + heuristic(v) < g[n] + heuristic(n):                    n = v


            if n == stop_node or Graph_nodes[n] == None:
pass            else:                for (m, weight) in
get_neighbors(n):                    if m not in open_set and
m not in closed_set:
                        open_set.add(m)
parents[m] = n                        g[m]
= g[n] + weight                    else:

                        if g[m] > g[n] + weight:
g[m] = g[n] + weight
parents[m] = n                            if m in
closed_set:
```

```
closed_set.remove(m)
open_set.add(m)

        if n == None:
print('Path does not exist!')
return None            if n ==
stop_node:              path = []
while parents[n] != n:
path.append(n)                  n =
parents[n]

            path.append(start_node)
path.reverse()              print('Path found:
{}'.format(path))              return path

        closed_set.add(n)
print('Path does not exist!')
return None

def get_neighbors(v):    if
v in Graph_nodes:
return Graph_nodes[v]
else:
    return None

def heuristic(n):
H_dist = {
        'A': 11,
        'B': 6,
        'C': 99,
```

```
            'D': 1,
            'E': 7,
            'G': 0,
        }
        return H_dist[n]
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1),('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],
}
aStarAlgo('A', 'G')
```

Output:

Conclusion:

Thus, we successfully studied and implemented A* Algorithm

Name: Kartik Jolapara **SAPID:** 60004200107

DIV: B/B1

A.I.

Exp4

Aim: To study and implement Hill-Climbing Search

Theory:

Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbour has a higher value. Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance travelled by the salesman.

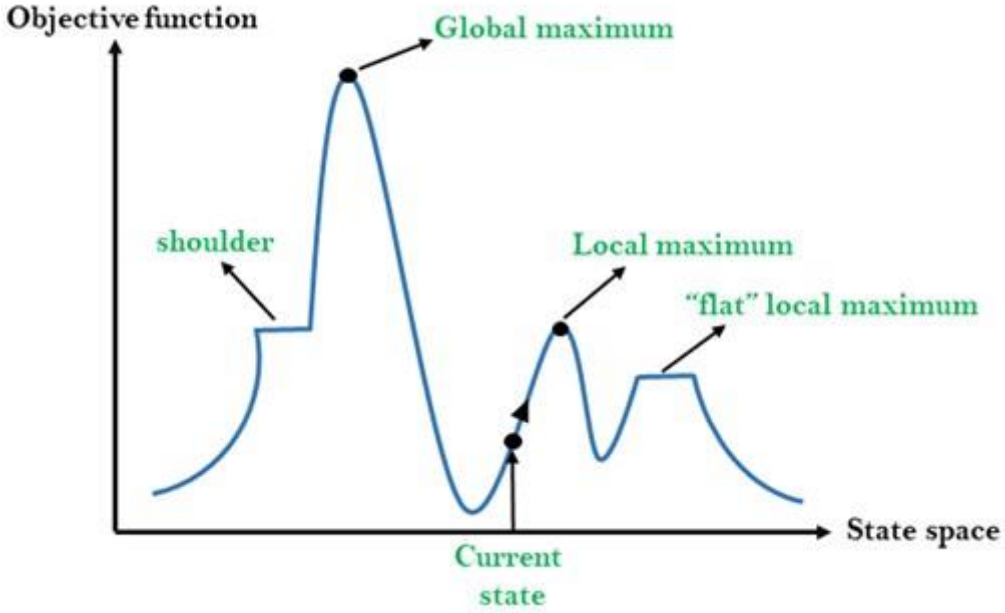
It is also called greedy local search as it only looks to its good immediate neighbour state and not beyond that. A node of hill climbing algorithm has two components which are state and value. Hill Climbing is mostly used when a good heuristic is available. In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

Features:

1. Generate and Test variant: Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.

2. Greedy approach: Hill-climbing algorithm search moves in the direction which optimizes the cost.

3. No backtracking: It does not backtrack the search space, as it does not remember the previous state



Problems:

1. **Local Maximum:** A local maximum is a peak state in the landscape which is better than each of its neighbouring states, but there is another state also present which is higher than the local maximum
2. **Plateau:** A plateau is the flat area of the search space in which all the neighbour states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area
3. **Ridges:** A ridge is a special form of the local maximum. It has an area which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.

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, 100, 700, 50],
```

```
        [100, 0, 330, 1200],
        [700, 330, 0, 400],
        [50, 1200, 400, 0]    ]

    print(hillClimbing(tsp))

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

Output:

```
[3, 2, 1, 0], 880)
```

Conclusion:

Thus we successfully studied and implemented Hill-Climbing Search

Name: Kartik Jolapara **SAPID:** 60004200107

DIV: B/B1

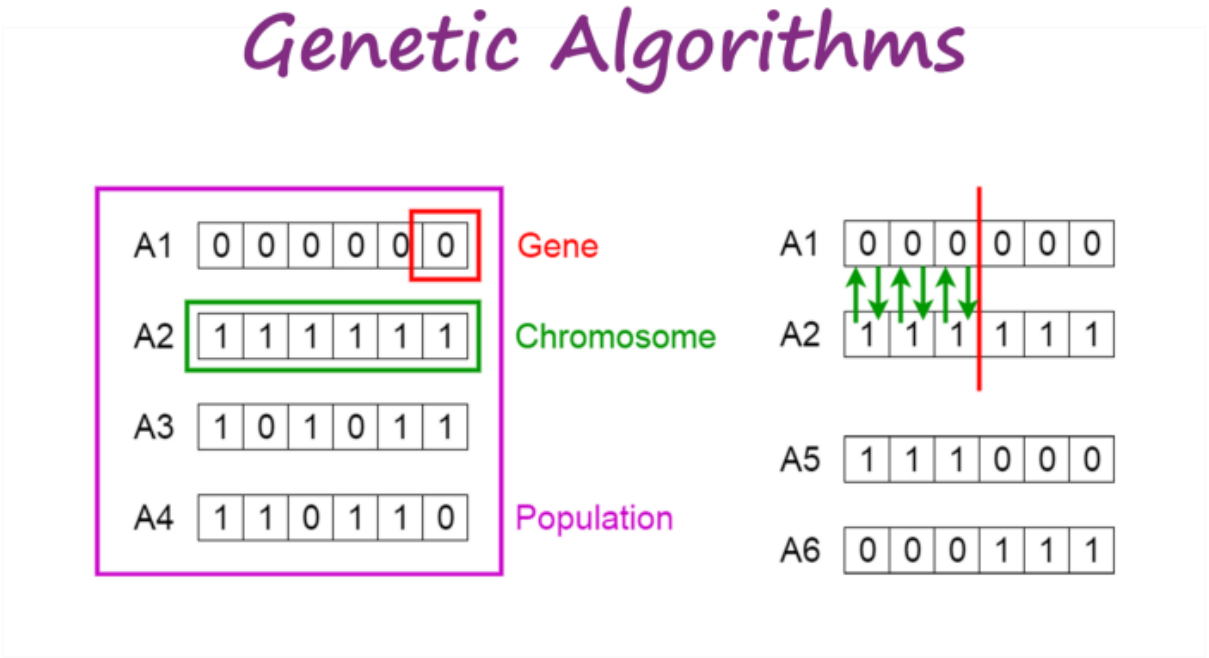
A.I.

Exp5

Aim: To study and implement Genetic Algorithm

Theory:

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



Five phases are considered in a genetic algorithm.

1. Initial population
2. Fitness function
3. Selection
4. Crossover
5. Mutation

Initial Population

The process begins with a set of individuals which is called a Population. Each individual is a solution to the problem you want to solve.An individual is characterized by a set of parameters (variables) known as Genes. Genes are joined into a string to form a Chromosome (solution).

Fitness Function

The fitness function determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. The probability that an individual will be selected for reproduction is based on its fitness score.

Selection

The idea of selection phase is to select the fittest individuals and let them pass their genes to the next generation.Two pairs of individuals (parents) are selected based on their fitness scores. Individuals with high fitness have more chance to be selected for reproduction.

Crossover

Crossover is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a crossover point is chosen at random from within the genes.

Mutation

In certain new offspring formed, some of their genes can be subjected to a mutation with a low random probability. This implies that some of the bits in the bit string can be flipped.

Code:

```
from numpy.random import randint

from numpy.random import rand

import math def crossover(parent1,

parent2, r_cross):

    child1, child2 = parent1.copy(),

parent2.copy()     r = rand()     point = 0     if r >

r_cross:

        point = randint(1, len(parent1) - 2)

child1 = parent1[:point] + parent2[point:]

child2 = parent2[:point] + parent1[point:]

return child1, child2, point
```

```
def mutate(chromosome, r_mut):     for i in
range(len(chromosome)):         if rand() <
r_mut:              chromosome[i] = 1 -
chromosome[i]     return chromosome


def         bin_to_dec(bin):
decimal  =  0        for  i  in
range(len(bin)):
        decimal += bin[i] * pow(2, 4 - i)
return decimal


def dec_to_bin(dec):
binaryVal = []
while dec > 0:
        binaryVal.append(dec % 2)
dec = math.floor(dec / 2)     for _ in
range(5 - len(binaryVal)):
binaryVal.append(0)     binaryVal =
binaryVal[::-1]
    return binaryVal
```

```
def fitness_function(x):
    return pow(x, 2) def genetic_algorithm(iterations,
population_size, r_cross, r_mut):
    input = [randint(0, 32) for _ in
range(population_size)]     pop = [dec_to_bin(i) for i in
input]     for generation in range(iterations):
        print(f"\nGeneration : {generation+1}", end="\n\n")        decimal =
[bin_to_dec(i) for i in pop]        fitness_score = [fitness_function(i) for i
in decimal]        f_by_sum = [            fitness_score[i] /
sum(fitness_score) for i in range(population_size)
        ]
        exp_cnt = [            fitness_score[i] / (sum(fitness_score)
/ population_size)            for i in range(population_size)
        ]
        act_cnt = [round(exp_cnt[i]) for i in range(population_size)]
        print(
            "SELECTION\n\nInitial Population\tDecimal Value\tFitness
Score\tFi/Sum\tExpected count\tActual Count"
        )
        for i in range(population_size):
```

```
            print(
pop[i],                 "\t",
decimal[i],
"\t\t",
fitness_score[i],
                "\t\t",
                round(f_by_sum[i], 2),
                "\t\t",
round(exp_cnt[i], 2),
                "\t\t",
act_cnt[i],
            )
        print("Sum : ", sum(fitness_score))         print("Average
: ", sum(fitness_score) / population_size)
print("Maximum : ", max(fitness_score), end="\n")
max_count = max(act_cnt)         min_count = min(act_cnt)
max_count_index = 0         for i in range(population_size):
if max_count == act_cnt[i]:                 max_count_index = i
break         for i in range(population_size):             if
min_count == act_cnt[i]:                 pop[i] =
```

```
pop[max_count_index]        crossover_children = list()
crossover_point = list()        for i in range(0,
population_size, 2):
            child1, child2, point_of_crossover = crossover(pop[i], pop[i +
1], r_cross)            crossover_children.append(child1)
crossover_children.append(child2)
crossover_point.append(point_of_crossover)
crossover_point.append(point_of_crossover)
        print(
            "\nCROSS OVER\n\nPopulation\t\tMate\t Crossover Point\t Crossover
Population"
        )
        for i in range(population_size):
if (i + 1) % 2 == 1:
                mate = i +
2            else:
mate = i            print(
pop[i],                "\t",
mate,
                "\t",
crossover_point[i],
```

```
            "\t\t\t",
crossover_children[i],
        )
    mutation_children = list()
for i in range(population_size):
child = crossover_children[i]
mutation_children.append(mutate(c
hild, r_mut))      new_population
= list()      new_fitness_score =
list()      for i in
mutation_children:
        new_population.append(bin_to_dec(i))
for i in new_population:
        new_fitness_score.append(fitness_function(i))
    print("\nMUTATION\n\nMutation population\t New Population\t Fitness
Score")      for i in range(population_size):
        print(
mutation_children[i],
            "\t",
            new_population[i],
```

```
            "\t\t",
            new_fitness_score[i],
        )
    print("Sum : ", sum(new_fitness_score))
print("Maximum : ", max(new_fitness_score))
pop = mutation_children
genetic_algorithm(iterations=2, population_size=4,
r_cross=0.5, r_mut=0.05)
```

Output:

```
Maximum :  729

CROSS OVER

Population              Mate     Crossover Point         Crossover Population
[1, 0, 1, 0, 0]          2       1                       [1, 1, 0, 1, 1]
[1, 1, 0, 1, 1]          1       1                       [1, 0, 1, 0, 0]
[1, 1, 0, 1, 1]          4       0                       [1, 1, 0, 1, 1]
[1, 1, 0, 1, 1]          3       0                       [1, 1, 0, 1, 1]

MUTATION

Mutation population      New Population  Fitness Score
[1, 1, 0, 1, 0]          26              676
[1, 0, 1, 0, 0]          20              400
[1, 1, 0, 1, 1]          27              729
[1, 1, 0, 1, 1]          27              729
Sum :  2534
Maximum :  729

Generation : 2

SELECTION

Initial Population      Decimal Value   Fitness Score   Fi/Sum  Expected count  Actual Count
[1, 1, 0, 1, 0]          26              676             0.27            1.07            1
[1, 0, 1, 0, 0]          20              400             0.16            0.63            1
[1, 1, 0, 1, 1]          27              729             0.29            1.15            1
[1, 1, 0, 1, 1]          27              729             0.29            1.15            1
Sum :  2534
Average :  633.5
Maximum :  729

CROSS OVER

Population              Mate     Crossover Point         Crossover Population
[1, 1, 0, 1, 0]          2       1                       [1, 1, 0, 1, 0]
[1, 1, 0, 1, 0]          1       1                       [1, 1, 0, 1, 0]
[1, 1, 0, 1, 0]          4       1                       [1, 1, 0, 1, 0]
[1, 1, 0, 1, 0]          3       1                       [1, 1, 0, 1, 0]

MUTATION

Mutation population      New Population  Fitness Score
[1, 0, 0, 0, 0]          16              256
[1, 1, 0, 1, 0]          26              676
[1, 1, 0, 1, 0]          26              676
[1, 1, 0, 1, 0]          26              676
Sum :  2284
Maximum :  676
```

Conclusion:

Thus we successfully studied and applied Genetic Algorithm

Name: Kartik Jolapara **SAPID:** 60004200107

DIV: B/B1

AI

Exp6

Aim: Implementation of Wumpus World Program in Prolog.

Theory:

Unlike many other programming languages, Prolog is intended primarily as a declarative programming language. In prolog, logic is expressed as relations (called Facts and Rules). The core heart of the prolog lies in the logic being applied.

Formulation or Computation is carried out by running a query over these relations. Prolog features are 'Logical variable', which means that they behave like uniform data structures, a backtracking strategy to search for proofs, a pattern-matching facility, mathematical variables, and input and out are interchangeable. Prolog is a declarative language that means we can specify what problem we want to solve rather than how to solve it. Prolog is used in some areas like databases, natural language processing, and artificial intelligence, but it is pretty useless in some areas like numerical algorithms or instance graphics.

Key Features of Prolog:

1. Unification: The basic idea is, can the given terms be made to represent the same structure.
2. Backtracking: When a task fails, prolog traces backward and tries to satisfy the previous task.
3. Recursion: Recursion is the basis for any search in the program.

Prolog provides an easy to build a database. Doesn't need a lot of programming effort. Pattern matching is easy. Search is recursion based. It has built-in list handling. Makes it easier to play with any algorithm involving lists. LISP (another logic programming language) dominates over prolog with respect to I/O features. Sometimes input and output is not easy.

Applications:

- Specification Language
- Robot Planning
- Natural language understanding
- Machine Learning
- Problem Solving
- Intelligent Database retrieval
- Expert System
- Automated Reasoning

Code:

```
/* Facts */ male(jack).

male(oliver).

male(ali).

male(james).

male(simon).

male(harry).

male(bhupesh).

male(ram).

female(helen).

female(sophie).

female(jess).
```

female(lily).

female(sita).

parent_of(jack,jess).

parent_of(jack,lily).

parent_of(helen, jess).

parent_of(helen, lily).

parent_of(oliver,james).

parent_of(sophie, james).

parent_of(jess, simon).

parent_of(ali, simon). parent_of(lily,

harry). parent_of(james, harry).

parent_of(jack,bhupesh).

parent_of(helen,bhupesh).

parent_of(ram,helen).

parent_of(sita,helen).

parent_of(ram,sophie).

parent_of(sita,sophie).

/* Rules */ father_of(X,Y):-

male(X), parent_of(X,Y).

mother_of(X,Y):- female(X),

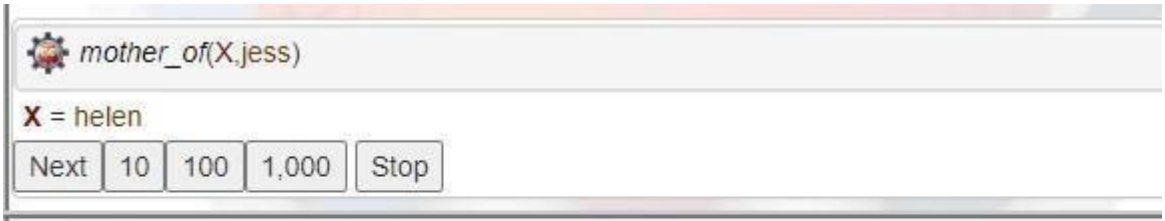
parent_of(X,Y).

grandfather_of(X,Y):- male(X), parent_of(X,Z), parent_of(Z,Y). grandmother_of(X,Y):- female(X), parent_of(X,Z), parent_of(Z,Y). sister_of(X,Y):- %(X,Y or Y,X)% female(X),

```
father_of(F,     Y), father_of(F,X),X
\=       Y.
sister_of(X,Y):-     female(X),
mother_of(M,                 Y),
mother_of(M,X),X    \=    Y.
aunt_of(X,Y):-       female(X),
parent_of(Z,Y),
sister_of(Z,X),!.
brother_of(X,Y):-  %(X,Y   or
Y,X)%                  male(X),
father_of(F,                 Y),
father_of(F,X),X     \=      Y.
brother_of(X,Y):-    male(X),
mother_of(M,                 Y),
mother_of(M,X),X    \=    Y.
uncle_of(X,Y):-
parent_of(Z,Y),
brother_of(Z,X).
ancestor_of(X,Y):-
parent_of(X,Y).
ancestor_of(X,Y):-
parent_of(X,Z),
ancestor_of(Z,Y).
```

Output:


father_of(X,jess)
X = jack
Next
10
100
1,000
Stop
?-
father_of(X,jess)




```
?- mother_of(X,jess)
```


```
?- grandfather_of(X,simon)
```


```
?- sister_of(X,jess)
```

Conclusion: Thus, we successfully implemented family tree in prolog.

Name: Kartik Jolapara **SAPID:** 60004200107

DIV: B/B1

AI

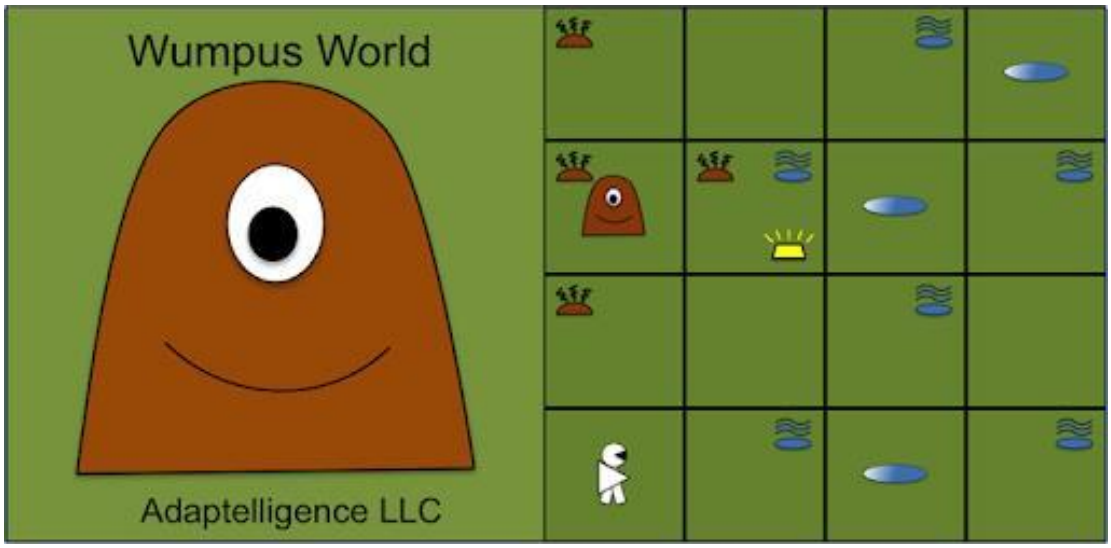
Exp7

Aim: Implementation of Wumpus World Program in Prolog.

Theory:

The Wumpus World's agent is an example of a knowledge-based agent that represents Knowledge representation, reasoning, and planning. Knowledge Based agent links general knowledge with current precepts to infer hidden characters of the current state before selecting actions. Its necessity is vital in partially observable environments.

Wumpus world is a cave with 16 rooms (4×4). Each room is connected to others through walkways (no rooms are connected diagonally). The knowledge-based agent starts from Room [1, 1]. The cave has – some pits, a treasure, and a beast named Wumpus. The Wumpus cannot move but eats the one who enters its room. If the agent enters the pit, it gets stuck there. The goal of the agent is to take the treasure and come out of the cave. The agent is rewarded, when the goal conditions are met. The agent is penalized, when it falls into a pit or is eaten by the
Wumpus.
Some elements support the agent to explore the cave, like -The Wumpus's adjacent rooms are stench. -The agent is given one arrow which it can use to kill the Wumpus when facing it (Wumpus screams when it is killed). – The adjacent rooms of the room with pits are filled with breeze. -The treasure room is always glittery.



PEAS for Wumpus World:

PEAS represents Performance Measures, Environment, Actuators, and Sensors. The PEAS description helps in grouping the agents. PEAS Description for the Wumpus World problem:

1. Performance measures:
    1. Agent gets the gold and returns back safe = +1000 points
    2. Agent dies = -1000 points
    3. Each move of the agent = -1 point
    4. Agent uses the arrow = -10 points 2. Environment:
    1. A cave with 16(4×4) rooms
    2. Rooms adjacent (not diagonally) to the Wumpus are stinking
    3. Rooms adjacent (not diagonally) to the pit are breezy
    4. The room with the gold glitters
    5. Agent's initial position – Room [1, 1] and facing right side
    6. Location of Wumpus, gold, and 3 pits can be anywhere, except in Room
    [1, 1].
3. Actuators:
    1. Devices that allow the agent to perform the following actions in the environment.
    2. Move forward
    3. Turn right
    4. Turn left
    5. Shoot 6. Grab-Release
4. Sensors:

1. Devices that help the agent in sensing the following from the environment.
2. Breeze
3. Stench
4. Glitter
5. Scream (When the Wumpus is killed)
6. Bump (when the agent hits a wall)

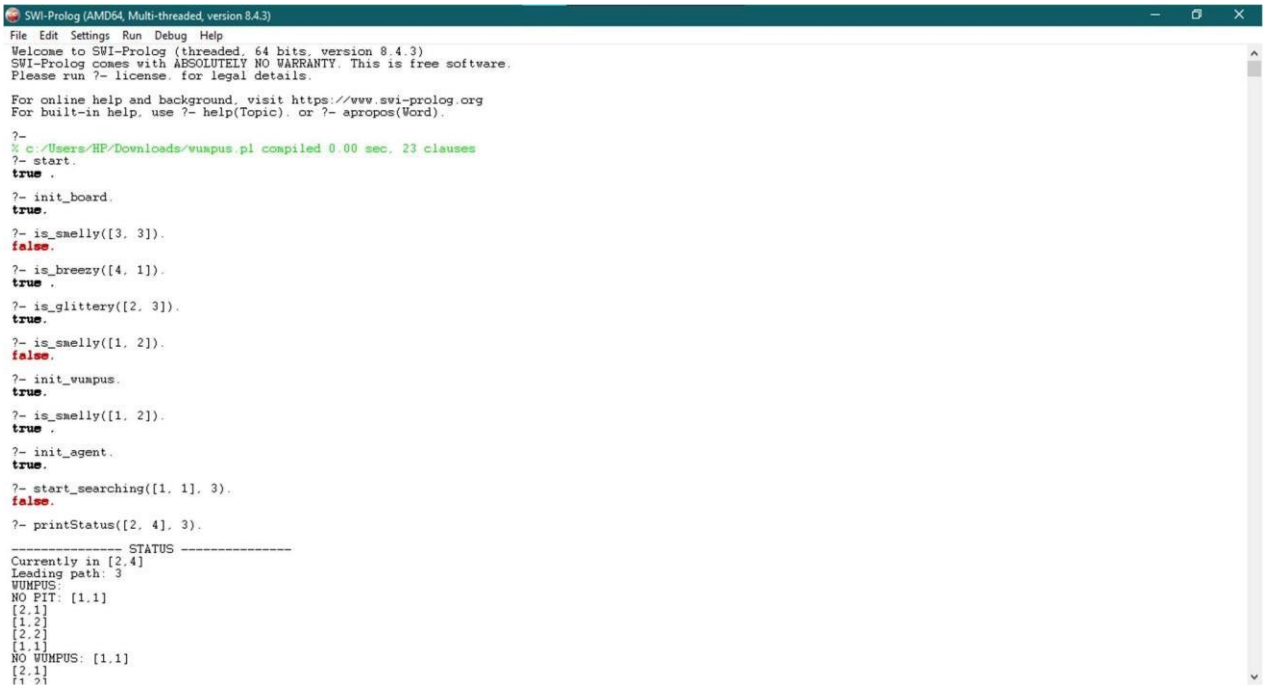
Code:

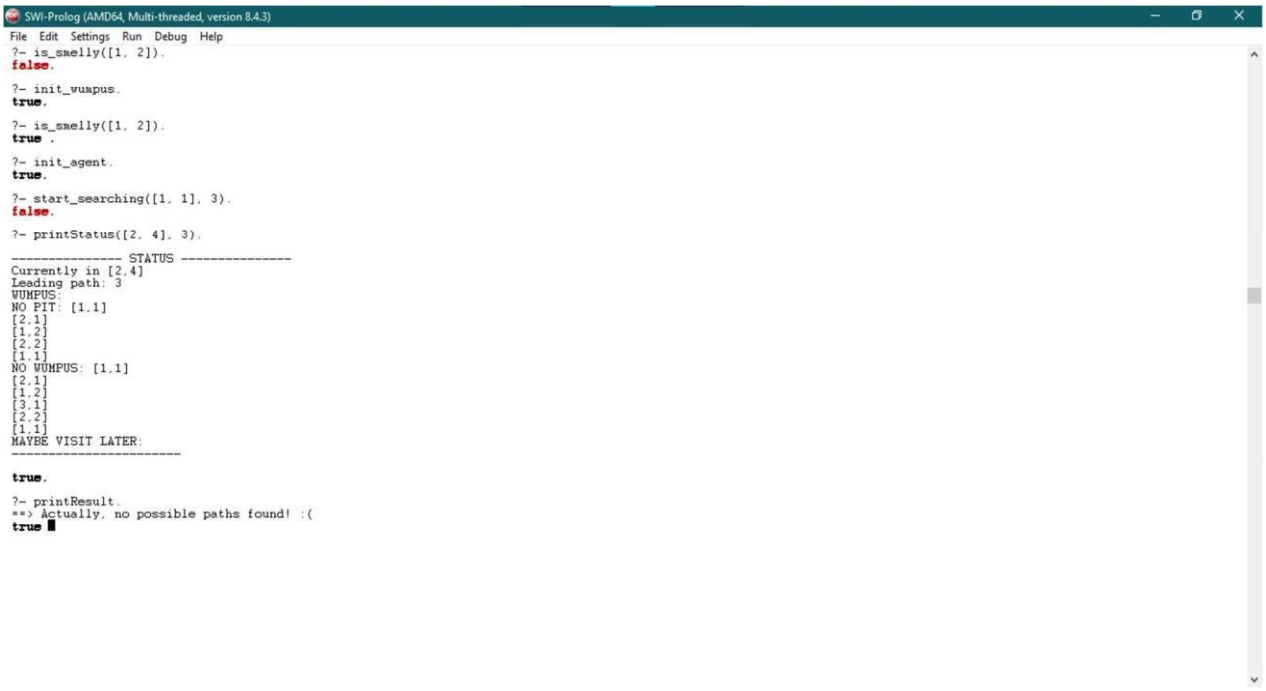
```
dynamic([
world_size/1,
position/2,
wumpus/1, noPit/1,
noWumpus/1, maybeVisitLater/2,
goldPath/1
]).
start:- retractall(wumpus(_)), retractall(noPit(_)),
retractall(noWumpus(_)),
retractall(maybeVisitLater(_,_)),
retractall(goldPath(_)), init_board, init_agent,
init_wumpus, start_searching([1, 1], []),
maybeVisitLater(PausedCell, LeadingPath),
retract(maybeVisitLater(PausedCell, _)),
start_searching(PausedCell, LeadingPath).
init_board:- retractall(world_size(_)),
assert(world_size([5, 5])),
retractall(position(_, _)),
assert(position(gold, [2, 3])),
assert(position(pit, [3, 1])),
assert(position(pit, [5, 1])),
assert(position(pit, [3, 3])),
assert(position(pit, [4, 4])),
assert(position(pit, [2, 5])), assert(noPit([1,
1])).
init_agent:- assert(position(agent, [1, 1])). init_wumpus:-
assert(position(wumpus, [1, 3])), assert(noWumpus([1, 1])).
valid_position([X, Y]):- X>0, Y>0, world_size([P, Q]), X@=<P, Y@=<Q. adjacent([X,
Y], Z):- Left is X-1, valid_position([Left, Y]), Z=[Left, Y]. adjacent([X, Y], Z):- Right is
X+1, valid_position([Right, Y]), Z=[Right, Y]. adjacent([X, Y], Z):- Above is Y+1,
```

```
valid_position([X, Above]), Z=[X, Above]. adjacent([X, Y], Z):- Below is Y-1,
valid_position([X, Below]), Z=[X, Below].
is_smelly([X, Y]):-   position(wumpus, Z), \+
noWumpus(Z),   adjacent([X, Y], Z). is_breezy([X, Y]):-
adjacent([X, Y], Z), position(pit, Z). is_glittery([X, Y]):-
position(gold, Z), Z==[X, Y].
moreThanOneWumpus:-   wumpus(X),
wumpus(Y), X\=Y.
killWumpusIfPossible(AgentCell):-
  wumpus([Xw, Yw]), \+ moreThanOneWumpus,
  AgentCell=[Xa, Ya],   (Xw==Xa; Yw==Ya),   assert(noWumpus([Xw, Yw])),
format('~nAgent confirmed Wumpus cell to be ~w and shot an arrow from cell  ~w.~nThe
WUMPUS has been killed!~n', [[Xw, Yw], AgentCell]),   retractall(wumpus(_)).
start_searching(Cell,      LeadingPath):-   is_glittery(Cell),
append(LeadingPath, [Cell], CurrentPath),
  \+ goldPath(CurrentPath), assert(goldPath(CurrentPath)).
start_searching(Cell, _):-
is_breezy(Cell).
start_searching(Cell, _):-   \+
is_breezy(Cell),
adjacent(Cell, X),
  \+ noPit(X), assert(noPit(X)).  start_searching(Cell,
            _):-
is_smelly(Cell),   adjacent(Cell,
X),
  \+ noWumpus(X), assert(wumpus(X)).
start_searching(Cell, _):-   \+
is_smelly(Cell),   adjacent(Cell,
X),
  \+ noWumpus(X), assert(noWumpus(X)),   wumpus(Y),
X==Y, retract(wumpus(Y)). start_searching(CurrentCell,
LeadingPath):-        (killWumpusIfPossible(CurrentCell);
format('')),          append(LeadingPath,    [CurrentCell],
CurrentPath),
  \+ is_glittery(CurrentCell),   adjacent(CurrentCell, X), \+
member(X, LeadingPath),
  (( noWumpus(X), noPit(X)) -> write('');
    (\+ maybeVisitLater(CurrentCell, _) -> assert(maybeVisitLater(CurrentCell, LeadingPath)); write(''))
  ),
  noWumpus(X), noPit(X),   start_searching(X, CurrentPath).
printResult:-
  \+ goldPath(_), write("==> Actually, no possible paths found! :("). printResult:-
goldPath(_), !, format('The following paths to the Gold are found: ~n'),   forall(goldPath(X),
writeln(X)).  printStatus(Cell,  LeadingPath):-      format('~n---------- STATUS ----------
```

~nCurrently in ~w~nLeading path: ~w~n', [Cell, LeadingPath]), write('WUMPUS: '), forall(wumpus(X), writeln(X)),nl, write('NO PIT: '), forall(noPit(Y), writeln(Y)), write('NO WUMPUS: '), forall(noWumpus(Z), writeln(Z)), write('MAYBE VISIT LATER: '), forall(maybeVisitLater(M, _), writeln(M)), format('~n----------~n~n').

Output:





Conclusion:

Thus, we successfully implemented Wumpus World in SWI-Prolog.

Name: Kartik Jolapara **SAPID:** 60004200107

DIV: B/B1

AI

Exp8

Aim: A Study on Planning Problem in AI.

Theory:

Artificial Intelligence is a critical technology in the future. Whether it is intelligent robots or self-driving cars or smart cities, they will all use different aspects of Artificial Intelligence!!! But to create any such AI project, **Planning** is very important. So much so that Planning is a critical part of Artificial Intelligence which deals with the actions and domains of a particular problem. Planning is considered as the reasoning side of acting.
Everything we humans do is with a certain goal in mind and all our actions are oriented towards achieving our goal. In a similar fashion, planning is also done for Artificial Intelligence. For example, reaching a particular destination requires planning. Finding the best route is not the only requirement in planning, but the actions to be done at a particular time and why they are done is also very important. That is why planning is considered as the reasoning side of acting. In other words, planning is all about deciding the actions to be performed by the Artificial Intelligence system and the functioning of the system on its own in domain independent situations.

For any planning system, we need the domain description, action specification, and goal description. A plan is assumed to be a sequence of actions and each action has its own set of preconditions to be satisfied before performing the action and also some effects which can be positive or negative.

So, we have Forward State Space Planning (FSSP) and Backward State Space Planning (BSSP) at the basic level.

Forward State Space Planning (FSSP)

FSSP behaves in a similar fashion like forward state space search. It says that given a start state S in any domain, we perform certain actions required and acquire a new state S' (which includes some new conditions as well) which is called progress and this proceeds until we reach the goal state. The actions have to be applicable in this case. Disadvantage: Large branching factor

Advantage: Algorithm is Sound

Backward State Space Planning (BSSP)

BSSP behaves in a similar fashion like backward state space search. In this, we move from the goal state g towards sub-goal g' that is finding the previous action to be done to achieve that respective goal. This process is called regression (moving back to the previous goal or sub-goal). These sub-goals have to be checked for consistency as well. The actions have to be relevant in this case.

Disadvantage: Not a sound algorithm (sometimes inconsistency can be found)

Advantage: Small branching factor (very small compared to FSSP)

Example:

Planning in artificial intelligence is about decision-making actions performed by robots or computer programs to achieve a specific goal.

Execution of the plan is about choosing a sequence of tasks with a high probability of accomplishing a specific task.

Block-world planning problem

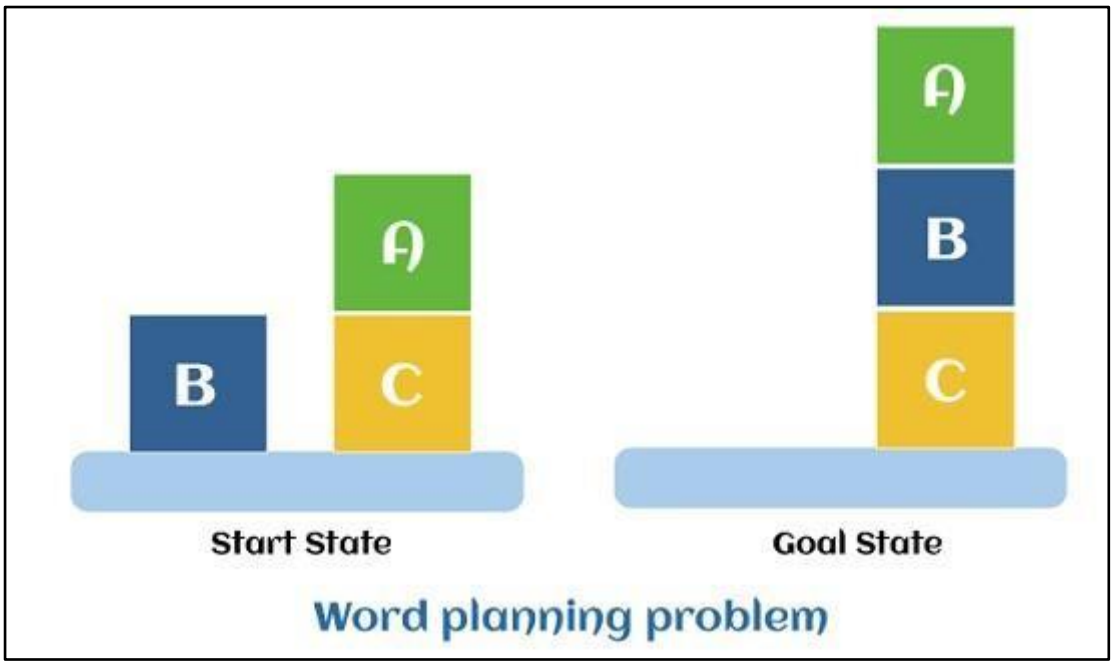
The block-world problem is known as the Sussmann anomaly.

The non-interlaced planners of the early 1970s were unable to solve this problem. Therefore it is considered odd.

When two sub-goals, G1 and G2, are given, a non-interleaved planner either produces a plan for G1 that is combined with a plan for G2 or vice versa.

In the block-world problem, three blocks labeled 'A', 'B', and 'C' are allowed to rest on a flat surface. The given condition is that only one block can be moved at a time to achieve the target.

The start position and target position are shown in the following diagram.



Components of the planning system

The plan includes the following important steps:

Choose the best rule to apply the next rule based on the best available guess. o Apply the chosen rule to calculate the new problem condition.

Find out when a solution has been found. o Detect dead ends so they can be discarded and direct system effort in more useful directions.

Find out when a near-perfect solution is found.

Target stack plan o It is one of the most important planning algorithms used by STRIPS. o Stacks are used in algorithms to capture the action and complete the target. A knowledge base is used to hold the current situation and actions.

A target stack is similar to a node in a search tree, where branches are created with a choice of action.

The important steps of the algorithm are mentioned below:

Start by pushing the original target onto the stack. Repeat this until the pile is empty. If the stack top is a mixed target, push its unsatisfied sub-targets onto the stack.

If the stack top is a single unsatisfied target, replace it with action and push the action precondition to the stack to satisfy the condition.

iii. If the stack top is an action, pop it off the stack, execute it and replace the knowledge base with the action's effect.

If the stack top is a satisfactory target, pop it off the stack.

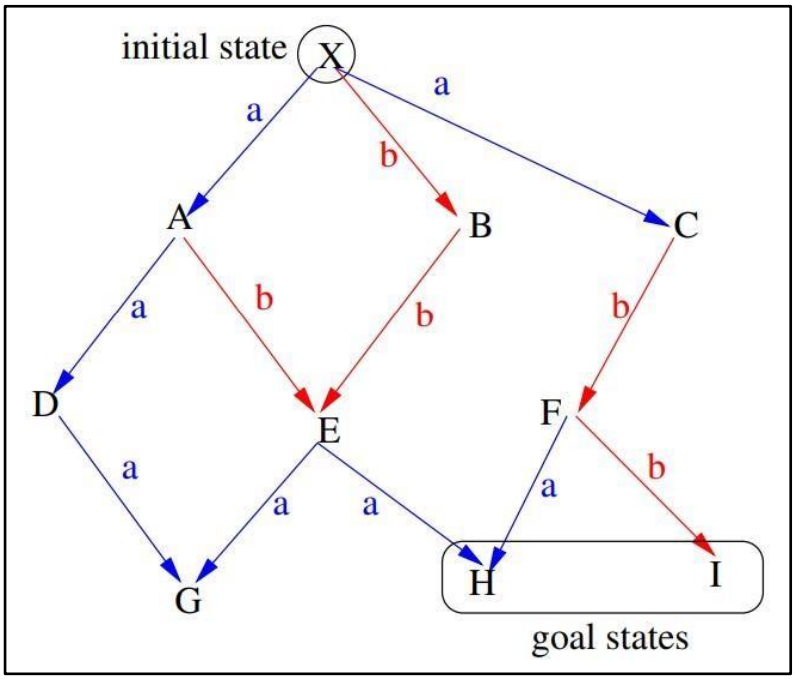
➢ Planning Through State Space Search

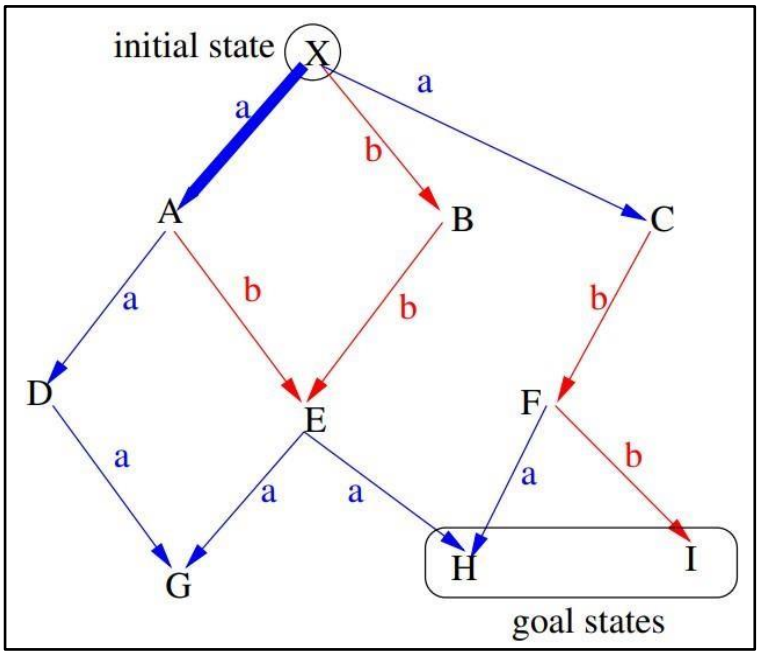
We can view planning problems as searching for goal nodes in a large labeled graph (transition system)

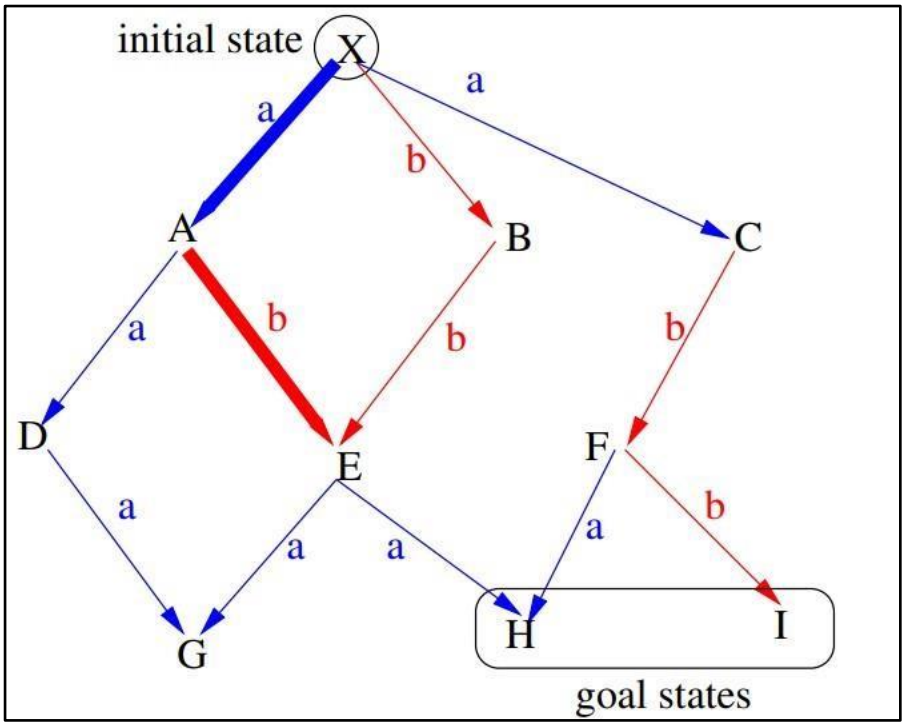
Nodes are defined by the value assignment to the fluents = states

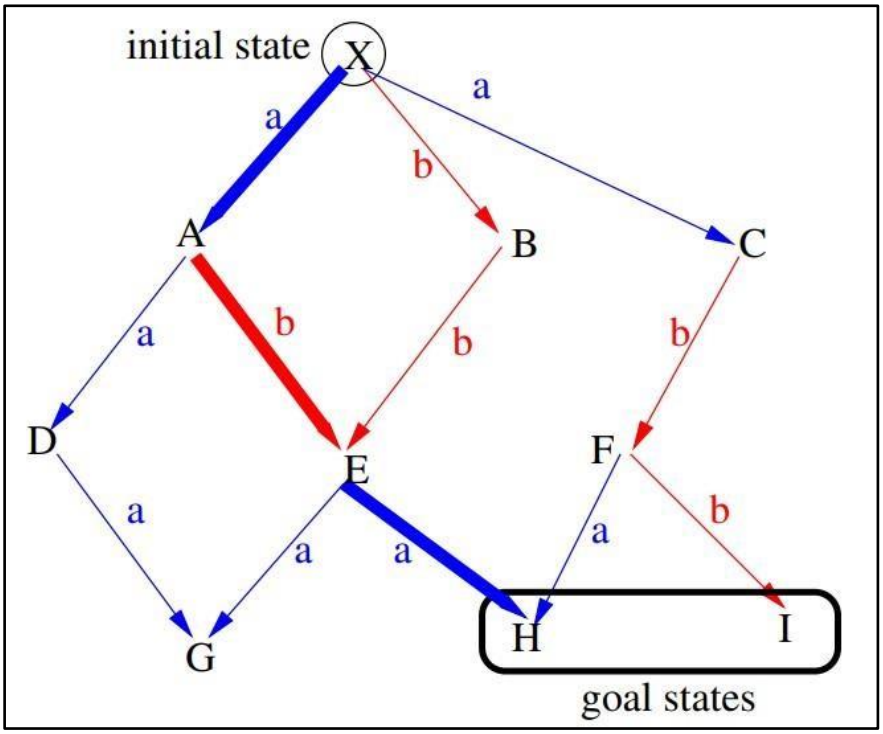
Labeled edges are defined by actions that change the appropriate fluents

Use graph search techniques to find a (shortest) path in this graph! Note: The graph can become huge: 50 Boolean variables lead to 250 = 1015 states Create the transition system on the fly and visit only the parts that are necessary


initial state
X
a
a
b
A
B
C
a
b
b
b
D
E
F
a
a
a
b
a
G
H
I
goal states


initial state
X
a
a
b
A
B
C
a
b
b
b
D
E
F
a
a
a
b
a
G
H
I
goal states





Let’s Consider a planning problem and try to solve it with Forward state space search planning and Backward state space search planning.

Forward state space search planning

Search through transition system starting at initial state

1. Initialize partial plan $\Delta := \langle\,\rangle$ and start at the unique initial state $\mathbf{I}$ and make it the current state S
2. Test whether we have reached a goal state already: $\mathbf{G} \subseteq S$? If so, return plan Δ.
3. Select one applicable action o_i non-deterministically and
   - compute successor state $S := App(S, o_i)$,
   - extend plan $\Delta := \langle \Delta, o_i \rangle$, and continue with step 2.

Instead of non-deterministic choice use some search strategy.
Progression planning can be easily extended to more expressive planning languages

Problem:

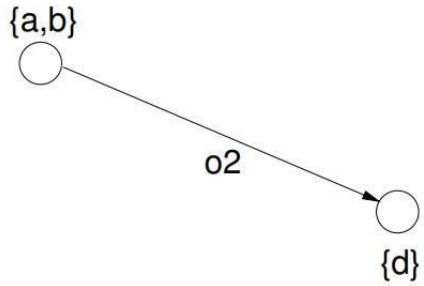
$$
\begin{aligned}
\mathcal{S} &= \{a, b, c, d\}, \\
\mathbf{O} &= \{\ o_1 = \langle \emptyset, \{a, b\}, \{\neg b, c\} \rangle, \\
&\quad\ o_2 = \langle \emptyset, \{a, b\}, \{\neg a, \neg b, d\} \rangle, \\
&\quad\ o_3 = \langle \emptyset, \{c\}, \{b, d\} \rangle, \\
\mathbf{I} &= \{a, b\} \\
\mathbf{G} &= \{b, d\}
\end{aligned}
$$

{a,b}

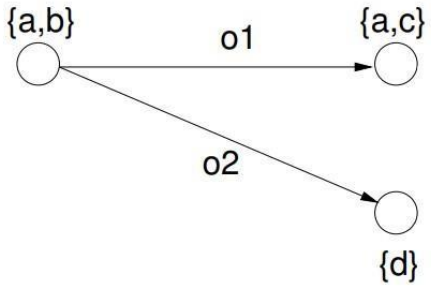
Solution:

Backward state space search planning Problem:

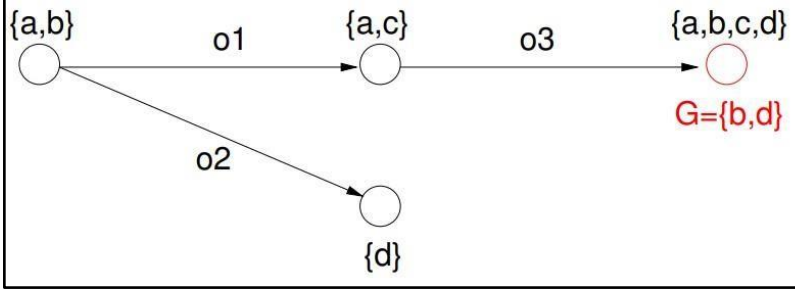
$$
\begin{aligned}
\mathcal{S} &= \{a, b, c, d\}, \\
\mathbf{O} &= \{\ o_1 = \langle \emptyset, \{a, b\}, \{\neg b, c\} \rangle, \\
&\quad\ o_2 = \langle \emptyset, \{a, b\}, \{\neg a, \neg b, d\} \rangle, \\
&\quad\ o_3 = \langle \emptyset, \{c\}, \{b, d\} \rangle, \\
\mathbf{I} &= \{a, b\} \\
\mathbf{G} &= \{b, d\}
\end{aligned}
$$



$$
\begin{aligned}
\mathcal{S} &= \{a, b, c, d\}, \\
\mathbf{O} &= \{\ o_1 = \langle \emptyset, \{a, b\}, \{\neg b, c\} \rangle, \\
&\quad\ o_2 = \langle \emptyset, \{a, b\}, \{\neg a, \neg b, d\} \rangle, \\
&\quad\ o_3 = \langle \emptyset, \{c\}, \{b, d\} \rangle, \\
\mathbf{I} &= \{a, b\} \\
\mathbf{G} &= \{b, d\}
\end{aligned}
$$



$$
\begin{aligned}
\mathcal{S} &= \{a, b, c, d\}, \\
\mathbf{O} &= \{\ o_1 = \langle \emptyset, \{a, b\}, \{\neg b, c\} \rangle, \\
&\quad\ o_2 = \langle \emptyset, \{a, b\}, \{\neg a, \neg b, d\} \rangle, \\
&\quad\ o_3 = \langle \emptyset, \{c\}, \{b, d\} \rangle, \\
\mathbf{I} &= \{a, b\} \\
\mathbf{G} &= \{b, d\}
\end{aligned}
$$



Search through transition system starting at goal states. Consider sets of states, which are described by the atoms that are necessarily true in them

1. Initialize partial plan $\Delta := \langle\,\rangle$ and set $\mathbf{S} := \mathbf{G}$
2. Test whether we have reached the unique initial state already: $\mathbf{I} \supseteq \mathbf{S}$? If so, return plan Δ.
3. Select one action o_i non-deterministically which does not make (sub-)goals false ($\mathbf{S} \cap \neg \mathit{eff}^-(o_i) = \emptyset$) and
   - compute the regression of the description $\mathbf{S}$ through o_i:

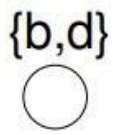
$$\mathbf{S} := \mathbf{S} - \mathit{eff}^+(o_i) \cup \mathit{pre}(o_i)$$

   - extend plan $\Delta := \langle o_i, \Delta \rangle$, and continue with step 2.

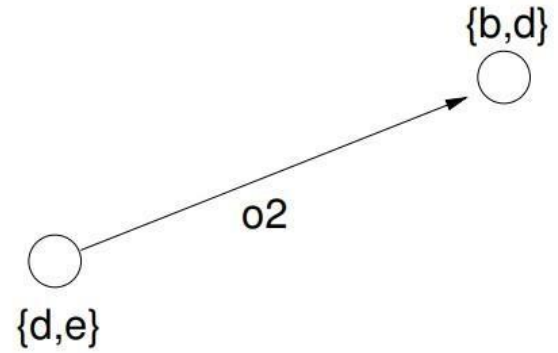
Instead of non-deterministic choice use some search strategy
Regression becomes much more complicated, if e.g. conditional effects are allowed. Then the result of a regression can be a general Boolean formula

Solution:

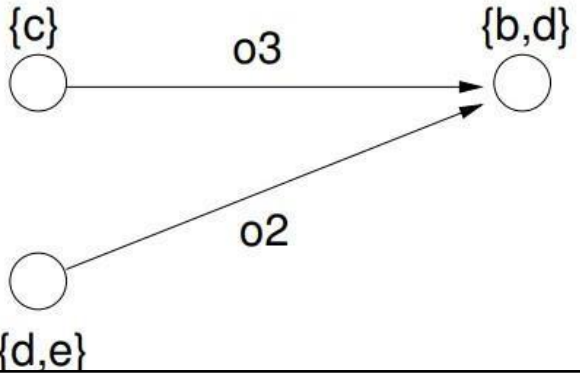
$$
\begin{aligned}
\mathcal{S} &= \{a, b, c, d, e\}, \\
\mathbf{O} &= \{\ o_1 = \langle \emptyset, \{b\}, \{\neg b, c\} \rangle, \\
&\quad\ \ o_2 = \langle \emptyset, \{e\}, \{b\} \rangle, \\
&\quad\ \ o_3 = \langle \emptyset, \{c\}, \{b, d, \neg e\} \rangle, \\
\mathbf{I} &= \{a, b\} \\
\mathbf{G} &= \{b, d\}
\end{aligned}
$$



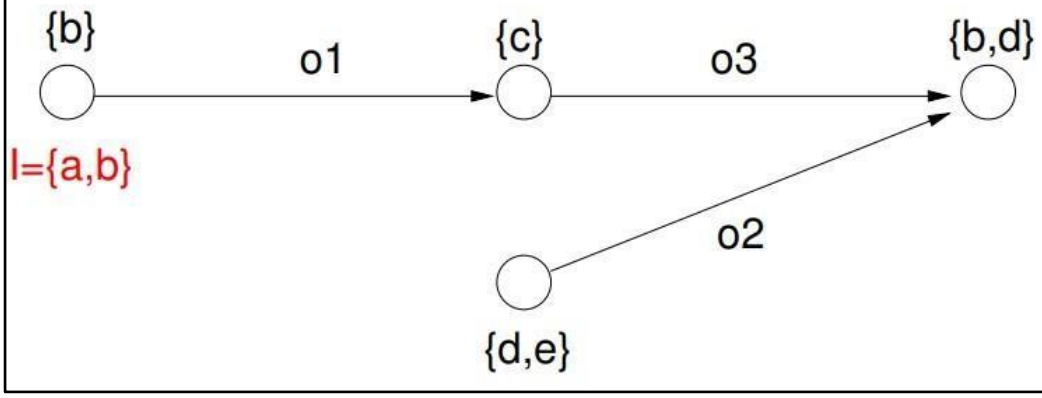
$$
\begin{aligned}
\mathcal{S} &= \{a, b, c, d, e\}, \\
\mathbf{O} &= \{\ o_1 = \langle \emptyset, \{b\}, \{\neg b, c\} \rangle, \\
&\quad\ \ o_2 = \langle \emptyset, \{e\}, \{b\} \rangle, \\
&\quad\ \ o_3 = \langle \emptyset, \{c\}, \{b, d, \neg e\} \rangle, \\
\mathbf{I} &= \{a, b\} \\
\mathbf{G} &= \{b, d\}
\end{aligned}
$$



$$
\begin{aligned}
\mathcal{S} &= \{a, b, c, d, e\}, \\
\mathbf{O} &= \{\ o_1 = \langle \emptyset, \{b\}, \{\neg b, c\} \rangle, \\
&\quad\ \ o_2 = \langle \emptyset, \{e\}, \{b\} \rangle, \\
&\quad\ \ o_3 = \langle \emptyset, \{c\}, \{b, d, \neg e\} \rangle, \\
\mathbf{I} &= \{a, b\} \\
\mathbf{G} &= \{b, d\}
\end{aligned}
$$



$$
\begin{aligned}
\mathcal{S} &= \{a, b, c, d, e\}, \\
\mathbf{O} &= \{\ o_1 = \langle \emptyset, \{b\}, \{\neg b, c\} \rangle, \\
&\quad\ o_2 = \langle \emptyset, \{e\}, \{b\} \rangle, \\
&\quad\ o_3 = \langle \emptyset, \{c\}, \{b, d, \neg e\} \rangle, \\
\mathbf{I} &= \{a, b\} \\
\mathbf{G} &= \{b, d\}
\end{aligned}
$$



Conclusion: Thus, we have successfully performed study on planning problem.

Name: Kartik Jolapara **SAPID:** 60004200107

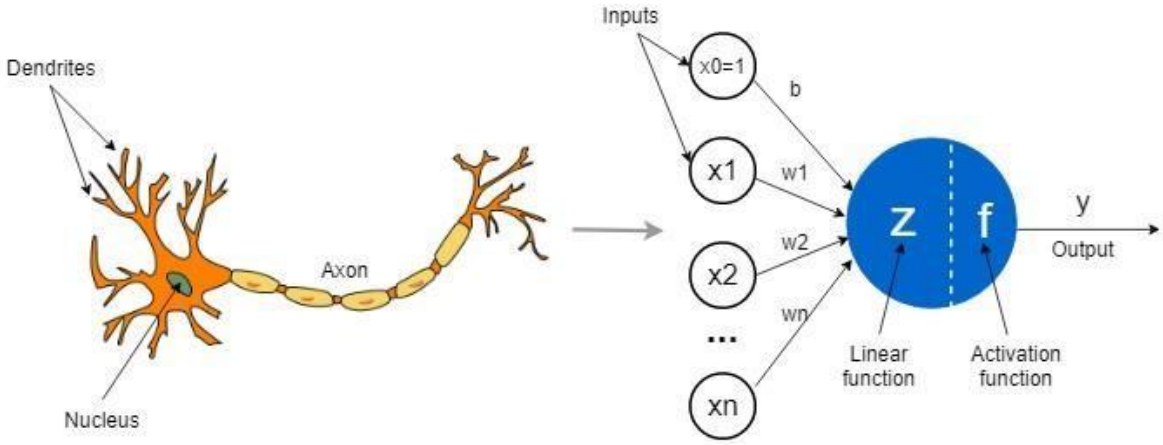
DIV: B/B1

AI

Exp9

Aim: Neural Network Perceptron Learning

Theory: Artificial neurons (also called Perceptrons, Units or Nodes) are the simplest elements or building blocks in a neural network. They are inspired by biological neurons that are found in the human brain.It is worth discussing how artificial neurons (perceptrons) are inspired by biological neurons. You can consider an artificial neuron as a mathematical model inspired by a biological neuron.

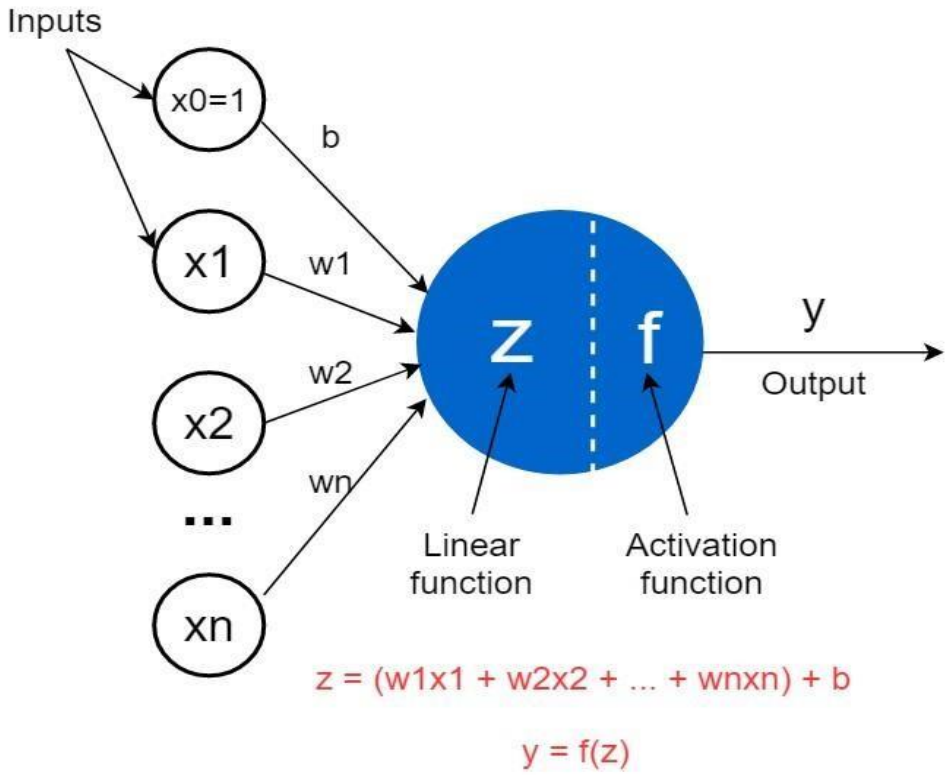


- A biological neuron receives its input signals from other neurons through **dendrites** (small fibers). Likewise, a perceptron receives its data from other perceptrons through **input neurons** that take numbers.
- The connection points between dendrites and biological neurons are called **synapses**. Likewise, the connections between inputs and perceptrons are called **weights**. They measure the importance level of each input.
- In a biological neuron, the ***nucleus*** produces an output signal based on the signals provided by dendrites. Likewise, the ***nucleus*** (colored in blue) in a perceptron performs some calculations based on the input values and produces an output.

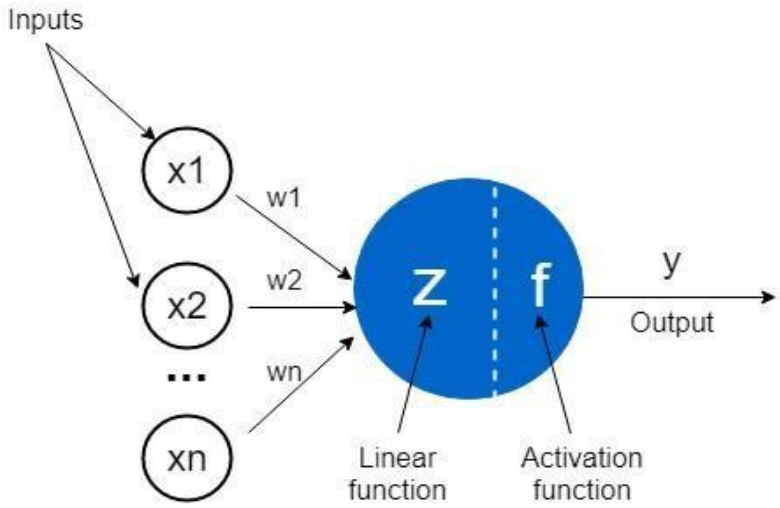
- In a biological neuron, the output signal is carried away by the ***axon***. Likewise, the axon in a perceptron is the **output value** which will be the input for the next perceptrons.

The structure of a perceptron

The following image shows a detailed structure of a perceptron. In some contexts, the bias, **b** is denoted by **w0**. The input, **x0** always takes the value 1. So, **b*1 = b**.



A perceptron takes the inputs, x1, x2, …, xn, multiplies them by weights, w1, w2, …, wn and adds the bias term, b, then computes the linear function, z on which an activation function, f is applied to get the output, y.
When drawing a perceptron, we usually ignore the bias unit for our convenience and simplify the diagram as follows. But in calculations, we still consider the bias unit.



Inside a perceptron

A perceptron usually consists of two mathematical functions.

Perceptron's linear function

This is also called the linear component of the perceptron. It is denoted by **z**. Its output is the weighted sum of the inputs plus bias unit and can be calculated as follows.

$$z = (w1.x1 + w2.x2 + \ldots + wn.xn) + b$$

Perceptron’s linear function (Image by author, made with draw.io)

- The **x1**, **x2**, …, **xn** are inputs that take numerical values. There can be several (finite) inputs for a single neuron. They can be raw input data or outputs of the other perceptrons.
- The **w1**, **w2**, …, **wn** are ***weights*** that take numerical values and control the level of importance of each input. The higher the value, the more important the input.
- **w1.x1 + w2.x2 + … + wn.xn** is called the weighted sum of inputs.
- The **b** is called the ***bias term*** or ***bias unit*** that also takes a numerical value. It is added to the weighted sum of inputs. The purpose of including a bias term

is to shift the activation function of each perceptron to not get a zero value. In other words, if all **x1**, **x2**, …, **xn** inputs are 0, the **z** is equal to the value of bias.

The weights and biases are called the ***parameters*** in a neural network model. The optimal values for those parameters are found during the learning (training) process of the neural network.

You can also think of the above **z** function as a linear regression model in which weights are known as ***coefficients*** and the bias term is known as the ***intercept***. This is just the terminology used to identify the same thing in different contexts.

Perceptron's non-linear (activation) function

This is also called the non-linear component of the perceptron. It is denoted by **f**. It is applied on **z** to get the output **y** based on the type of activation function we use.

$$y = f(z)$$

$$y = f((w1.x1 + w2.x2 + \ldots + wn.xn) + b)$$

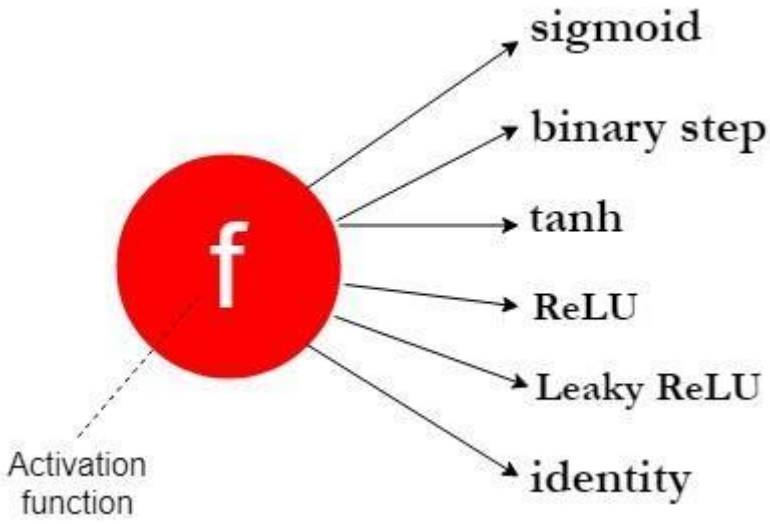
Output

Activation function (non-linear)

Weighted sum of inputs (linear combination)

Bias

The function **f** can be a different type of activation function.



Type of activation functions (Image by author, made with draw.io)

As there are many different types of activation functions, we'll discuss them in detail in a separate article. For now, it is enough to remember that the purpose of an activation function is to introduce non-linearity to the network. Without an activation function, a neural network can only model linear relationships and can't model non-linear relationships present in the data. Most of the relationships are non-linear in real-world data. Therefore, neural networks would be useless without activation functions.

What does it mean by "firing a neuron"?

For this, consider the following ***binary step*** activation function which is also known as the ***threshold activation function***. We can set any value to the threshold and here we specify the value 0.

$$\text{binary step}(z) = \begin{cases} \mathbf{1} & \text{if } z > 0 \\ \mathbf{0} & \text{if } z \leq 0 \end{cases}$$

Binary step activation function (Image by author, made with draw.io)

We say that a neuron or perceptron fires (or activates) only when the value of **z** exceeds the threshold value, 0. In other words, a neuron outputs 1 (fires or activates) if the value of **z** exceeds the threshold value, 0. Otherwise, it outputs

0.

Therefore, the type of activation function determines how the neuron activates or fires and the bias term ***b*** controls the ease of firing. Now consider the linear function, **z**. **z = (w1.x1 + w2.x2 + … + wn.xn) + b z = (weighted sum of inputs) + bias**

Let's assume bias is -2. Here also, we consider the binary step activation function. So, the neuron fires (activates) only when the weighted sum of inputs exceeds +2. In mathematical terms, this can be expressed as follows.

To fire the neuron, it should output 1 according to the binary step activation function defined above. It happens only when,
z > 0 (weighted sum of inputs) + bias > 0 (weighted sum of inputs) > -bias
When the bias is -2 in our example,

(weighted sum of inputs) > -(-2) (weighted sum of inputs) > 2

Therefore, in this case, the weighted sum of inputs should exceed +2 to fire or activate the neuron.

Perform calculations inside a perceptron

Let's perform a simple calculation inside a perception. Imagine that we have 3 inputs with the following values.

x1=2, **x2=3** and **x3=1**

Because we have 3 inputs, we also have 3 weights that control the level of importance of each input. Assume the following values for the weights.

w1=0.5, **w2=0.2** and **w3=10**

We also have the following value for the bias unit.

b=2

Let's calculate the linear function, **z.**

z = (0.5*2 + 0.2*3 + 10*1) + 2

z = 13.6

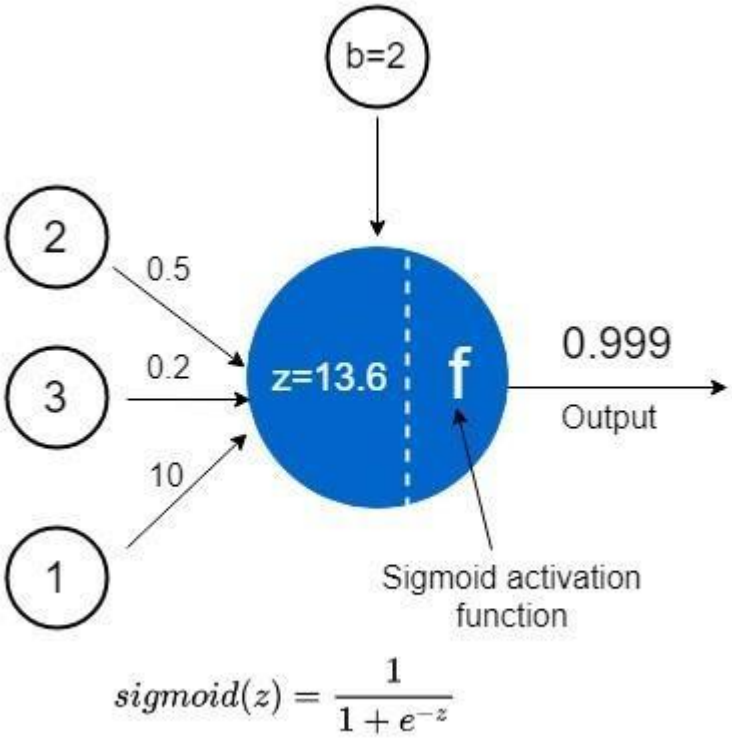
The activation function takes the output of **z** (13.6) as its input and calculates the output **y** based on the type of activation function we use. For now, we use the ***sigmoid*** activation function defined below.

$$sigmoid(z) = \frac{1}{1 + e^{-z}}$$

Sigmoid activation function (Image by author, made with draw.io)

y = sigmoid(13.6) y = 0.999 y ~ 1

The entire calculation process can be denoted in the following diagram. For ease of understanding, we also denote the bias term in a separate node.



$$sigmoid(z) = \frac{1}{1 + e^{-z}}$$

The calculation process inside a perceptron (Image by author, made with draw.io)

Code:

```
from random import random

class Perceptron:
    def __init__(self, inputSize):
        self.inputLayerSize = inputSize
        self.weights = []
        self.bias = random()*10 - 5
        for i in range(inputSize):
            self.weights.append(random()*10 - 5)
    def normalizeValue(val):
        if val < 0:
            return -1
        else:
            return 1
    def processInput(self, nnInput): # nnInput is an array containing all input values
        # The number of elements in nnInput should be equal to the number of weights
        assert len(nnInput) == len(self.weights)
        unprocessedOutputVal = self.bias # Initialize the weighted sum with the bias
        for i in range(len(nnInput)): # Add the rest of the weighted sum
            unprocessedOutputVal += nnInput[i]*self.weights[i]
        return normalizeValue(unprocessedOutputVal) # Return the formatted value
def lineY(x):
    return 1.6*x + 3.4 # My line equation, feel free to choose a different one
def generateTrainingSet(trainingSize):
    trainingSet = []
    for i in range(trainingSize):
        # Make sure your points live somewhat around your line
        x = random()*20 - 10
        y = random()*20 - 10
        if y > lineY(x):
            output = 1 # Point above the line
        else:
            output = -1 # Point below the line
```

```
            trainingSet.append([x, y, output])          return
trainingSet      def trainOnInput(self, inputVals,
expectedOutputVal, learningRate):
        nnVal = self.processInput(inputVals)         error = expectedOutputVal
- nnVal
self.adjustForError(inputVals, error, learningRate)


    def adjustForError(self, inputVals, error, learningRate):        for i in
range(len(self.weights)):              self.weights[i] +=
error*inputVals[i]*learningRate         self.bias
+= error*learningRate
 perceptron = Perceptron(2) trainingRate = 0.1
trainingSetSize = 1000000 trainingSet =
generateTrainingSet(trainingSetSize)   testSetSize =
1000 testSet = generateTrainingSet(testSetSize)
score = 0 for test in testSet:     if
perceptron.processInput([test[0], test[1]]) == test[2]:
        score += 1   print("Score before training
{}/{}".format(score, testSetSize))   for j in
range(trainingSetSize):
perceptron.trainOnInput([trainingSet[j][0], trainingSet[j][1]],
trainingSet[j][2], trainingRate)   score = 0 for test in testSet:
if perceptron.processInput([test[0], test[1]]) == test[2]:
        score += 1

print("Score after training {}/{}".format(score, testSetSize))
```

Output:

```
Iteration 1
Generated Output vector for Iteration 1 : [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, 1]
Weight vector after Iteration 1 : [0.2, 0.6, 0.0, 0.6, 0.2, -0.9, 0.4, 0.6, -0.6, 0.1, 0.1, -0.1, 0.4,
0.9, -0.9, 0.1, 1.0, -0.3, 1.0, 0.1]
--------------------------------------------------------------------------------------------------------------------------------
Iteration 2
Generated Output vector for Iteration 2 : [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1, 1, -1, -1, -1, -1, -1,
-1]
Weight vector after Iteration 2 : [0.1, 0.5, 0.0, 0.5, 0.1, -1.0, 0.4, 0.5, -0.6, 0.0, 0.0, -0.1, 0.3,
0.9, -1.0, 0.0, 1.0, -0.3, 1.0, 0.0]
--------------------------------------------------------------------------------------------------------------------------------
Iteration 3
Generated Output vector for Iteration 3 : [1, 1, 1, 1, -1, 1, 1, 1, 1, 1, -1, -1, -1, 1, -1, -1, -1, -1, -1,
-1]
Weight vector after Iteration 3 : [0.1, 0.4, 0.0, 0.4, 0.0, -1.0, 0.4, 0.4, -0.6, -0.1, 0.0, -0.1, 0.2,
0.9, -1.0, 0.0, 1.1, -0.2, 1.1, 0.0]
--------------------------------------------------------------------------------------------------------------------------------
Accuracy of Classifier : 90.0 %
Classifying an Unknown Sample of L (Output = 1)
Unknown Sample :  [1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0]
Predicted Output :  1
```

Conclusion: We successfully implemented Neural Network Perceptron Learning

Name: Kartik Jolapara **SAPID:** 60004200107

DIV: B/B1

AI

Experiment 10

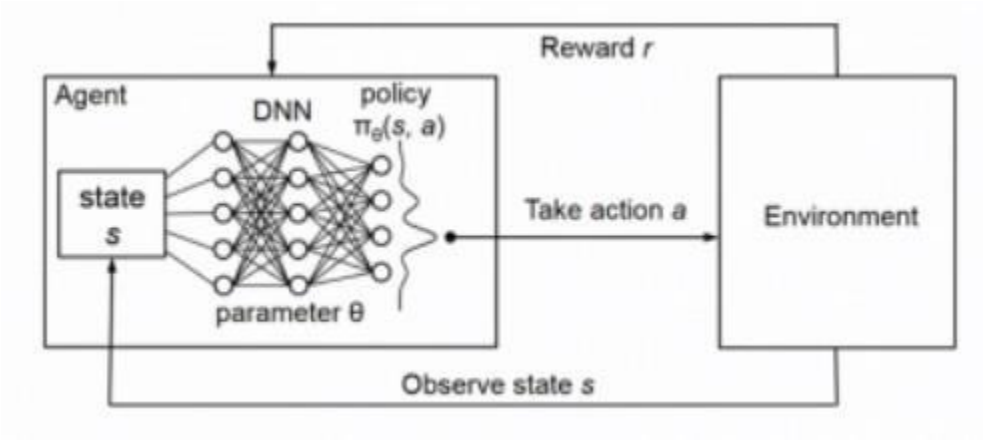
CASE STUDY
Paper Link: https://ieeexplore.ieee.org/abstract/document/9325622

Introduction:
This paper is a comparative study for deep reinforcement learning with CNN, RNN, and LSTM in autonomous navigation. For the comparison, a PyGame simulator has been used with the final goal that the representative will learn to move without hitting four different fixed obstacles. Autonomous vehicle movements were simulated in the training environment and the conclusion drawn was that the LSTM model was better than the others.

Approach:
The research is wholly based on reinforcement learning which is a machine learning training method based on rewarding desired behaviors and/or punishing undesired ones. This method assigns positive values to the desired actions to encourage the agent and negative values to undesired behaviors. This programs the agent to seek long-term and maximum overall reward to achieve an optimal solution.
These long-term goals help prevent the agent from stalling on lesser goals. With time, the agent learns to avoid the negative and seek the positive. This learning method has been adopted in artificial intelligence (AI) as a way of directing unsupervised machine learning through rewards and penalties.
The main advantage of reinforcement learning in this scenario is that unlike deep learning (DL) algorithms, it does not require a data set during the training phase, increasing its popularity and making it more suitable.

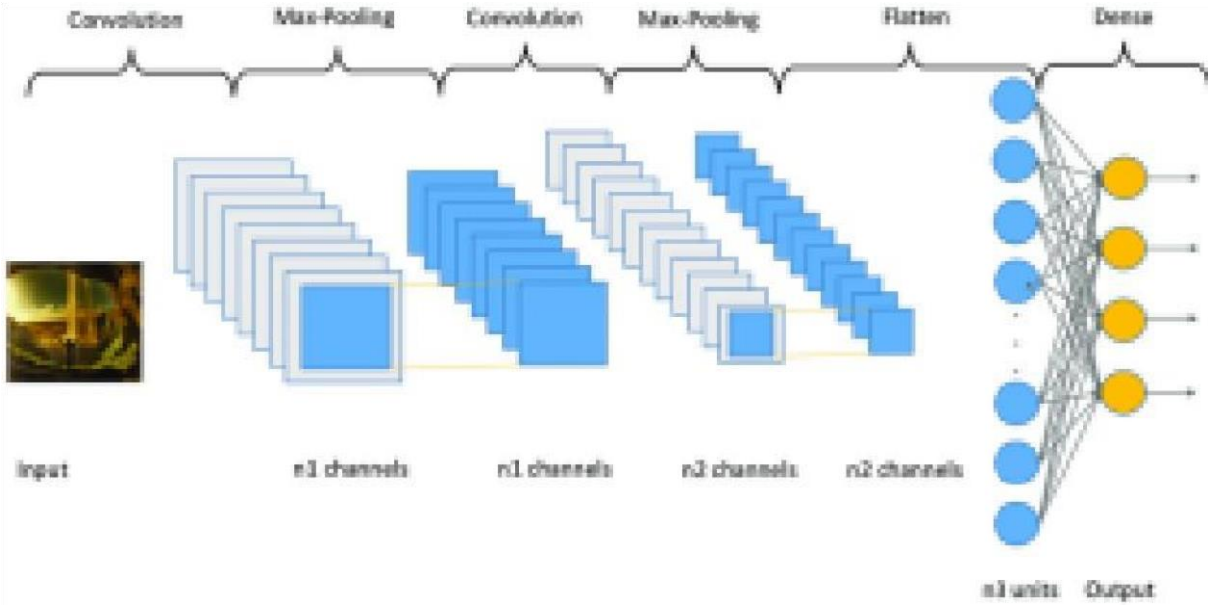
The PyGame simulator interface consists of an agent that learns to move without hitting 4 different randomly positioned obstacles and edges limiting the area. In addition, the paper presents a model-free, off policy approach in this study.
During the research, 4 algorithms were compared. They are:

1. **Deep Q-Network:**It trains on inputs that represent active players in areas or other experienced samples and learns to match those data with desired outputs. This is a powerful method in the development of artificial intelligence that can play games like chess at a high level, or carry out other high-level cognitive activities – the Atari or chess video game playing example is also a good example of how AI uses the types of interfaces that were traditionally used by human agents.



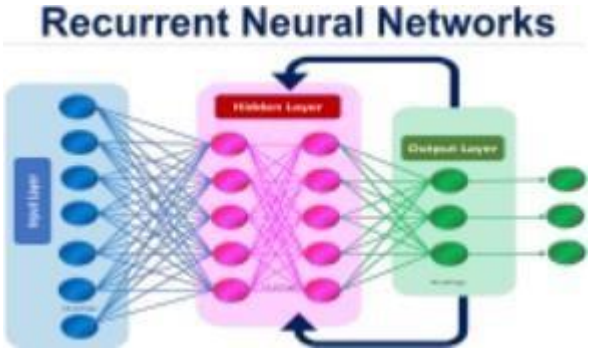
2. CNN: A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other.

   The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics.
   The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area. The main purpose of the convolution process is to extract the feature map from the input data.



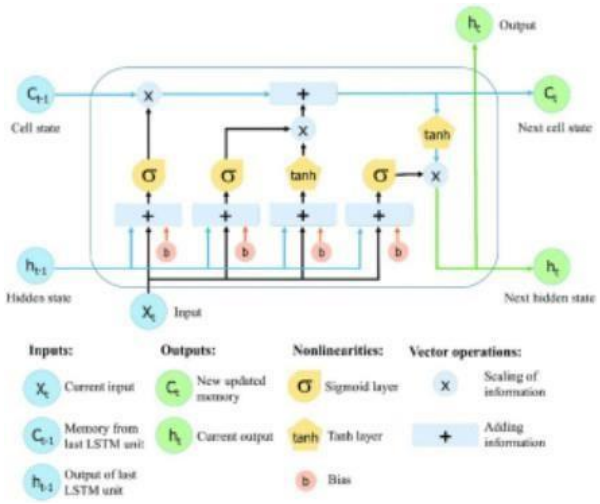
3. RNN: Recurrent Neural Network(RNN) is a type of Neural Network where the output from the previous step is fed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other, but in cases like when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words.

Thus RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is the Hidden state, which stores some information about a sequence.



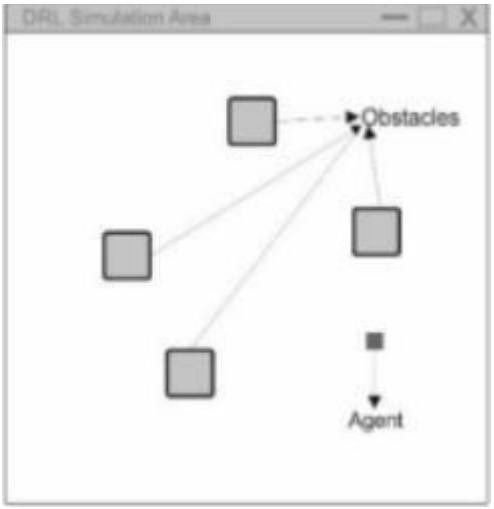
4. LSTM: Long Short Term Memory is a kind of recurrent neural network. In RNN output from the last step is fed as input in the current step. LSTM was designed by Hochreiter & Schmidhuber. It tackled the problem of long-term dependencies of RNN in which the RNN cannot predict the word stored in the long-term memory but can give

more accurate predictions from the recent information. As the gap length increases RNN does not give an efficient performance. LSTM can by default retain the information for a long period of time. It is used for processing, predicting, and classifying on the basis of time-series data. LSTM has a chain structure that contains four neural networks and different memory blocks called cells. Information is retained by the cells and the memory manipulations are done by the gates. There are three gates – Forget gate, Input gate and the output gate. With LSTMs, there is no need to keep a finite number of states from beforehand as required in the hidden Markov model (HMM). LSTMs provide us with a large range of parameters such as learning rates, and input and output biases. Hence, no need for fine adjustments.



The complexity to update each weight is reduced to O(1) with LSTMs, similar to that of Back Propagation Through Time (BPTT), which is an advantage.

SIMULATION: In this work, PyGame library was used as a robot simulation environment. 4 different obstacles were placed randomly in a 360 * 360 pixel area and the agent was allowed to float within the specified area without hitting these obstacles as shown below.



The agent loses -150 points when hitting obstacles during the learning phase and -50 points when it hits walls. It gets +2 points for every step where it does not hit walls and obstacles. Four different actions in this simulation are shown in the table below.

Num.	Action
0	go to the left
1	go to the right
2	go to the up
3	go to the bottom

For the model training, python was used as the software language and Keras, a deep learning library was used to create the neural networks. Mean Squared Error was used as the Loss function.

Linear was preferred as the activation function. Sigmoid function is used as the activation function in the output layer. The training took 5 hours for CNN, 18 hours for RNN and 35 hours for LSTM on a standard equipped (core i5 Processor and 8GB RAM) computer.

CONCLUSION: Multiple deep learning algorithms were separately tested on the PyGame simulation interface and the conclusions were drawn. The first conclusion was that even though deep reinforcement learning (DRL) models provide fast and safe solutions for autonomous vehicles, their training time is very high. After training it was observed that RNN and LSTM,which are generally used to solve language processing problems, can also be successful in such autonomous navigation problems. The second conclusion was that while the LSTM model took the maximum time to train, it showed the highest success in the success-episode graphics. The paper then concludes with saying that this is a very rich field in terms of future research prospects.

6000420010 7

B1

PAGE NO.:
DATE: / /

A.I.

Assignment - 1

Q1) Write short note on any two :

Ans a) Hierarchical Planning :

It is an Artificial Intelligence problem solving approach for a certain kind of planning problems the kind favoring problem decomposition where problems are stepwise refined into smaller and smaller ones until the problem is finally solved. A solution here by is a sequence of actions that's executable in a given initial state (and a refinement of initial compound tasks that needed to be refined). This form of hierarchical planning is usually referred to as Hierarchical Task Network (HTN) planning but many variants and extensions exist. A solution to HTN problem is of primitive tasks that can obtained from initial state by decomposing compound tasks into their set of simpler tasks and by inserting ordering constraints.

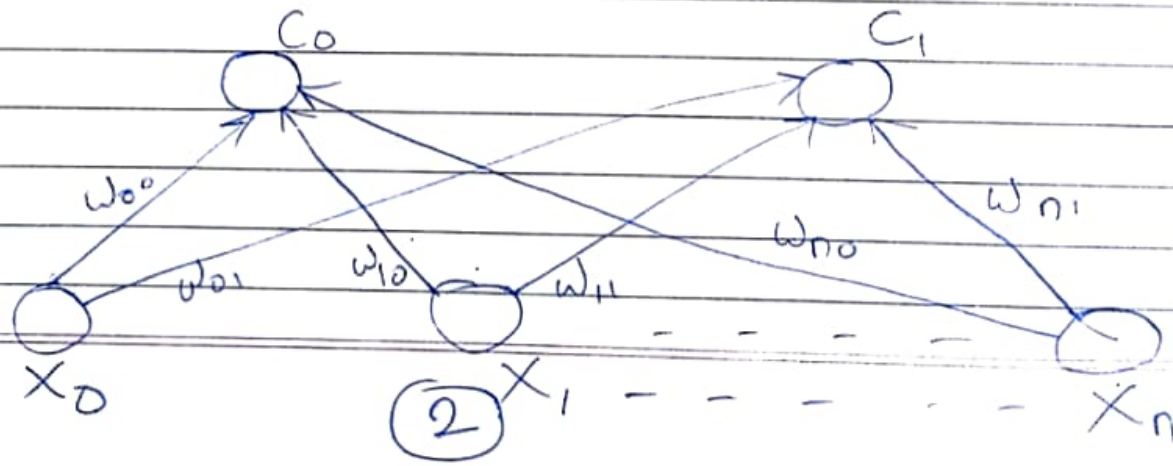
c) Multiagent Planning :

It involves coordinating resources and activities of multiple agents NASA says "Multi-agent planning is concerned with planning by multiple agents. It can involve agents planning for a common goal or agent coordinating the plans or planning of others

of agents refining their own plans while negotiating over task or resources. The topic also involves how agents can do this in real time while executing plans. Multiagent scheduling differs from multiagent planning in same ways scheduling and planning differ.

Q2) Self Organizing Maps (SOM)

Ans) Self organizing Maps is type of Artificial Neural Network which also inspired by biological models and of neural systems from 1970's. It follows an unsupervised learning approach and trained its network through a competitive learning algorithm. SOM is used for clustering and mapping techniques to map multidimensional data onto lower dimensional which allows people to reduce complex problem for easy interpretation. SOM has 2 layers, one is Input layer and other one is Output layer. The architecture of SOM with 2 clusters and n inputs features of an sample given below



For input data of size (m, n) where m is number of training examples and n is number of features in each example. First it initializes the weights of size (n, c) where c is number of clusters.

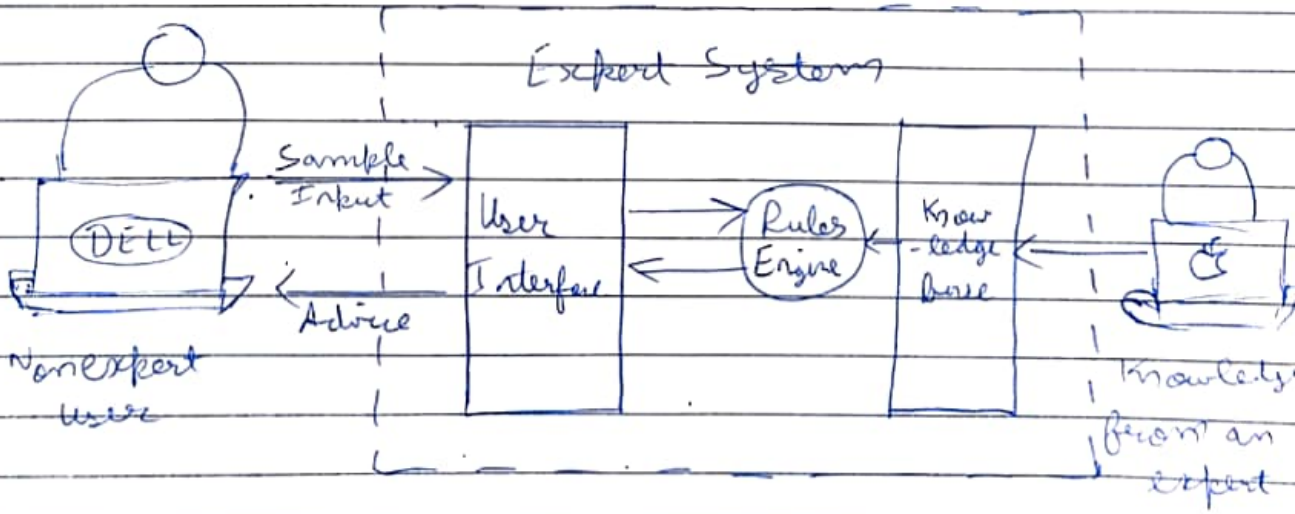
$w_{ij} = w_{ij}(old) + alpha(t) * (x_i^k - w_{ij}(old))$

where alpha is learning rate at time t, After training SOM network, trained weights are used for clustering new examples.

Q3)

Ans An Expert system is computer program that is designed to solve complex problems and to provide decision making ability like human expert. It performs this by extracting knowledge from its knowledge base using reasoning and inference rules according to user queries.

Block diagram



PAGE NO
DATE

Components of Expert System

- **User Interface:**
  It helps a non expert user to communicate with expert system to find a solution.

- **Interface Engine (Rules of engine):**
  It applies interface rules to knowledge base to derive conclusion or deduce new information and this helps system

- **Knowledge Base:**
  It is type of storage that stores knowledge aquired from different experts of particular domain. The more knowledge base, the more precise will be expert systems

Development of expert system.

Taking an example of MYCINES. Some steps to build an MYCIN are

- ES should be fed with expert knowledge. In case of MYCIN, human experts specialized in medical field of bacterial infection, provide information about causes, symptoms and other knowledge in domain.

- The KB of MYCIN is updated successfully - I, order to test it, the doctor provides a new problem to it. The problem is to identify presence of bacteria by inputting details of patient including symptoms.

PRACTICAL
PAGE NO.:
DATE: / /

- The ES will need a questionnaire to filled by patient to know general information about patient such as gender, age, etc.
- Now the system has collected all information so it will find solution for problem by applying if then rules using inference engine and using facts stored within the KB
- In the end it will provide a response to patient by using the user interface.

Q4)

Ans. Natural Language Processing (NLP)

NLP is subfield of Linguistic, computer science and artificial intelligence concerned with interaction between computers and human language in particular how to program computers to process and analyse large amounts of natural language data. The goal is computer capable of understand the contents of documents including contextual nuances of language within them. The technology can then accurately extract info and insights contained in the documents as well as categorize and organize the documents themselves.

Challenges in NLP frequently involve speech recognition, natural language understanding and natural language generation.

PRADNAT
PAGE NO.:
DATE: / /

Advantages of NLP

- NLP helps users to ask questions about any subject and get a direct response within seconds.
- NLP offers exact answers to questions means it does not offer necessary and unwanted information.
- NLP helps computers to communicate with humans in their languages.
- It is very time efficient.
- Most of companies use NLP to improve efficiency of documentation processes accuracy of documentation and identify the information from large database.

Disadvantages of NLP

- NLP may not show context
- NLP is unpredictable
- NLP may require more keystrokes
- NLP is unable to adopt to new domain and it has limited function that's why NLP is built for single and specific task only.

Applications of NLP

- Question answering :-

It focuses on building systems that automatically answer question asked by human in natural language.

PRABHAT
PAGE NO. :
DATE : / /

- Spam detection
  It is used to detect unwanted e-mails getting to a user's inbox

- Sentiment Analysis
  It is used on web to analyse attitude behaviour and emotional state of sender.

- Machine Translation
  Translate text or speech from 1 language to another apart from these same other applications are.
  - Spelling correction
  - Chatbot
  - Information extraction
  - Natural Language Understanding.

60004200107

B-1

PRABHAT
PAGE NO. :
DATE : / /

AI

Assignment - 2

Q 1)

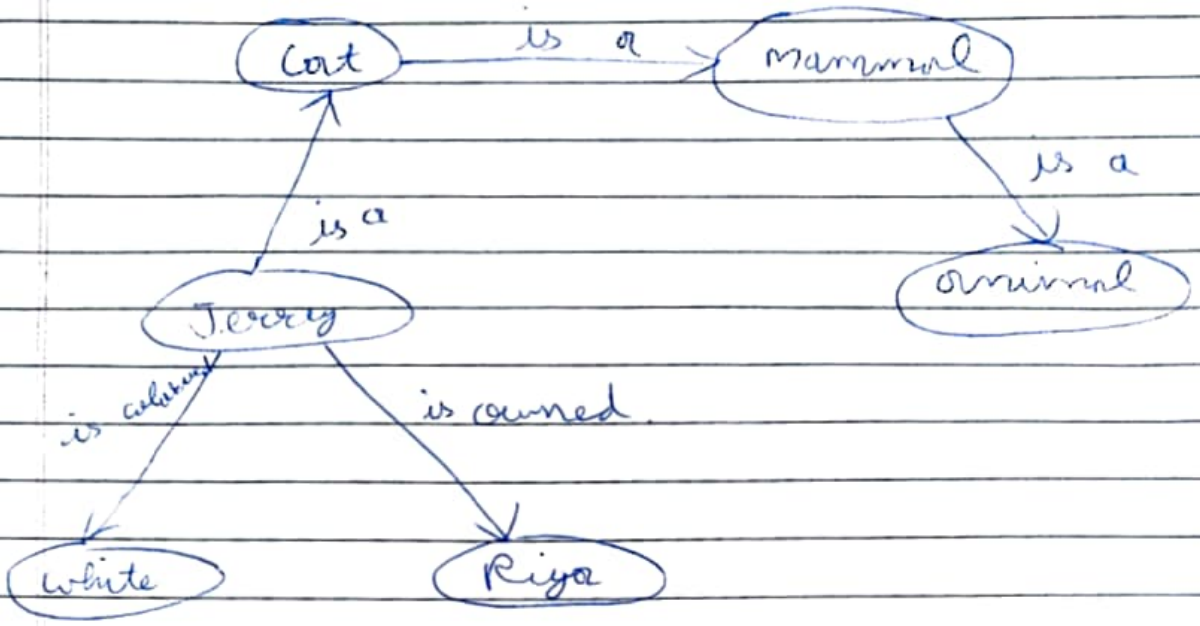
Ans i) Semantic Networks:

A semantic network is a graphic notation for representation knowledge in patterns of interconnected nodes. It became popular in patterns of popular in Artificial Intelligence and Natural Language Processing only because it knowledge or support reasoning. These acts as another alternation for predicate logic in a form of knowledge representation.

This network consists of nodes representing objects and arc which describe the relationship between these objects. which describe the semantic networks categorise in different forms and can also link these objects. This network are easy to understand and can be easily extended.

Example

- Jerry is cat
- Jerry is mammal
- Jerry is owned by Riya
- Jerry is white coloured
- All mammals are animal



ii) The semantic web makes use of RDF and OWL which occur in 2 layers as follows.

RDF is acronym for Resource Description Framework which is special type of framework found online that is dedicated with representation of online exchange of data

OWL is acronym for Ontology Web Language which is special language used in description of ontologies online

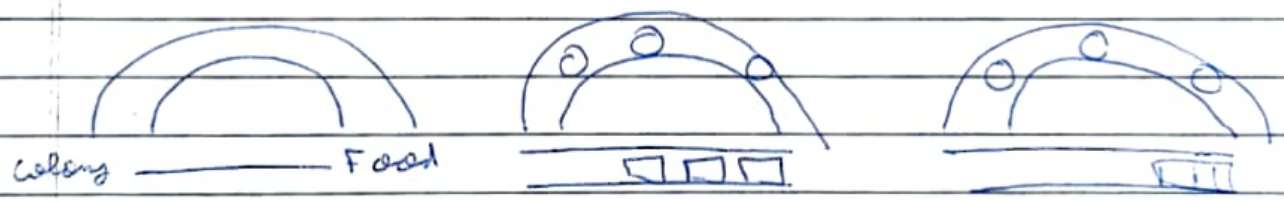
RDF allows expression of relationship between things while OWL is similar but bigger better and broader. Some other major differences are Vocabulary, logical consistency and Annotations / metadata OWL gives rich variety of annotation unlike RDF, which OWL satisfies

all metaheuristic modelling needs.

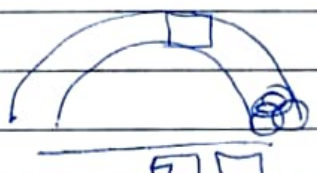
Q 2)

Ans Ant Colony Optimization is a population based metaheuristic that can be used to find approximate solutions to difficult optimization problems. 3/7

In ant colony optimization a set of software agents called artificial ants search for good solutions to given optimization problem. These ants incrementally build solutions by moving on weight. The solution construction process is clockwise and is based by pheromone model, that is, set of parameters associated with graph components



Stages : 1 2 3



4

Ants all are in nest. There is no pheromone evaporation or no new ants begin

to search with equal probability. More ants return via shorter path. Therefore while colony generally uses shorter path.

Q 3>

Ans Unification is process of making 2 different logical atomic expressions identical by finding a substitution. It depends on substitution process.

It takes literals as inputs and makes them identical using substitution. Let ψ_1 & ψ_2 be 2 atomic sentences that $\psi_1 \sigma = \psi_2 \sigma$ then it can be expressed as

$$\text{Unify}(\psi_1, \psi_2)$$

For example:

Find MGU for Unify (King (x), King (John))

Let ψ_1 = king(x) ψ_2 = king(John)

substitution θ = {John/x} is a unifier for these atoms and applying this substitution and both expression will be identical. Unification is key component of all first order inference. It returns fail if expression don't match with each other. Substitution variables are called as Most general unifier MGU.

PRABHAT
PAGE NO.:
DATE: / /

Q4>

Ans Bayesian belief network is key computer technology for dealing with probabilistic events to solve a problem which has uncertainty.

A Bayesian Network can be defined as follows

'A probabilistic graphical model which represents set of variables and their conditional dependencies using a directed acyclic graph'

Bayesian network are probabilistic because these network are built from a probability distribution and also are probability for prediction and anomaly detection

It can also be used in various tasks including prediction, diagnostic, automated, insight, reasoning, time series prediction and decision making under uncertainty.

Bayesian network can be used for building models for data and experts opinions

It consist 2 parts:

Directed Acyclic graph.

Table of Acyclic Conditional Probability

The generalized form of Bayesian network. represents and solves problems under certain knowledge is called Influence diag.

PRABHAT
PAGE NO. :
DATE : / /

Q5)

Ans Fuzzy refers to something that is vague. Here Fuzzy set is a set where every key is associated with value which is between 0 to 1 based on uncertainty. This value is often called as degree of membership. Fuzzy set is denoted with a double sign on top of the normal set.

Fuzzy set operations:

1) Union:

degree-of-membership (U) = max (degree-of-membership (A), degree-of-membership (B))

For example:

A = {'a': 0.2, 'b': 0.3, 'c': 0.6}

B = {'a': 0.9, 'b': 0.9, 'c': 0.4}

U = {'a': 0.9, 'b': 0.9, 'c': 0.6}

2) Insertion:

degree-of-membership (I) = min (degree of membership of x and y)

For example:

A = {'a': 0.2, 'b': 0.3}

B = {'a': 0.9, 'b': 0.9}

I = {'a': 0.2, 'b': 0.3}

PRABHAT
PAGE NO. :
DATE : / /

3) Complement

degree-of-membership (Y) = 1 - degree of membership (A)

For example:

A = {'a': 0.2, 'b': 0.3}

Y = {'a': 0.8, 'b': 0.7}

4) Difference:

degree of membership (Y) = min (degree of membership (A), 1 - degree of membership (B))

For example:

A = {'a': 0.2, 'b': 0.3}

B = {'a': 0.9, 'b': 0.9}

Y = {'a': 0.1, 'b': 0.1}