SRM VALLIAMMAI ENGINEERING COLLEGE

(An Autonomous Institution)

SRM Nagar, Kattankulathur-603203

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

ACADEMIC YEAR: 2024-2025

EVEN SEMESTER

LAB MANUAL

(REGULATION - 2023)

AD3465 – ARTIFICIAL INTELLIGENCE - I LABORATORY

FOURTH SEMSTER

B. Tech – Artificial Intelligence and Data Science

Prepared By

V. ABARNA, A.P (O.G) / AI&DS

INDEX

PROGRAMME EDUCATIONAL OBJECTIVES (PEOs)

1. To afford the necessary background in the field of Information Technology to deal with engineering problems to excel as engineering professionals in industries.
2. To improve the qualities like creativity, leadership, teamwork and skill thus contributing towards the growth and development of society.
3. To develop ability among students towards innovation and entrepreneurship that caters to the needs of Industry and society.
4. To inculcate and attitude for life-long learning process through the use of information technology sources.
5. To prepare then to be innovative and ethical leaders, both in their chosen profession and in other activities.

PROGRAMME OUTCOMES (POs)

After going through the four years of study, Information Technology Graduates will exhibit ability to:

PO#	Graduate Attribute	Programme Outcome
1	Engineering knowledge	Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization for the solution of complex engineering problems.
2	Problem analysis	Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3	Design/development of solutions	Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for public health and safety, and cultural, societal, and environmental considerations.
4	Conduct investigations of complex problems	Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions

5	Modern tool usage	Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools, including prediction and modeling to complex engineering activities, with an understanding of the limitations.
6	The engineer and society	Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal, and cultural issues and the consequent responsibilities relevant to the professional engineering practice
7	Environment and sustainability	Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8	Ethics	Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice
9	Individual and team work	Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings
10	Communication	Communicate effectively on complex engineering activities with the engineering community and with the society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions
11	Project management and finance	Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments
12	Life-long learning	Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change

PROGRAMME SPECIFIC OUTCOMES (PSOs)

After the completion of Bachelor of Technology in Artificial Intelligence and Data Science programme the student will have following Program specific outcomes

1. Design and develop secured database applications with data analytical approaches of data preprocessing, optimization, visualization techniques and maintenance using state of the art methodologies based on ethical values.
2. Design and develop intelligent systems using computational principles, methods and systems for extracting knowledge from data to solve real time problems using advanced technologies and tools.
3. Design, plan and setting up the network that is helpful for contemporary business environments using latest software and hardware.
4. Planning and defining test activities by preparing test cases that can predict and correct errors ensuring a socially transformed product catering all technological needs.

AD3465	ARTIFICIAL INTELLIGENCE I LABORATORY	L T P C
		0 0 3 1.5

OBJECTIVES:

This course will enable students to

- Understand simple PEAS descriptions for given AI tasks.
- Apply and implement simulated annealing and genetic algorithms.
- Solve problems using searching and backtracking.
- Design and implement simple reasoning systems using inference mechanisms.
- Understand the Implementation of inference mechanisms and planning algorithms.

LIST OF EXPERIMENTS:

1. Develop PEAS descriptions for given AI tasks
2. Study of PROLOG
3. Implement basic search strategies for selected AI applications
4. Implement A* and memory bounded A* algorithms
5. Implement genetic algorithms for AI tasks
6. Implement simulated annealing algorithms for AI tasks
7. Implement alpha-beta tree search
8. Implement backtracking algorithms for CSP
9. Implement local search algorithms for CSP
10. Implement propositional logic inferences for AI tasks
11. Implement resolution based first order logic inferences for AI tasks
12. Implement classical planning algorithms

Total: 45 Periods

LIST OF EQUIPMENTS FOR A BATCH OF 30 STUDENTS

SOFTWARE:

Python, PROLOG, C++ or Java Software

HARDWARE:

Standalone Desktops: 30 Nos.

COURSE OUTCOMES

1922405.1	Implement simple PEAS descriptions for given AI tasks.
1922405.2	Develop programs to implement simulated annealing and genetic algorithms.
1922405.3	Demonstrate the ability to solve problems using searching and backtracking.
1922405.4	Ability to Implement simple reasoning systems using inference mechanisms.
1922405.5	Will be able to choose and implement planning algorithms.

CO- PO-PSO MATRIX

CO	PO												PSO			
	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3	4
1	2	-	-	3	-	-	-	-	-	-	-	1	2	2	-	-
2	-	2	-	3	-	-	-	-	-	2	-	-	1	3	-	-
3	3	-	3	-	1	-	-	-	1	-	-	-	-	1	-	3
4	-	-	-	1	1	-	-	-	2	-	-	-	-	2	-	1
5	3	-	-	-	1	-	-	-	-	-	2	3	3	3	-	-
Average	**2.7**	**2.0**	**3.0**	**2.3**	**1.0**	**-**	**-**	**-**	**1.5**	**2.0**	**2.0**	**2.0**	**2.0**	**2.2**	**-**	**2.0**

EVALUATION PROCEDURE FOR EACH EXPERIMENT

S. No	Description	Mark
1.	Aim & Procedure	20
2.	Observation	30
3.	Conduction and Execution	30
4.	Output & Result	10
5.	Viva	10
Total		**100**

INTERNAL ASSESSMENT FOR LABORATORY

S. No	Description	Mark
1.	Conduction & Execution of Experiment	25
2.	Record	10
3.	Model Test	15
Total		**50**

Ex.No.1 **Develop PEAS Description for given AI tasks**

Aim:

To develop PEAS description for the following AI tasks

(a) Automated Taxi Driving

(b) Vacuum Cleaner Agent

(c) Medical Diagnosis System

Algorithm:

Step1: Start the program

Step2: Read the variable for getting the choice

Step3: Using switch statement to print the PEAS description for the given tasks.

Step4: Stop the program.

Program:

```
import java.io.*;
import java.util.*;
class PEAS
{
public static void main(String args[])throws IOException
{
int choice;
Scanner sc= new Scanner(System.in);
System.out.println(" **** PEAS DESCRIPTION ****\n ----------");
System.out.println("\n 1. Automated Taxi Driving\n 2. Vacuum cleaner \n 3. Medical Diagnosis
System");
System.out.println("Enter your choice:");
choice= sc.nextInt();
switch(choice)
{
case 1:
{
System.out.println("Performance : Safety, Comfort, Affordable");
System.out.println("Environment : Road, Signals, Pedastrian Cross, other vehicles");
System.out.println("Actuators : Steering, breaking, acceleration");
System.out.println("Sensors : Accelerometer, Speedometer,Front lights,cameras");
break;
}
case 2:
{
System.out.println("Performance: Cleanness, efficiency: distance travelled to clean, battery life,
security");
System.out.println("Environment: Room, table, wood floor, carpet, different obstacles");
System.out.println("Actuators: Wheels, different brushes, vacuum extractor");
System.out.println("Sensors: Camera, dirt detection sensor, cliff sensor, bump sensors, infrared
```

```
wall sensors");
break;
}
case 3:
{
System.out.println("Performance: Cleanness, efficiency: distance travelled to clean, battery life,
security");
System.out.println("Environment: Room, table, wood floor, carpet, different obstacles");
System.out.println("Actuators: Wheels, different brushes, vacuum extractor");
System.out.println("Sensors: Camera, dirt detection sensor, cliff sensor, bump sensors, infrared
wall sensors");
break;
}
default:
{
System.out.println("Enter the right choice");
break;
}}}}
```

Output:

```
D:\ AI Lab>javac PEAS.java
D:\AI Lab>java PEAS
 **** PEAS DESCRIPTION ****
 ---------------------------
 1. Automated Taxi Driving
 2. Vacuum cleaner
 3. Medical Diagnosis System
Enter your choice:
2
Performance: Cleanness, efficiency: distance travelled to clean, battery life, security
Environment: Room, table, wood floor, carpet, different obstacles
Actuators: Wheels, different brushes, vacuum extractor
Sensors: Camera, dirt detection sensor, cliff sensor, bump sensors, infrared wall sensors
```


Result:

Thus, the program is executed successfully and the output is verified.

Ex.No. 2: **Study of PROLOG**

Aim: Study of Prolog
Requirements: PROLOG-programming in logic.

Theory: PROLOG stands for Programming in Logic – an idea that emerged in the early 1970's to use logic as programming language. The early developers of this idea included Robert Kowalski at Edinburgh (on the theoretical side), Marrten van Emden at Edinburgh (experimental demonstration) and Alian Colmerauer at Marseilles (implementation). David D.H. Warren's efficient implementation at Edinburgh in the mid - 1970's greatly contributed to the popularity of PROLOG.

PROLOG is a programming language centered on a small set of basic mechanisms, including pattern matching, tree based data structuring and automatic backtracking. This Small set constitutes a surprisingly powerful and flexible programming framework. PROLOG is especially well suited for problems that involve objects- in particular, structured objects- and relations between them.

SYMBOLIC LANGUAGE
PROLOG is a programming language for symbolic, non-numeric computation. It is especially well suited for solving problems that involve objects and relations between objects.

For example, it is an easy exercise in prolog to express spatial relationship between objects, such as the blue sphere is behind the green one. It is also easy to state a more general rule: if object X is closer to the observer than object Y, and object Y is closer than Z, then X must be closer than Z. PROLOG can reason about the spatial relationships and their consistency with respect to the general rule. Features like this make PROLOG a powerful language for Artificial Language (AI) and non- numerical programming.

There are well-known examples of symbolic computation whose implementation in other standard languages took tens of pages of indigestible code. When the same algorithms were implemented in PROLOG, the result was a crystal-clear program easily fitting on one page.

FACTS, RULES AND QUERIES
Programming in PROLOG is accomplished by creating a database of facts and rules about objects, their properties, and their relationships to other objects. Queries then can be posed about the objects and valid conclusions will be determined and returned by the program. Responses to user queries are determined through a form of inference control known as resolution

FOR EXAMPLE:

a) **FACTS:**
Some facts about family relationships could be written as:
Sister (sue,bill)
parent (ann,sam)
male (jo)
female (riya)

b) **RULES:**
To represent the general rule for grandfather, we write:
grandfather (X,Z)
parent (X,Y)
parent (Y,Z)
male (X)

c) QUERIES:

Given a database of facts and rules such as that above, we may make queries by typing after a query a symbol '?_' statements such as:

?_ parent (X,sam)

X=ann ?_

grandfather(X,Y)

X=jo, Y=sam

PROLOG IN DESIGINING EPERT SYSTEMS

An expert system is a set of programs that manipulates encoded knowledge to solve problems in a specialized domain that normally requires human expertise. An expert system's knowledge is obtained from expert sources such as texts, journal articles, databases etc. and encoded in a form suitable for the system to use in its inference or reasoning processes. Once a sufficient body of expert knowledge has been acquired, it must be encoded in some form, loaded into knowledge base, then tested, and refined continually throughout the life of the system. PROLOG serves as a powerful language in designing expert systems because of its following features:

a) Use of knowledge rather than data
b) Modification of the knowledge base without recompilation of the control programs.
c) Capable of explaining conclusion.
d) Symbolic computations resembling manipulations of natural language.
e) Reason with meta-knowledge.

META PROGRAMMING

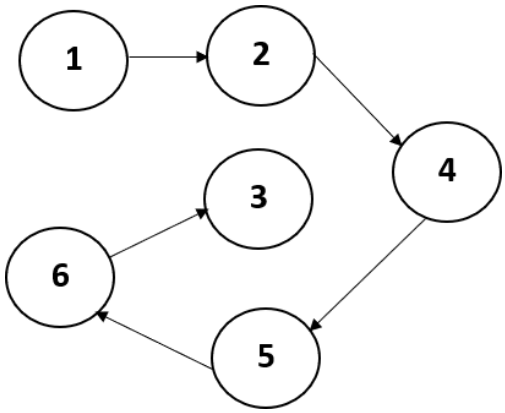
A Meta program is a program that takes other programs as data. Interpreters and compilers are examples of meta-programs. Meta-interpreter is a particular kind of meta- program: an interpreter for a language written in that language. So a PROLOG meteinterpreter is an interpreter for PROLOG, itself written in PROLOG. Due to its symbol- manipulation capabilities, PROLOG is a powerful language for meta-programming. Therefore, it is often used as an implementation language for other languages. PROLOG is particularly suitable as a language for rapid prototyping where we are interested in implementing new ideas quickly. New ideas are rapidly implemented and experimented with.

Ex.No. 3 **Implement basic search strategies for selected AI applications.**

Ex.No. 3 (a) **Perform Topological Sorting for a Directed Acyclic Graph using DFS Algorithm**

Aim:

To perform topological sorting for the given DA graph using Depth First Search Algorithm.



Algorithm:

Depth First Search (DFS):

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 (who's STATUS = 1) and set their STATUS = 2 (waiting state)

Step 6: EXIT

Topological Sorting:

Step 1: Create the graph by calling addEdge(a,b).

Step 2: Call the TopSort( )

Step 2.1: Create a stack and a boolean array named as visited[ ];

Step 2.2: Mark all the vertices as not visited i.e., initialize visited[ ] with 'false' value.

Step 2.3: Call the recursive helper function topologicalSortUtil() to store Topological Sort starting from all vertices one by one.

Step 3: def topologicalSortUtil(int v, bool visited[],stack<int> &Stack):

Step 3.1: Mark the current node as visited.

Step 3.2: Recur for all the vertices adjacent to this vertex.

Step 3.3: Push current vertex to stack which stores result.

Step 4: Atlast after return from the utility function, print contents of stack.

Program:

```
import java.util.InputMismatchException;
import java.util.Scanner;
import java.util.Stack;
public class TopSort
{
private Stack<Integer> stack;
public TopSort()
{
stack = new Stack<Integer>();
}
public int[] topological(int adjacency_matrix[][], int source) throws NullPointerException
{
int number_of_nodes = adjacency_matrix[source].length - 1;
int[] topological_sort = new int[number_of_nodes + 1];
int pos = 1;
int j;
int visited[] = new int[number_of_nodes + 1];
int element = source;
int i = source;
visited[source] = 1;
stack.push(source);
while (!stack.isEmpty())
{
element = stack.peek();
while (i <= number_of_nodes)
{
if (adjacency_matrix[element][i] == 1 && visited[i] == 1)
{
if (stack.contains(i))
{
System.out.println("TOPOLOGICAL SORT NOT POSSIBLE");
return null;
}
}
if (adjacency_matrix[element][i] == 1 && visited[i] == 0)
{
stack.push(i);
visited[i] = 1;
element = i;
i = 1;
continue;
}
i++;
}
```

```
j = stack.pop();
topological_sort[pos++] = j;
i = ++j;
}
return topological_sort;
}
public static void main(String... arg)
{
int number_no_nodes, source;
Scanner scanner = null;
int topological_sort[] = null;
try
{
System.out.println("Enter the number of nodes in the graph");
scanner = new Scanner(System.in);
number_no_nodes = scanner.nextInt();
int adjacency_matrix[][] = new int[number_no_nodes + 1][number_no_nodes + 1];
System.out.println("Enter the adjacency matrix");
for (int i = 1; i <= number_no_nodes; i++)
for (int j = 1; j <= number_no_nodes; j++)
adjacency_matrix[i][j] = scanner.nextInt();
System.out.println("Enter the source for the graph");
source = scanner.nextInt();
System.out.println("The Topological sort for the graph is given by ");
TopSort toposort = new TopSort();
topological_sort = toposort.topological(adjacency_matrix, source);
for (int i = topological_sort.length - 1; i > 0; i--)
{
if (topological_sort[i] != 0)
System.out.print(topological_sort[i] + "\t");
}
}
catch (InputMismatchException inputMismatch)
{
System.out.println("Wrong Input format");
}
catch (NullPointerException nullPointer)
{
}
scanner.close();
}
}
```

Output:

```
D:\ AI Lab>javac TopSort.java
D:\AI Lab>java

Top Sort Enter the number of nodes in the graph
6
```

Enter the adjacency matrix

0

1

0

0

0

1

0

0

1

1

0

0

0

0

0

0

0

0

0

0

0

1

0

0

0

0

0

0

0

1

0

0

0

1

0

0

0

Enter the source for the graph

1

The Topological sort for the graph is given by

1 2 4 5 6 3

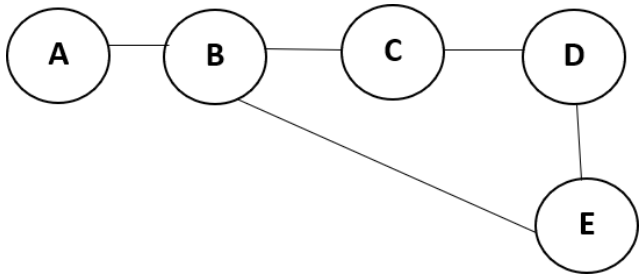
Result:

Thus, the topological sorting can be done successfully for the given graph and the output is verified.

Ex.No. 3 (b) Finding Shortest Path in Unweighted Graph using BFS Algorithm

Aim:

To find the shortest path for the given unweighted graph using Breadth First Search Algorithm.



Algorithm:

Breadth First Search:

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 (who's STATUS = 1) and set their STATUS = 2 (waiting state)

Step 6: EXIT

Shortest Path in Unweighted Graph:

Step1: Set all vertices distances = infinity except for the source vertex, set the source distance = 0.

Step2: Push the source vertex in a min-priority queue in the form (distance, vertex), as the comparison in the min-priority queue will be according to vertices distances.

Step3: Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).

Step4: Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.

Step5: If the popped vertex is visited before, just continue without using it.

Step6: Repeat step 2 to 5, until all the vertices are visited.

Program:

```
import java.util.*;
//Class representing graph nodes
class Node{
String name;
List<Node> neighbors;
boolean visited = false;
Node prev = null;
Node(String name){
this.name = name;
this.neighbors = new ArrayList<>();
}
//Method to connect nodes
void add_neighbor(Node node){
this.neighbors.add(node);
node.neighbors.add(this);
}
//Node representation
public String toString(){
return this.name;
}
}
//class implmenting the algorithm
class ShortestPath{
Node start, end;
ShortestPath(Node start, Node end){
this.start = start;
this.end = end;
}
public void bfs(){
//Create queue
Queue<Node> queue = new LinkedList<>();
//Visit and add start node to the queue
start.visited = true;
queue.add(start);
//BFS until queue is empty and not reached to the end node
while(!queue.isEmpty()){
//pop a node from queue for search operation
Node current_node = queue.poll();
//Loop through neighbors node to find the 'end'
for(Node node: current_node.neighbors){
```

```
if(!node.visited){
//Visit and add the node to the queue
node.visited  =true;
queue.add(node);
//update its precedings nodes
node.prev = current_node;
//If reached the end node then stop BFS
if(node==end){
queue.clear();
break;
}
}
}
}
trace_route();
}
//Function to trace the route using preceding nodes
private void trace_route(){
Node node = end;
List<Node> route = new ArrayList<>();
//Loop until node is null to reach start node
//becasue start.prev == null
while(node != null){
route.add(node);
node = node.prev;
}
//Reverse the route - bring start to the front
Collections.reverse(route);
//Output the route
System.out.println(route);
}
}
//Driver Code
class ShortPath {
public static void main(String[] args) {
//create nodes
Node node_A = new Node("A");
Node node_B = new Node("B");
Node node_C = new Node("C");
Node node_D = new Node("D");
Node node_E = new Node("E");
//connect nodes (i.e. create graph)
node_A.add_neighbor(node_B);
node_B.add_neighbor(node_C);
node_C.add_neighbor(node_D);
node_D.add_neighbor(node_E);
node_B.add_neighbor(node_E);
new ShortestPath(node_A, node_E).bfs();
}
}
```

Output:

```
D:\AI Lab>javac ShortPath.java

D:\AI Lab>java ShortPath

[A, B, E]
```

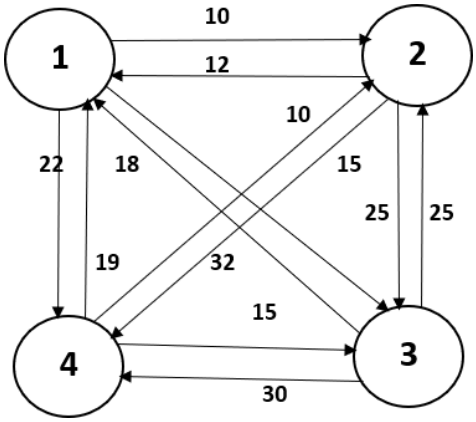

Result:

Thus, the shortest path can be identified in a graph and the output is verified.

Ex.No. 4 Implement A* and Memory Bounded A* Algorithm in Traveling Salesman Problem

Aim:

To implement TSP for the following graph using A* algorithm.



Algorithm:

A* Algorithm:

Step1: Place the starting node in the OPEN list.

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise

Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g (n') value.

Step 6: Return to **Step 2**.

Traveling Salesman Problem:

Step1: Consider city n as the starting and ending point. Since route is cyclic, we can consider any point as starting point.

Step2: Generate all (n-1)! Permutations of cities.

Step3: Calculate cost of every permutation and keep track of minimum cost permutation.

Step4: Return the permutation with minimum cost.

Program:

```
import  java.util.*;
import java.text.*;
class TSP
{
int weight[][],n,tour[],finalCost;
final int INF=1000;
public TSP()
{
Scanner s=new Scanner(System.in);
System.out.println("Enter no. of nodes:=>");
n=s.nextInt();
weight=new int[n][n];
tour=new int[n-1];
for(int i=0;i<n;i++)
{
for(int j=0;j<n;j++)
{
if(i!=j)
{
System.out.print("Enter weight of "+(i+1)+" to "+(j+1)+":=>");
weight[i][j]=s.nextInt();
}
}
}
System.out.println();
System.out.println("Starting node assumed to be node1.");
eval();
}
public int COST(int currentNode,int inputSet[],int setSize)
{
if(setSize==0)
return weight[currentNode][0];
int min=INF,minindex=0;
int setToBePassedOnToNextCallOfCOST[]=new int[n-1];

for(int i=0;i<setSize;i++)
{
int k=0;
//initialise new set
for(int j=0;j<setSize;j++)
{
if(inputSet[i]!=inputSet[j])
setToBePassedOnToNextCallOfCOST[k++]=inputSet[j];
}
int temp=COST(inputSet[i],setToBePassedOnToNextCallOfCOST,setSize-1);
if((weight[currentNode][inputSet[i]]+temp) < min)
{
min=weight[currentNode][inputSet[i]]+temp;
minindex=inputSet[i];
```

```
}
}
return min;
}
public int MIN(int currentNode,int inputSet[],int setSize)
{
if(setSize==0)
return weight[currentNode][0];
int min=INF,minindex=0;
int setToBePassedOnToNextCallOfCOST[]=new int[n-1];
for(int i=0;i<setSize;i++)
//considers each node of inputSet
{
int k=0;
for(int j=0;j<setSize;j++)
{
if(inputSet[i]!=inputSet[j])
setToBePassedOnToNextCallOfCOST[k++]=inputSet[j];
}
int temp=COST(inputSet[i],setToBePassedOnToNextCallOfCOST,setSize-1);
if((weight[currentNode][inputSet[i]]+temp) < min)
{
min=weight[currentNode][inputSet[i]]+temp;
minindex=inputSet[i];
}
}
return minindex;
}
public void eval()
{
int dummySet[]=new int[n-1];

for(int i=1;i<n;i++)
dummySet[i-1]=i;
finalCost=COST(0,dummySet,n-1);
constructTour();
}
public void constructTour()

{
int previousSet[]=new int[n-1];
int nextSet[]=new int[n-2];
for(int i=1;i<n;i++)
previousSet[i-1]=i;
int setSize=n-1;
tour[0]=MIN(0,previousSet,setSize);
for(int i=1;i<n-1;i++)
{
int k=0;
for(int j=0;j<setSize;j++)
{
if(tour[i-1]!=previousSet[j])
nextSet[k++]=previousSet[j];
}
```

```
--setSize;
tour[i]=MIN(tour[i-1],nextSet,setSize);
for(int j=0;j<setSize;j++)
previousSet[j]=nextSet[j];
}
display();
}
public void display()
{
System.out.println();
System.out.print("The tour is 1-");
for(int i=0;i<n-1;i++)
System.out.print((tour[i]+1)+"-");
System.out.print("1");
System.out.println();
System.out.println("The final cost is "+finalCost);
}
}
class TSPExp
{
public static void main(String args[])
{
TSP obj=new TSP();
}
}
```

Output:

```
D:\ AI Lab>javac TSPExp.java
D:\ AI Lab>java TSPExp

Enter no. of nodes: => 4

Enter weight of 1 to 2: =>10

Enter weight of 1 to 3: =>15

Enter weight of 1 to 4: =>22

Enter weight of 2 to 1: =>12

Enter weight of 2 to 3: =>25

Enter weight of 2 to 4: =>32

Enter weight of 3 to 1: =>18

Enter weight of 3 to 2: =>25

Enter weight of 3 to 4: =>30

Enter weight of 4 to 1: =>19

Enter weight of 4 to 2: =>10
```

Enter weight of 4 to 3: =>15

starting node assumed to be node1.

The tour is 1-3-4-2-1

The final cost is 67

Result:

Thus, the Traveling Salesman Problem can be solved with the help of A* algorithm and the output is verified

Ex.No. 5 **Implement Genetic Algorithms for AI tasks**

Aim:

To implement simple Genetic Algorithm.

Algorithm:

Step1: Initialize new population =0, n=length(x),c= 1 to n, x,y

Step2: Repeat step 3 until the size of the population reached maximum limit.

Step3: x=RANDOM-SELECTION(population, FITNESS-FN)

y=RANDOM-SELECTION(population, FITNESS-FN)

Child=REPRODUCE(x , y)

Step4: if (small random probability)

then child = MUTATE (child)

add child to new population

population = new population

Step5: Repeat Step4 until small random probability = 0

Step6: Return the best individual in population, according to fitness function

REPRODUCE(x, y) returns an individual

Step 7: Return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))

Program:

```
import java.util.Random;
//Main class
public class GeneAlgo {
Population population = new Population();
Individual fittest;
Individual secondFittest;
int generationCount = 0;
public static void main(String[] args) {
Random rn = new Random();
SimpleDemoGA demo = new SimpleDemoGA();
//Initialize population
demo.population.initializePopulation(10);
//Calculate fitness of each individual
demo.population.calculateFitness();
System.out.println("Generation: " + demo.generationCount + " Fittest: " + demo.population.fittest);
```

```
//While population gets an individual with maximum fitness
while (demo.population.fittest < 5) {
++demo.generationCount;
//Do selection
demo.selection();
//Do crossover
demo.crossover();
//Do mutation under a random probability
if (rn.nextInt()%7 < 5) {
demo.mutation();
}

//Add fittest offspring to population
demo.addFittestOffspring();
//Calculate new fitness value
demo.population.calculateFitness();
System.out.println("Generation: " + demo.generationCount + " Fittest: " + demo.population.fittest);
}
System.out.println("\nSolution found in generation " + demo.generationCount);
System.out.println("Fitness: "+demo.population.getFittest().fitness);
System.out.print("Genes: ");
for (int i = 0; i < 5; i++) {
System.out.print(demo.population.getFittest().genes[i]);
}
System.out.println("");
}
//Selection
void selection() {

//Select the most fittest individual
fittest = population.getFittest();
//Select the second most fittest individual
secondFittest = population.getSecondFittest();
}
//Crossover
void crossover() {
Random rn = new Random();
//Select a random crossover point
int crossOverPoint = rn.nextInt(population.individuals[0].geneLength);
//Swap values among parents
for (int i = 0; i < crossOverPoint; i++) {
int temp = fittest.genes[i];
fittest.genes[i] = secondFittest.genes[i];
secondFittest.genes[i] = temp;
}
}
```

```
//Mutation
    void mutation() {
    Random rn = new Random();
    //Select a random mutation point
    int mutationPoint = rn.nextInt(population.individuals[0].geneLength);
    //Flip values at the mutation point
    if (fittest.genes[mutationPoint] == 0) {
    fittest.genes[mutationPoint] = 1;
    } else {
    fittest.genes[mutationPoint] = 0;
    }
    mutationPoint = rn.nextInt(population.individuals[0].geneLength);
    if (secondFittest.genes[mutationPoint] == 0) {
    secondFittest.genes[mutationPoint] = 1;
    } else {
    secondFittest.genes[mutationPoint] = 0;
    }
    }
    //Get fittest offspring
    Individual getFittestOffspring() {
    if (fittest.fitness > secondFittest.fitness) {
    return fittest;
    }
    return secondFittest;
    }
    //Replace least fittest individual from most fittest offspring
    void addFittestOffspring() {

    //Update fitness values of offspring
    fittest.calcFitness();
    secondFittest.calcFitness();
    //Get index of least fit individual
    int leastFittestIndex = population.getLeastFittestIndex();
    //Replace least fittest individual from most fittest offspring
    population.individuals[leastFittestIndex] = getFittestOffspring();
    }
    }
    //Individual  class
    class Individual {
    int fitness = 0;
    int[] genes = new int[5];
    int geneLength = 5;
    public Individual() {
    Random rn = new Random();
    //Set genes randomly for each individual
    for (int i = 0; i < genes.length; i++) {
    genes[i] = Math.abs(rn.nextInt() % 2);
    }
```

```
fitness = 0;
}
//Calculate fitness
public void calcFitness() {
fitness = 0;
for (int i = 0; i < 5; i++) {
if (genes[i] == 1) {
++fitness;
}
}
}
}
//Population  class
class Population {
int popSize = 10;
Individual[] individuals = new Individual[10];
int fittest = 0;
//Initialize population
public void initializePopulation(int size) {
for (int i = 0; i < individuals.length; i++) {
individuals[i] = new Individual();
}
}
//Get the fittest individual

public Individual getFittest() {
int maxFit = Integer.MIN_VALUE;
int maxFitIndex = 0;
for (int i = 0; i < individuals.length; i++) {
if (maxFit <= individuals[i].fitness) {
maxFit = individuals[i].fitness;
maxFitIndex = i;
}
}
fittest = individuals[maxFitIndex].fitness;
return individuals[maxFitIndex];
}
//Get the second most fittest individual
public Individual getSecondFittest() {
int maxFit1 = 0;
int maxFit2 = 0;
for (int i = 0; i < individuals.length; i++) {
if (individuals[i].fitness > individuals[maxFit1].fitness) {
maxFit2 = maxFit1;
maxFit1 = i;
} else if (individuals[i].fitness > individuals [maxFit2].fitness) {
maxFit2 = i;
}
```

```
    }
    return individuals[maxFit2];
    }
    //Get index of least fittest individual
    public int getLeastFittestIndex() {
    int minFitVal = Integer.MAX_VALUE;
    int minFitIndex = 0;
    for (int i = 0; i < individuals.length; i++) {
    if (minFitVal >= individuals[i].fitness) {
    minFitVal = individuals[i].fitness;
    minFitIndex = i;
    }
    }
    return minFitIndex;
    }
    //Calculate fitness of each individual
    public void calculateFitness() {
    for (int i = 0; i < individuals.length; i++) {
    individuals[i].calcFitness();
    }
getFittest();}}
```

Output:

```
D:\AI Lab>javac GeneAlgo.java
D:\AI Lab>java GeneAlgo Generation: 0 Fittest: 4
Generation: 1 Fittest: 3
Generation: 2 Fittest: 3
Generation: 3 Fittest: 4
Generation: 4 Fittest: 4
Generation: 5 Fittest: 4
Generation: 6 Fittest: 3
Generation: 7 Fittest: 3
Generation: 8 Fittest: 5

Solution found in generation 8
Fitness: 5
Genes: 11111
```

Result:

Thus, the genetic algorithms implemented successfully and the output is verified.

Ex.No.6 **Implementation of Traveling Salesman Problem using Simulated Annealing Algorithms**

Aim:

To implement TSP for finding shortest distance between two cities using Simulated Annealing algorithms.

Algorithm:

Simulated Annealing

Step1: Generating an initial solution x0;

Step2: xbest ← x0;

Step3: Computing the value of the objective function f(x0) and f(xb est);

Step4: Ti ← T0;

Step5: while Ti>Tmin do

Step6: $\Delta f \leftarrow f(xnew) - f(xbest)$;

Step7: if $\Delta f < 0$ then

Step8: xbest ← xnew;

Step9: end if

Step10: if $\Delta f \geq 0$ then

Step11: $p \leftarrow e^{\Delta f T}$;

Step12: if c ← random[0, 1] ≥ p then

Step13: xbest ← xnew;

Step14: else

Step15: xbest ← xbest;

Step16: end if

Step17: end if

Step18: i ← i + 1;

Step19: Ti ← ρ × Ti ;

Step20: end while

Step21: Return xbest;

Algorithm :

City.java

Step1: Create a java file for city

Step2: With the help of this operator, initialize a random number

Step3: Get the X and Y coordinate of the city

Step4: Calculate the distance of X and Y coordinate of the city

Step5: Calculate the distance by using the following formula:

distance = sqrt ((xdistance^2) – (ydistance)^2)

Step6: Get the result.

Algorithm:

TourManager.java

Step1: Create a java file for TourManager

Step2: Create an object to store the names of various cities.

Step3: Add the destination cities

Step4: Get a city to calculate the distance.

Step5: Get the destination city

Algorithm:

Tour.java

Step1: Create a java file for tour

Step2: Get the tour of cities of a person

Step3: Construct a tour for a person

Step4: Get the city

Step5: Calculate the tour distance between the cities.

Algorithm:

SimulatedAnnealing.java

Step1: Create the main file for TSP

Step2: Calculate the acceptance probability

Step3: If the new solution is better, accept it otherwise, calculate an acceptance probability

Step4: Create and add our cities

Step5: Set the initial temperature as solution

Step6: Set as current best and Loop until system has cooled

Step7: Get a random position in the tour

Step8: Choose the best solution

Program:

city.java

```
package sa;
public class City {
int x;
int y;
// Constructs a randomly placed city
public City(){
```

```
this.x = (int)(Math.random()*200);
this.y = (int)(Math.random()*200);
}
// Constructs a city at chosen x, y location
public City(int x, int y){
this.x = x;
this.y = y;
}
// Gets city's x coordinate
public int getX(){
return this.x;
}
// Gets city's y coordinate
public int getY(){
return this.y;
}
// Gets the distance to given city
public double distanceTo(City city){
int xDistance = Math.abs(getX() - city.getX());
int yDistance = Math.abs(getY() - city.getY());
double distance = Math.sqrt( (xDistance*xDistance) + (yDistance*yDistance) );
return distance;
}
@Override
public String toString(){
return getX()+", "+getY();
}
}
```


TourManager.java

```
/*
* TourManager.java
* Holds the cities of a tour
*/

package sa;
import java.util.ArrayList;
public class TourManager {
// Holds our cities
private static ArrayList destinationCities = new ArrayList<City>();
// Adds a destination city
public static void addCity(City city) {
destinationCities.add(city);
}
// Get a city
public static City getCity(int index){
return (city)destinationCities.get(index);
}
// Get the number of destination cities
public static int numberOfCities(){
return destinationCities.size();
}
}
```

Tour.java

```
/*
* Tour.java
* Stores a candidate tour through all cities
*/
package sa;
import java.util.ArrayList;
import java.util.Collections;
public class Tour{
// Holds our tour of cities
private ArrayList tour = new ArrayList<City>();
// Cache
private int distance = 0;
// Constructs a blank tour
public Tour(){
for (int i = 0;i<TourManager.numberOfCities();i++) {
tour.add(null);
}
}
// Constructs a tour from another tour
public Tour(ArrayList tour){
this.tour = (ArrayList) tour.clone();
}
// Returns tour information
public ArrayList getTour(){
return tour;
}
// Creates a random individual
public void generateIndividual() {
// Loop through all our destination cities and add them to our tour
for (int cityIndex = 0; cityIndex < TourManager.numberOfCities(); cityIndex++) {
setCity(cityIndex, TourManager.getCity(cityIndex));
}
// Randomly reorder the tour
Collections.shuffle(tour);
}
// Gets a city from the tour
public City getCity(int tourPosition) {
return (City)tour.get(tourPosition);
}
// Sets a city in a certain position within a tour
public void setCity(int tourPosition, City city) {
tour.set(tourPosition, city);
// If the tours been altered we need to reset the fitness and distance
distance = 0;
}
// Gets the total distance of the tour
public int getDistance(){
if (distance == 0) {
int tourDistance = 0;
// Loop through our tour's cities
for (int cityIndex=0; cityIndex < tourSize(); cityIndex++) {
```

```
// Get city we're traveling from
City fromCity = getCity(cityIndex);
// City we're traveling to
City destinationCity;
// Check we're not on our tour's last city, if we are set our
// tour's final destination city to our starting city
if(cityIndex+1 < tourSize()){
destinationCity = getCity(cityIndex+1);
}
else{
destinationCity = getCity(0);
}
// Get the distance between the two cities
tourDistance += fromCity.distanceTo(destinationCity);
}
distance = tourDistance;
}
return distance;
}
// Get number of cities on our tour
public int tourSize() {
return tour.size();
}
@Override
public String toString() {
String geneString = "|";
for (int i = 0; i < tourSize(); i++) {
geneString += getCity(i)+"|";
}
return geneString;
}}
```

SimulatedAnnealing.java

```
package sa;
public class SimulatedAnnealing {
// Calculate the acceptance probability
public static double acceptanceProbability(int energy, int newEnergy, double temperature) {
// If the new solution is better, accept it
if (newEnergy < energy) {
return 1.0;
}
// If the new solution is worse, calculate an acceptance probability
return Math.exp((energy - newEnergy) / temperature);
}
public static void main(String[] args) {
// Create and add our cities
City city = new City(60, 200);
TourManager.addCity(city);
City city2 = new City(180, 200);
TourManager.addCity(city2);
City city3 = new City(80, 180);
TourManager.addCity(city3);
City city4 = new City(140, 180);
 TourManager.addCity(city4);
```

```
City city5 = new City(20, 160);
TourManager.addCity(city5);
City city6 = new City(100, 160);
TourManager.addCity(city6);
City city7 = new City(200, 160);
TourManager.addCity(city7);
City city8 = new City(140, 140);
TourManager.addCity(city8);
City city9 = new City(40, 120);
TourManager.addCity(city9);
City city10 = new City(100, 120);
TourManager.addCity(city10);
City city11 = new City(180, 100);
TourManager.addCity(city11);
City city12 = new City(60, 80);
TourManager.addCity(city12);
City city13 = new City(120, 80);
TourManager.addCity(city13);
City city14 = new City(180, 60);
TourManager.addCity(city14);
City city15 = new City(20, 40);
TourManager.addCity(city15);
City city16 = new City(100, 40);
TourManager.addCity(city16);
City city17 = new City(200, 40);
TourManager.addCity(city17);
City city18 = new City(20, 20);

TourManager.addCity(city18);
City city19 = new City(60, 20);
TourManager.addCity(city19);
City city20 = new City(160, 20);
TourManager.addCity(city20);
// Set initial temp
double temp = 10000;
// Cooling rate
double coolingRate = 0.003;
// Initialize intial solution
Tour currentSolution = new Tour();
currentSolution.generateIndividual();
System.out.println("Initial solution distance: " + currentSolution.getDistance());
// Set as current best
Tour best = new Tour(currentSolution.getTour());
// Loop until system has cooled
while (temp > 1) {
// Create new neighbour tour
Tour newSolution = new Tour(currentSolution.getTour());
// Get a random positions in the tour
int tourPos1 = (int) (newSolution.tourSize() * Math.random());
int tourPos2 = (int) (newSolution.tourSize() * Math.random());
// Get the cities at selected positions in the tour
City citySwap1 = newSolution.getCity(tourPos1);
City citySwap2 = newSolution.getCity(tourPos2);
```

```
// Swap them
newSolution.setCity(tourPos2, citySwap1);
newSolution.setCity(tourPos1, citySwap2);
// Get energy of solutions
int currentEnergy = currentSolution.getDistance();
int neighbourEnergy = newSolution.getDistance();
// Decide if we should accept the neighbour
if (acceptanceProbability(currentEnergy, neighbourEnergy, temp) > Math.random()) {
currentSolution = new Tour(newSolution.getTour());
}
if (currentSolution.getDistance() < best.getDistance()) {
best = new Tour(currentSolution.getTour());
}
temp *= 1-coolingRate;
}
System.out.println("Final solution distance: " + best.getDistance());
System.out.println("Tour: " + best);
}}
```

Output:

```
D:\AI Lab>javac SimulatedAnnealing.java

D:\AI Lab>java SimulatedAnnealing

Initial solution distance: 1966
Final solution distance: 911
Tour: |180, 200|200, 160|140, 140|180, 100|180, 60|200, 40|160, 20|120, 80|100, 40|60, 20|20,
20|20, 40|60, 80|100, 120|40, 120|20, 160|60, 200|80, 180|100, 160|140, 180|
```

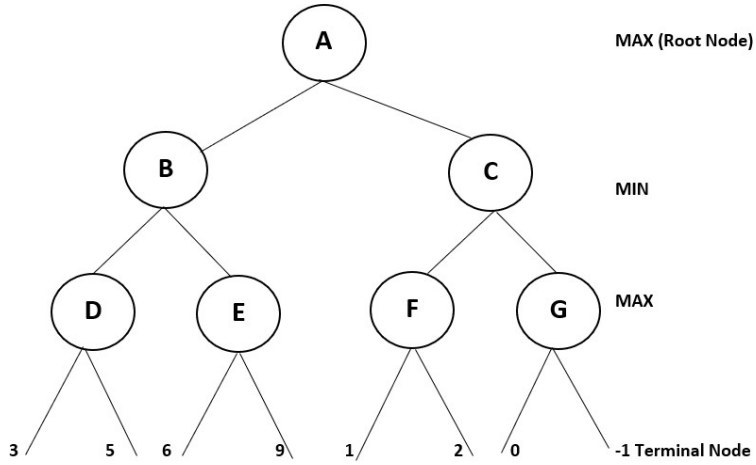
Result:

Thus, TSP was implemented successfully using Simulated Annealing and the output is verified.

Ex.No. 7 **Implementation of Alpha-Beta Search**

Aim:

To find the optimal solution for the given game tree using Alpha-Beta search algorithm.



Algorithm:

Step1: Initialize node=n, depth=m, alpha=-∞ beta= +∞

Step2: If node is a leaf node return value of the node, exit.

Step3: If is MaximizingPlayer then bestVal = - ∞

Step4: for each child node calculate the following values:
(i) value = minimax(node, depth+1, false, alpha, beta)
(ii) bestVal = max(bestVal, value)
(iii) alpha = max(alpha, bestVal)

Step5: Return step 4 until if beta <= alpha, and return bestVal

Step6: Otherwise bestVal = +∞

Step7: for each child node calculate the following values:
(i) value = minimax(node, depth+1, true, alpha, beta)
(ii) bestVal = min(bestVal, value)
(iii) beta = min(beta, bestVal)

Step8: if beta <= alpha, stop the program and return bestVal

Program:

```
import java.io.*;
class AlphaBeta {
// Initial values of Alpha and Beta
static int MAX = 1000;
static int MIN = -1000;
// Returns optimal value for current player (Initially called for root and maximizer)
static int minimax(int depth, int nodeIndex,Boolean maximizingPlayer,int values[], int alpha,int
beta)
{
// Terminating condition. i.e leaf node is reached
if (depth == 3)
return values[nodeIndex];
if (maximizingPlayer)
{
int best = MIN;
// Recur for left and right children
for (int i = 0; i < 2; i++){
int val = minimax(depth + 1, nodeIndex * 2 + i,false, values, alpha, beta);
best = Math.max(best, val);
alpha = Math.max(alpha, best);
// Alpha Beta Pruning
if (beta <= alpha)
break;
}
return best;
}
else
{
int best = MAX;
// Recur for left and right children
for (int i = 0; i < 2; i++)
{
int val = minimax(depth + 1, nodeIndex * 2 + i,true, values, alpha, beta);
best = Math.min(best, val);
beta = Math.min(beta, best);
// Alpha Beta Pruning
if (beta <= alpha)
break;
}
return best;
}
}
// Driver Code
public static void main (String[] args)
{
int values[] = {3, 5, 6, 9, 1, 2, 0, -1};
System.out.println("The optimal value is : " +minimax(0, 0, true, values, MIN, MAX));
}}
```

Output:

D:\AI Lab>javac AlphaBeta.java

D:\ AI Lab>java AlphaBeta

The optimal value is: 5

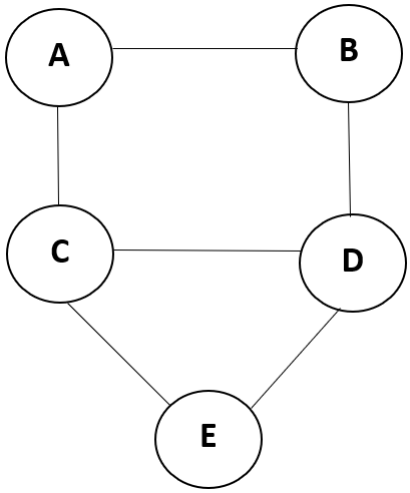
Result:

Thus, Alpha beta search algorithm was implemented successfully and the output is verified.

Ex.No.8 Implement Graph Coloring Problem using Backtracking Algorithm

Aim:

To solve the graph coloring problem for the following graph using backtracking search algorithm.



Algorithm:

Step1: Confirm whether it is valid to color the current vertex with the current color (by checking whether any of its adjacent vertices are colored with the same color).

If yes then color it and otherwise try a different color.

Check if all vertices are colored or not.

If not then move to the next adjacent uncolored vertex.

Step2: If no other color is available then backtrack (i.e. un-color last colored vertex).

Here backtracking means to stop further recursive calls on adjacent vertices by returning false.

Step3: If the same color for current vertex, then goto step 1.2 otherwise goto step 2.

Step4: Repeat step 2 and 3 until all the vertex to be colored.

Program:

```
import java.util.ArrayList;
import java.util.List;
class Vertex {
String name;
List<Vertex> adjacentVertices;
boolean colored;
String color;
public Vertex(String name) {
this.name = name;
```

```
this.adjacentVertices = new ArrayList<>();
this.colored =false;
this.color = "";
}
//connect two verticess bi-directional
public void addNeighbor(Vertex vertex){
this.adjacentVertices.add(vertex);
vertex.adjacentVertices.add(this);
}
}
class Coloring {
String colors[];
int colorCount;
int numberOfVertices;
public Coloring(String[] colors, int N) {
this.colors = colors;
this.numberOfVertices = N;
}
public boolean setColors(Vertex vertex){
//Step: 1
for(int colorIndex=0; colorIndex<colors.length; colorIndex++){
//Step-1.1: checking validity
if(!canColorWith(colorIndex, vertex))
continue;
//Step-1.2: continue coloring
vertex.color=colors[colorIndex];
vertex.colored=true;
colorCount++;
//Step-1.3: check whether all vertices colored?
if(colorCount== numberOfVertices) //base case
return true;
//Step-1.4: next uncolored vertex
for(Vertex nbrvertex: vertex.adjacentVertices){
if  (!nbrvertex.colored){
if(setColors(nbrvertex))
return true;
}
```

```
}
}
//Step-4: backtrack
vertex.colored = false;
vertex.color = "";
return false;
}
//Function to check whether it is valid to color with color[colorIndex]
boolean canColorWith(int colorIndex, Vertex vertex) {
for(Vertex nbrvertex: vertex.adjacentVertices){
if(nbrvertex.colored && nbrvertex.color.equals(colors[colorIndex]))
return  false;
}
return true;
}
}
public class CSPSolver{
public static void main(String args[]){
//define vertices
Vertex vertices [] = {new Vertex("A"), new Vertex("B"), new Vertex("C"), new Vertex("D"),new
Vertex("E")};
//join vertices
vertices[0].addNeighbor(vertices[1]);
vertices[1].addNeighbor(vertices[3]);
vertices[3].addNeighbor(vertices[2]);
vertices[2].addNeighbor(vertices[0]);
vertices[2].addNeighbor(vertices[4]);
vertices[4].addNeighbor(vertices[3]);
//define colors
String colors[] = {"Green","Blue","Red"};
//create coloring object
Coloring coloring = new Coloring(colors, vertices.length);
//start coloring with vertex-A
boolean hasSolution = coloring.setColors(vertices[0]);
//check if coloring was successful
if (!hasSolution)
```

```
System.out.println("No Solution");
else {
for (Vertex vertex: vertices) {
System.out.println(vertex.name + " "+ vertex.color +"\n");
}}}}
```

Output:

```
D:\AI Lab>javac CSPSolver.java
D:\AI Lab>java CSPSolver

A Green

B Blue

C Blue

D Green

E Red
```

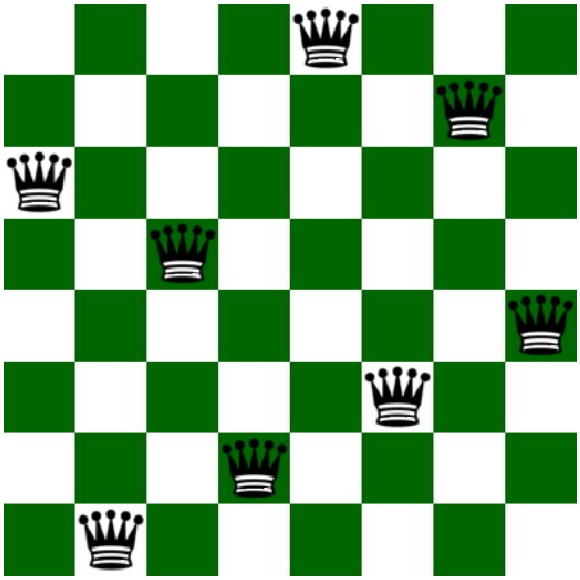

Result:

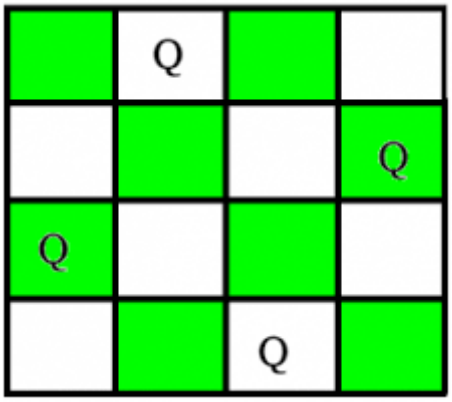
Thus, the graph coloring problem was implemented successfully and the output is verified successful

Ex.No.9 **Implement N Queens Problem using Random Restart Hill Climbing Algorithm**

Aim:

To solve N-Queens problem with the help of Random Restart Hill Climbing algorithm.





Algorithm:

Random Restart Hill Climbing:

Step1: Initial the current state

Step2: Get the solution and if it not good enough has not been found do a randomized restart.

Step3: Calculate Heuristic Evaluation Function for the current state

Step 3.1: If the current state is best solution, then stop.

Step 3.2: Otherwise, Move to the next best possible state

Step4: If current and best are STILL the same, then we reached a peak.

Step5: If best is equal to current, then return the best solution.

Program:

NQueen.java

```
public class NQueen
{
private int row;
private int column;
public NQueen(int row, int column)
{
this.row = row;
this.column = column;
}
public void move ()
```

```
{
row++;
}
public boolean ifConflict(NQueen q)
{
//  Check rows and columns
if(row == q.getRow() || column == q.getColumn())
return true;
//  Check diagonals
else if(Math.abs(column-q.getColumn()) == Math.abs(row-q.getRow()))
return true;
return false;
}
public int getRow()
{
return row;
}
public int getColumn()
{
return column;
}
}
```

HillClimbingRestart.java

```
import java.util.Scanner;
import java.util.Random;
public class HillClimbingRandomRestart {
private static int n ;
private static int stepsClimbedAfterLastRestart = 0;
private static int stepsClimbed =0;
private static int heuristic = 0;
private static int randomRestarts = 0;
//Method to create a new random board
public static NQueen[] generateBoard() {
NQueen[] startBoard = new NQueen[n];
Random rndm = new Random();
for(int i=0; i<n; i++){
startBoard[i] = new NQueen(rndm.nextInt(n), i);
}
return startBoard;
}
//Method to print the Current State
private static void printState (NQueen[] state) {
//Creating temporary board from the present board
int[][] tempBoard = new int[n][n];
for (int i=0; i<n; i++) {
//Get the positions of Queen from the Present board and set those positions as 1 in temp board
tempBoard[state[i].getRow()][state[i].getColumn()]=1;
}
System.out.println();
for (int i=0; i<n; i++) {
for (int j= 0; j < n; j++) {
System.out.print(tempBoard[i][j] + " ");
}
```

```
System.out.println();
}
}
// Method to find Heuristics of a state
public static int findHeuristic (NQueen[] state) {
int heuristic = 0;
for (int i = 0; i< state.length; i++) {
for (int j=i+1; j<state.length; j++ ) {
if (state[i].ifConflict(state[j])) {
heuristic++;
}
}
}
return heuristic;
}
// Method to get the next board with lower heuristic
public static NQueen[] nextBoard (NQueen[] presentBoard) {
NQueen[] nextBoard = new NQueen[n];
NQueen[] tmpBoard = new NQueen[n];
int presentHeuristic = findHeuristic(presentBoard);
int bestHeuristic = presentHeuristic;
int tempH;
for (int i=0; i<n; i++) {
//  Copy present board as best board and temp board
nextBoard[i] = new NQueen(presentBoard[i].getRow(), presentBoard[i].getColumn());
tmpBoard[i] = nextBoard[i];
}
// Iterate each column
for (int i=0; i<n; i++) {
if (i>0)
tmpBoard[i-1] = new NQueen (presentBoard[i-1].getRow(), presentBoard[i-1].getColumn());
tmpBoard[i] = new NQueen (0, tmpBoard[i].getColumn());
//  Iterate each row
for (int j=0; j<n; j++) {
//Get the heuristic
tempH = findHeuristic(tmpBoard);
//Check if temp board better than best board
if (tempH < bestHeuristic) {
bestHeuristic = tempH;
// Copy the temp board as best board
for (int k=0; k<n; k++) {
nextBoard[k] = new NQueen(tmpBoard[k].getRow(), tmpBoard[k].getColumn());
}
}
//Move the queen
if (tmpBoard[i].getRow()!=n-1)
tmpBoard[i].move();
}
}
//Check whether the present bord and the best board found have same heuristic
//Then randomly generate new board and assign it to best board
if (bestHeuristic == presentHeuristic) {
randomRestarts++;
stepsClimbedAfterLastRestart = 0;
```

```
nextBoard = generateBoard();
heuristic = findHeuristic(nextBoard);
} else
heuristic = bestHeuristic;
stepsClimbed++;
stepsClimbedAfterLastRestart++;
return nextBoard;
}
public static void main(String[] args) {
int presentHeuristic;
Scanner s=new Scanner(System.in);
while (true){
System.out.println("Enter the number of Queens :");
n = s.nextInt();
if ( n == 2 || n ==3) {
System.out.println("No Solution possible for "+n+" Queens. Please enter another number");
}
else
break;
}
System.out.println("Solution to "+n+" queens using hill climbing with random restart:");
//Creating the initial Board
NQueen[] presentBoard = generateBoard();
presentHeuristic = findHeuristic(presentBoard);
// test if the present board is the solution board
while (presentHeuristic != 0) {
//  Get the next board
// printState(presentBoard);
presentBoard = nextBoard(presentBoard);
presentHeuristic  = heuristic;
}
//Printing the solution
printState(presentBoard);
System.out.println("\nTotal number of Steps Climbed: " + stepsClimbed);
System.out.println("Number of random restarts: " + randomRestarts);
System.out.println("Steps Climbed after last restart: " + stepsClimbedAfterLastRestart);
}
}
```

Output:

```
D:\AI Lab>javac HillClimbingRandomRestart.java

D:\AI Lab>java HillClimbingRandomRestart

Enter the number of Queens:

8

Solution to 8 queens using hill climbing with random restart:

0 0 0 0 0 1 0 0

0 0 0 1 0 0 0 0
```

0 1 0 0 0 0 0 0

0 0 0 0 0 0 0 1

0 0 0 0 1 0 0 0

0 0 0 0 0 0 1 0

1 0 0 0 0 0 0 0

0 0 1 0 0 0 0 0

Total number of Steps Climbed: 32

Number of random restarts: 7

Steps Climbed after last restart: 7

Result:

Thus, the N Queen problem is implemented successfully using Hill climbing algorithm and the output is verified.

Ex.No.10 **Implement Propositional Logic**

Aim

To represent formulas of propositional logic and calculate the truth value for the given terms.

Algorithm:

Step1: Create an interface **Wff** to get the convert the input as logical values.

Step2: Create a class as **PropConstant** to convert the input as string.

Step3: Create a class as **Valuation** to evaluate the Boolean values.

Step4: Create a class as **Operator** to get the logical operators for evaluation.

Step5: Create a main class as **PropLogicLauncher** to implement the propositional logic.

Program:

Wff.java

```
public interface Wff
{
public boolean eval(Valuation val);
public String toString();
}
```


PropConstant.java

```
public class PropConstant implements Wff {
private String propConstant;
public PropConstant(String str) {
this.propConstant = str;
}
public String toString() {
return propConstant;
}
public boolean eval(Valuation val) {
return val.get(this);
}
}
```

Valuation.java

```
import java.util.HashMap;
public class Valuation {
HashMap<PropConstant, Boolean> val = new HashMap<PropConstant, Boolean>();
public boolean get(PropConstant propConstant) {
return val.get(propConstant);}

public Boolean put(PropConstant propConstant, boolean tv) {
return val.put(propConstant, tv);
}
}
```

Operator.java

```
public enum Operator {
NEG("~"), AND("&"), OR("|"), IMP("->"), IFF("<->");
private String symbol;
Operator(String symbol)
{
this.symbol = symbol;
}
public String toString() {
return symbol;
}}
```

PropLogicLauncher.java

```
public class PropLogicLauncher {
public static void main(String[] args) {
// Make some new constants
PropConstant p = new PropConstant("P");
PropConstant q = new PropConstant("Q");
PropConstant r = new PropConstant("R");
// Build some Wffs
Wff e0 = new NotWff(p);
Wff e1 = new AndWff(q, e0);
Wff e2 = new OrWff(e1, p);
Wff e3 = new IfWff(e1, p);
Wff e4 = new NotWff(e2);
// What does their toString() method produce?
System.out.println("Display form of Wff e0 is: " + e0);
System.out.println("Display form of Wff e1 is: " + e1);
System.out.println("Display form of Wff e2 is: " + e2);
System.out.println("Display form of Wff e3 is: " + e3);
System.out.println("Display form of Wff e4 is: " + e4);
System.out.println();
// Create a Valuation and set some truth values
Valuation val = new Valuation();
val.put(p, true);
val.put(q, false);
val.put(r, true);
```

```
// Compute the truth values and display the results
System.out.println("The value of Wff e0 is: " + e0.eval(val));
System.out.println("The value of Wff e1 is: " + e1.eval(val));
System.out.println("The value of Wff e2 is: " + e2.eval(val));
System.out.println("The value of Wff e3 is: " + e3.eval(val));
System.out.println("The value of Wff e4 is: " + e4.eval(val));
}
}
```

Output:

```
D:\AI Lab>javac PropLogicLauncher.java

D:\ AI Lab>java PropLogicLauncher

Display form of Wff e0 is: ~P
Display form of Wff e1 is: (Q & ~P)
Display form of Wff e2 is: ((Q & ~P) | P)
Display form of Wff e3 is: ((Q & ~P) -> P)
Display form of Wff e4 is: ~((Q & ~P) | P)

The value of Wff e0 is: false
The value of Wff e1 is: false
The value of Wff e2 is: true
The value of Wff e3 is: true
The value of Wff e4 is: false
```


Result:

Thus, the propositional logic implemented and the output is verified.

Ex.No.11 **Implement First Order Logic**

Aim:

To represent formulas of First Order Logic (FOL) and calculate the truth value for the given terms.

Algorithm:

Step1: Create a constant class to get the constant.

Step2: Create a variable class to get the variables in the terms.

Step3: Create a simplesentence class to find the sentences in the logic

Step4: Create a substitution class to convert the simple English sentences into FOL.

Program:

Constant.java

```
public class Constant implements Unifiable
{
private String printName = null;
private static int nextId = 1;
private int id;
public Constant()
{
this.id = nextId++;
}
public Constant(String printName)
{
this();
this. printName= printName;
}
public String toString()
{
if (printName!= null)
return printName;
return "constant_" + id;
}
}
```

Variable.java

```
public class Variable implements Unifiable
{
private String printName = null;
private static int nextId = 1;
private int id;
public Variable ()

{
this.id = nextId++;
}
public Variable(String printName)
{
this();
this.printName = printName;
}
public String toString()
{
if (printName != null)
return printName + "_" + id;
return "V" + id;
}
```

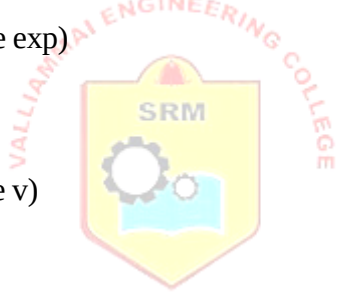
SimpleSentence.java

```
public class SimpleSentence implements Unifiable
{
private Unifiable[] terms;
public SimpleSentence(Constant predicateName,Unifiable... args)
{
this.terms = new Unifiable[args.length + 1];
terms[0] = predicateName;
System.arraycopy(args, 0, terms, 1,args.length);
}
private SimpleSentence(Unifiable... args)
{
terms = args
}
public String toString()
{
String s = null;
for (Unifiable p : terms)
if (s == null)
s = p.toString();
else
s += " " + p;
if (s == null)
return "null";
return "(" + s + ")";
}
```

```
public int length()
{
return terms.length;
}
public Unifiable getTerm(int index)
{
return terms[index];
}
```

SubstitutionSet.java

```
public class SubstitutionSet
{
private HashMap<Variable, Unifiable> bindings =new HashMap<Variable, Unifiable>();
public SubstitutionSet(){}
public SubstitutionSet(SubstitutionSet s)
{ this.bindings =new HashMap<Variable,
Unifiable>(s.bindings);
}
public void clear()
{
bindings.clear();
}
public void add(Variable v, Unifiable exp)
{
bindings.put(v, exp);
}
public Unifiable getBinding(Variable v)
{
return (Unifiable)bindings.get(v);
}
public boolean isBound(Variable v)
{
return bindings.get(v) != null;
}
public String toString()
{
return "Bindings:[" + bindings + "]";
}
```



Output:

```
D:\ AI Lab>javac SimpleSentence.java

D:\ AI Lab>java SimpleSentence

Goal = (friend X_1 Y_2)
(friend bill george)
(friend bill kate)
(friend bill merry)
(friend george bill)
```

(friend george kate)
(friend kate merry)
Goal = (friend bill Y_2)
(friend bill george)
(friend bill kate)
(friend bill merry)
False
False
False

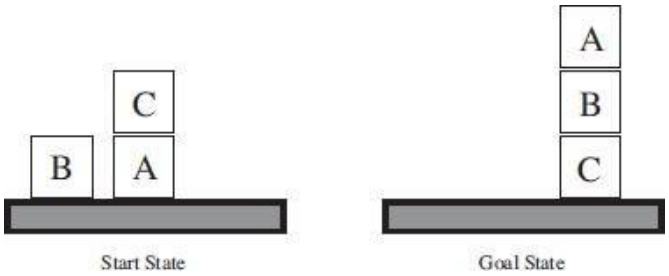
Result:

Thus, the First Order Logic was implemented successfully and the output is verified.

Ex.No. 12 Implement Block World problem using Classical Planning algorithms

Aim:

To implement Block World problem using Classical Planning algorithms.



Algorithm:

Step1: Initial that the blocks A on table, B on A and C on B.

Step2: Clear the blocks B and C

Step3: Apply the move action as (b,x,y)

Step4: Define the preconditions as On(b, x) ∧ Clear (b) ∧ Clear (y) ∧ Block (b) ∧ Block (y) ∧ (b_=x) ∧ (b_=y) ∧ (x_=y)

Step5: Define the EFFECT as On(b, y) ∧ Clear (x) ∧ ¬On(b, x) ∧ ¬Clear (y))

Step6: Apply the Action(MoveToTable (b, x))

Step7: Repeat the steps 3 to 6, until goal state is reached.

Program:

```
public class ClassicalPlanning
{
public static void main(String[] args) throws PlPlanException
{
System.out.println(&quot-- TEST1 --&quot);
PLPlan planner = new PLPlan();
planner.setAlgorithm(EnumAlgorithm.GRAPHPLAN);
planner.addFact("td");
planner.addFact("ocd");
planner.addFact("obc");
planner.addFact("oab");
planner.addFact("na");
planner.addGoalFact("oca");
planner.addGoalFact("odb");
planner.addGoalFact("ta");
planner.addGoalFact("tb");
planner.addGoalFact("nc");
planner.addGoalFact("nd");
List<String> precond = new ArrayList<String>();
precond.add("na");
precond.add("oab");
List<String> neg = new ArrayList<String>();
neg.add("oab");
List<String> pos = new ArrayList<String>();
pos.add("ta");
pos.add("nb");
planner.addOperator("uAB",precond, neg, pos);
precond = new ArrayList<String>();
precond.add("nb");
precond.add("obc");
neg = new ArrayList<String>();
neg.add("obc");
pos = new ArrayList<String>();
pos.add("tb");
pos.add("nc");
planner.addOperator("uBC",precond, neg, pos);
precond = new ArrayList<String>();
precond.add("nc");
precond.add("ocd");
neg = new ArrayList<String>();
neg.add("ocd");
pos = new ArrayList<String>();
pos.add("tc");
pos.add("nd");
planner.addOperator("uCD",precond, neg, pos);
precond = new ArrayList<String>();
precond.add("na");
precond.add("tc");
precond.add("nc");
```

```
neg = new ArrayList<String>();
neg.add("na");
neg.add("tc");
pos = new ArrayList<String>();
pos.add("oca");
planner.addOperator("sCA", precond, neg, pos);
precond = new ArrayList<String>();
precond.add("nb");
precond.add("td");
precond.add("nd");
neg = new ArrayList<String>();
neg.add("nb");
neg.add("td");
pos = new ArrayList<String>();
pos.add("odb");
planner.addOperator("sDB", precond, neg, pos);
List resultats = planner.findPlan();
System.out.println(resultats);
}
}
```

Output:

```
D:\ AI Lab>javac ClassicalPlanning.java

D:\ AI Lab> java ClassicalPlanning

The solution: [uBA, uAG, uGC, uCH, sCA, sFC, sBF]
```

Result:

Thus, block world problem has been implemented successfully and the output is verified.

Topic Beyond Syllabus

Implement Naïve Bayes Model

AIM

To implement Naïve Bayes Models.

ALGORITHM

1. The code starts by loading the iris dataset.
2. The data is then split into a training and test set of equal size.
3. Next, a Gaussian Naive Bayes classifier is trained using the training set.
4. Then predictions are made on the test set with accuracy scores calculated for each prediction.
5. Finally, a confusion matrix is created to show how well each prediction was classified as correct or incorrect.
6. The code is used to train a Gaussian Naive Bayes classifier and then use it to make predictions.
7. The code prints the model's predictions, as well as the test set's output for comparison.

PROGRAM

```
from sklearn import datasets from sklearn.metrics
import confusion_matrix
from sklearn.model_selection
import train_test_split from pandas
import DataFrame from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score
# The dataset
iris = datasets.load_iris()
X = iris.data
Y = iris.target X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=1/3)
# Training a Gaussian Naive Bayes classifier
model = GaussianNB() model.fit(X_train, Y_train)
# Predictions
model_predictions = model.predict(X_test)
print("\n",model_predictions)
print ("\n", Y_test)
# Accuracy of prediction
accuracyScore = accuracy_score(Y_test, model_predictions)
print ("\naccuracyScore is",accuracyScore )
# Creating a confusion matrix
cm=confusion_matrix(Y_test,model_predictions)
print ("\nconfusion matrix",cm)
```

OUTPUT

```
[0 2 1 1 0 0 2 2 1 2 1 0 1 2 0 2 2 0 2 1 1 2 2 0 1 0 0 2 2 0 0 2 0 1 1 1 1 1 2 1 1 0 0 2 1 0 1 0 2 2]
[0 2 1 1 0 0 2 2 1 2 1 0 1 2 0 2 2 0 2 1 1 2 2 0 1 0 0 2 2 0 0 2 0 1 1 1 1 1 2 1 1 0 0 2 1 0 1 0 2 2] accuracy
Score is 1.0
confusion matrix [[16 0 0] [ 0 17 0] [ 0 0 17]]
```

RESULT

Thus, the program to implement Naïve Bayes Model is implemented and executed successfully.