AD8413 ARTIFICIAL INTELLIGENCE – I LABORATORY

LIST OF EXPERIMENTS:

1. Develop PEAS descriptions for given AI tasks

2. Implement basic search strategies for selected AI applications

3. Implement A* and memory bounded A* algorithms

4. Implement genetic algorithms for AI tasks

5. Implement simulated annealing algorithms for AI tasks

6. Implement alpha-beta tree search

7. Implement backtracking algorithms for CSP

8. Implement local search algorithms for CSP

9. Implement propositional logic inferences for AI tasks

10. Implement resolution based first order logic inferences for AI tasks

11. Implement classical planning algorithms

12. Mini-Project

EX.NO.1 DEVELOP PEAS DESCRIPTION FOR GIVEN TASK

INTRODUCTION:

The PEAS system delivers the performance measure with respect to the environment, actuators, and sensors of the respective agent. Most of the highest performing agents are Rational Agents. **PEAS** stands for a Performance measure, Environment, Actuator, Sensor.

1. **Performance Measure:** Performance measure is the unit to define the success of an agent. Performance varies with agents based on their different precepts.
2. **Environment**: Environment is the surrounding of an agent at every instant. It keeps changing with time if the agent is set in motion.
3. **Actuator**: An actuator is a part of the agent that delivers the output of action to the environment.
4. **Sensor**: Sensors are the receptive parts of an agent that takes in the input for the agent.

Some PEAS Descriptors Examples/Problems:

1.PEAS descriptor for playing soccer:
Performance Measure: scoring goals, defending, speed

Environment: playground, teammates, opponents, ball

Actuators: body, dribbling, tackling, passing the ball, shooting

Sensors: camera, ball sensor, location sensor, other players' locator

2. PEAS descriptor for exploring the subsurface oceans of Titan.
Performance Measure: safety, images quality, video quality

Environment: ocean, water

Actuators: mobile diver, steering, brake, accelerator

Sensors: video, accelerometers, depth sensor, GPS

3. PEAS descriptor for practicing tennis against a wall.
Performance Measure: hit speed, hit accuracy

Environment: playground, racquet, ball, wall

Actuators: ball, racquet, joint arm

Sensors: ball locator, camera, racquet sensor

4.PEAS descriptor for performing a high jump.
Performance Measure: safety, altitude

Environment: wall

Actuators: jumping apparatus

Sensors: camera, height sensor

4.PEAS descriptor for knitting a sweater.
Performance Measure: size, looking, comfort

Environment: craft, pattern

Actuators: needles, yarn, jointed-arms

Sensors: pattern sensor

5. PEAS descriptor for bidding on an item at an auction.
Performance Measure: cost, value, necessity, quality

Environment: auctioneer, items, bidders

Actuators: speaker, display items

Sensors: camera, price monitor

6. PEAS descriptor for shopping for used AI books on the Internet.
Performance Measure: price, quality, authors, book review

Environment: web, vendors, shippers

Actuators: fill-in the form, follow URL, display to the user

Sensors: HTML

7.PEAS descriptor for playing a tennis match.
Performance Measure: winning

Environment: playground, racquet, ball, opponent

Actuators: ball, racquet, joint arm

Sensors: ball locator, camera, racquet sensor, opponent locator

9.PEAS descriptor for Automated Car Driver:
Performance Measure: Safety, Optimum speed, Comfortable journey

Environment: Roads, Traffic conditions

Actuators: Steering wheel, Accelerator, gear

Sensors: Cameras, Sonar system

10.PEAS descriptor for vacuum cleaner:
Performance Measure: Cleanness, Efficiency, Battery life, Security

Environment: Room, Table, Wood floor, Carpet

Actuators: Wheels, Brushes, Vacuum extractor

Sensors: Camera, Cliff sensor, Bump sensor, Infrared wall sensor

EX.NO.2 BASIC SEARCH STRATEGIES FOR AI

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key'), and explores all of the neighbour nodes at the present depth prior to moving on to the nodes at the next depth level.

PROGRAM FOR BFS

```
import java.io.*;
import java.util.*;
public class BFSTraversal
{
  private int node;       /* total number number of nodes in the graph */
   private LinkedList<Integer> adj[];      /* adjacency list */
   private Queue<Integer> que;          /* maintaining a queue */
   BFSTraversal(int v)
   {
      node = v;
     adj = new LinkedList[node];
      for (int i=0; i<v; i++)
     {
         adj[i] = new LinkedList<>();
      }
      que = new LinkedList<Integer>();
   }
  void insertEdge(int v,int w)
   {
      adj[v].add(w);      /* adding an edge to the adjacency list (edges are bidirectional in this
 example) */
   }
   void BFS(int n)
   {
      boolean nodes[] = new boolean[node];    /* initialize boolean array for holding the dat
a */
      nodes[n]=true;
      que.add(n);       /* root node is added to the top of the queue */
      while (que.size() != 0)
      {
        n = que.poll();        /* remove the top element of the queue */
         System.out.print(n+" ");    /* print the top element of the queue */
        for (int i = 0; i < adj[n].size(); i++)  /* iterate through the linked list and push all neig
hbors into queue */
         {
           a = adj[n].get(i);
            if (!nodes[a])  /* only insert nodes into queue if they have not been explored alrea
dy */
            {
               nodes[a] = true;
               que.add(a);
            }
         }
      }
   }
   public static void main(String args[])
   {
      BFSTraversal graph = new BFSTraversal(6);
      graph.insertEdge(0, 1);
      graph.insertEdge(0, 3);
      graph.insertEdge(0, 4);
      graph.insertEdge(4, 5);
      graph.insertEdge(3, 5);
      graph.insertEdge(1, 2);
      graph.insertEdge(1, 0);
      graph.insertEdge(2, 1);
      graph.insertEdge(4, 1);
      graph.insertEdge(3, 1);
      graph.insertEdge(5, 4);
      graph.insertEdge(5, 3);
```

```
            System.out.println("Breadth First Traversal for the graph is:");
            graph.BFS(0);
        }
    }
```

OUTPUT:

```
 Breadth First Traversal for the graph is:
 0 1 3 4 2 5
```

PROGRAM FOR DFS :

```
import java.util.*;
  class DFSTraversal {
  private LinkedList<Integer> adj[]; /*adjacency list representation*/
  private boolean visited[];
    /* Creation of the graph */
  DFSTraversal(int V) /*'V' is the number of vertices in the graph*/
  {
    adj = new LinkedList[V];
    visited = new boolean[V];

    for (int i = 0; i < V; i++)
      adj[i] = new LinkedList<Integer>();
  }

  /* Adding an edge to the graph */
  void insertEdge(int src, int dest) {
    adj[src].add(dest);
  }
    void DFS(int vertex) {
    visited[vertex] = true; /*Mark the current node as visited*/
    System.out.print(vertex + " ");
      Iterator<Integer> it = adj[vertex].listIterator();
    while (it.hasNext()) {
      int n = it.next();
      if (!visited[n])
        DFS(n);
    }
  }

  public static void main(String args[]) {
    DFSTraversal graph = new DFSTraversal(8);

        graph.insertEdge(0, 1);
        graph.insertEdge(0, 2);
        graph.insertEdge(0, 3);
        graph.insertEdge(1, 3);
        graph.insertEdge(2, 4);
        graph.insertEdge(3, 5);
        graph.insertEdge(3, 6);
        graph.insertEdge(4, 7);
        graph.insertEdge(4, 5);
        graph.insertEdge(5, 2);
         System.out.println("Depth First Traversal for the graph is:");
        graph.DFS(0);
  }
}
```

OUTPUT:

```
Depth First Traversal for the graph is:
0 1 3 5 2 4 7 6
```

{

EX.NO.3 IMPLEMENTATION OF A * AND MEMORY BOUNDED A * ALGORITHMS

A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.

Why A* Search Algorithm?

Informally speaking, A* Search algorithms, unlike other traversal techniques, it has “brains”. What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections.

And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

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 node 1.");
eval();
}
public int COST(int currentNode,intinputSet[],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,intinputSet[],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;
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 :

```
Enter no. of
nodes:=> 5
Enter weight of 1 to 2:=>4
Enter weight of 1 to 3:=>6
Enter weight of 1 to 4:=>3
Enter weight of 1 to 5:=>7
Enter weight of 2 to 1:=>3
Enter weight of 2 to 3:=>1
Enter weight of 2 to 4:=>7
Enter weight of 2 to 5:=>4
Enter weight of 3 to 1:=>7
Enter weight of 3 to 2:=>4

Enter weight of 3 to 4:=>3

Enter weight of 3 to 5:=>6

Enter weight of 4 to 1:=>8

Enter weight of 4 to 2:=>5

Enter weight of 4 to 3:=>3

Enter weight of 4 to 5:=>2

Enter weight of 5 to 1:=>4

Enter weight of 5 to 2:=>3

Enter weight of 5 to 3:=>2

Enter weight of 5 to 4:=>1

Starting node assumed to be node 1.

The tour is 1-2-3-4-5-1

The final cost is 14
```

EX NO:4 Implement genetic algorithms for AI tasks

Genetic Algorithms(GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. These are intelligent exploitation of random search provided with historical data to direct the search into the region of better performance in solution space. They are commonly used to generate high-quality solutions for optimization problems and search problems.

Genetic algorithms simulate the process of natural selection which means those species who can adapt to changes in their environment are able to survive and reproduce and go to next generation. In simple words, they simulate "survival of the fittest" among individual of consecutive generation for solving a problem. Each generation consist of a population of individuals and each individual represents a point in search space and possible solution. Each individual is represented as a string of character/integer/float/bits. This string is analogous to the Chromosome.

Foundation of Genetic Algorithms

Genetic algorithms are based on an analogy with genetic structure and behaviour of chromosomes of the population. Following is the foundation of GAs based on this analogy –

Individual in population compete for resources and mate

Those individuals who are successful (fittest) then mate to create more offspring than others

Genes from "fittest" parent propagate throughout the generation, that is sometimes parents create offspring which is better than either parent.

Thus each successive generation is more suited for their environment.

Search space

The population of individuals are maintained within search space. Each individual represents a solution in search space for given problem. Each individual is coded as a finite length vector (analogous to chromosome) of components. These variable components are analogous to Genes. Thus a chromosome (individual) is composed of several genes (variable components).

Fitness Score

A Fitness Score is given to each individual which shows the ability of an individual to "compete". The individual having optimal fitness score (or near optimal) are sought.

The GAs maintains the population of n individuals (chromosome/solutions) along with their fitness scores.The individuals having better fitness scores are given more chance to reproduce than others. The individuals with better fitness scores are selected who mate and produce better offspring by combining chromosomes of parents. The population size is static so the room has to be created for new arrivals. So, some individuals die and get replaced by new arrivals eventually creating new generation when all the mating opportunity of the old population is exhausted. It is hoped that over successive generations better solutions will arrive while least fit die.

Each new generation has on average more "better genes" than the individual (solution) of previous generations. Thus each new generations have better "partial solutions" than previous generations. Once the offspring produced having no significant difference from offspring produced by previous populations, the population is converged. The algorithm is said to be converged to a set of solutions for the problem.

Operators of Genetic Algorithms

Once the initial generation is created, the algorithm evolves the generation using following operators –

1) Selection Operator: The idea is to give preference to the individuals with good fitness scores and allow them to pass their genes to successive generations.

2) Crossover Operator: This represents mating between individuals. Two individuals are selected using selection operator and crossover sites are chosen randomly. Then the genes at these crossover sites are exchanged thus creating a completely new individual (offspring)

PROGRAM:

```
import java.util.Random;
//Main class
public class SimpleDemoGA {
    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:

```
Generation: 0 Fittest: 5

Solution found in generation 0
Fitness: 5
Genes: 11111
```

EX NO.5 Java program to implement Simulated Annealing

Simulated annealing (SA) is a probabilistic technique for approximating the global optimum of a given function. Specifically, it is a metaheuristic to approximate global optimization in a large search space for an optimization problem. It is often used when the search space is discrete (for example the traveling salesman problem, the boolean satisfiability problem, protein structure prediction, and job-shop scheduling). For problems where finding an approximate global optimum is more important than finding a precise local optimum in a fixed amount of time, simulated annealing may be preferable to exact algorithms such as gradient descent or branch and bound.

The name of the algorithm comes from annealing in metallurgy, a technique involving heating and controlled cooling of a material to alter its physical properties. Both are attributes of the material that depend on their thermodynamic free energy. Heating and cooling the material affects both the temperature and the thermodynamic free energy or Gibbs energy. Simulated annealing can be used for very hard computational optimization problems where exact algorithms fail; even though it usually achieves an approximate solution to the global minimum, it could be enough for many practical problems.

The problems solved by SA are currently formulated by an objective function of many variables, subject to several constraints. In practice, the constraint can be penalized as part of the objective function

PROGRAM:

```
import java.util.*;
public class SimulatedAnnealing {
        // Initial and final temperature
        public static double T = 1;
        // Simulated Annealing parameters
        // Temperature at which iteration terminates
        static final double Tmin = .0001;
        // Decrease in temperature
        static final double alpha = 0.9;
```

```
// Number of iterations of annealing
// before decreasing temperature
static final int numIterations = 100;

// Locational parameters
// Target array is discretized as M*N grid
static final int M = 5, N = 5;
// Number of objects desired
static final int k = 5;
public static void main(String[] args) {
        // Problem: place k objects in an MxN target
        // plane yielding minimal cost according to
        // defined objective function
        // Set of all possible candidate locations
        String[][] sourceArray = new String[M][N];



        // Global minimum
        Solution min = new Solution(Double.MAX_VALUE, null);
        // Generates random initial candidate solution
        // before annealing process
        Solution currentSol = genRandSol();
        // Continues annealing until reaching minimum
        // temperature
        while (T > Tmin) {
                for (int i=0;i<numIterations;i++){
                        // Reassigns global minimum accordingly
                        if (currentSol.CVRMSE < min.CVRMSE){
                                min = currentSol;
                        }
                        Solution newSol = neighbor(currentSol);
                        double ap = Math.pow(Math.E,
                                (currentSol.CVRMSE - newSol.CVRMSE)/T);
                        if (ap > Math.random())
                                currentSol = newSol;
                }
                T *= alpha; // Decreases T, cooling phase
        }
        //Returns minimum value based on optimization
        System.out.println(min.CVRMSE+"\n\n");
        for(String[] row:sourceArray) Arrays.fill(row, "X");
        // Displays
        for (int object:min.config) {
                int[] coord = indexToPoints(object);
                sourceArray[coord[0]][coord[1]] = "-";
        }

        // Displays optimal location
        for (String[] row:sourceArray)
                System.out.println(Arrays.toString(row));
}

// Given current configuration, returns "neighboring"
// configuration (i.e. very similar)
// integer of k points each in range [0, n)
/* Different neighbor selection strategies:
        * Move all points 0 or 1 units in a random direction
        * Shift input elements randomly
        * Swap random elements in input sequence
        * Permute input sequence
        * Partition input sequence into a random number
        of segments and permute segments */
public static Solution neighbor(Solution currentSol){
        // Slight perturbation to the current solution
        // to avoid getting stuck in local minimas
        // Returning for the sake of compilation
        return currentSol;
}

// Generates random solution via modified Fisher-Yates
// shuffle for first k elements
// Pseudorandomly selects k integers from the interval
// [0, n-1]
public static Solution genRandSol(){
        // Instantiating for the sake of compilation
        int[] a = {1, 2, 3, 4, 5};

        // Returning for the sake of compilation
        return new Solution(-1, a);
}

// Complexity is O(M*N*k), asymptotically tight
public static double cost(int[] inputConfiguration){
        // Given specific configuration, return object
        // solution with assigned cost
        return -1; //Returning for the sake of compilation
}
// Mapping from [0, M*N] --> [0,M]x[0,N]
public static int[] indexToPoints(int index){
        int[] points = {index%M, index/M};
```

```
		return points;
	}
	// Class solution, bundling configuration with error
	static class Solution {
		// function value of instance of solution;
		// using coefficient of variance root mean
		// squared error
		public double CVRMSE;
		public int[] config; // Configuration array
		public Solution(double CVRMSE, int[] configuration) {
			this.CVRMSE = CVRMSE;
			config = configuration;
		}
	}
}
```

EX NO:6 **Implement alpha-beta tree search**

Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.

As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called pruning. This involves two threshold parameter Alpha and beta for future expansion, so it is called alpha-beta pruning. It is also called as Alpha-Beta Algorithm.

Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.

The two-parameter can be defined as:

Alpha: The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is -∞.

Beta: The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.

The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

Note: To better understand this topic, kindly study the minimax algorithm.

Condition for Alpha-beta pruning:

The main condition which required for alpha-beta pruning is:

$\alpha >= \beta$

Key points about alpha-beta pruning:

The Max player will only update the value of alpha.

The Min player will only update the value of beta.

While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.

We will only pass the alpha, beta values to the child nodes.

<u>PROGRAM:</u>

```
import java.io.*;
 class GFG {
 // 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:

```
Output:

The optimal value is : 5
```

EX NO:7 Implement backtracking algorithms for CSP

constraint satisfaction problems, or **CSPs** for short, are a flexible approach to searching that have proven useful in many AI-style problems

CSPs can be used to solve problems such as

- **graph-coloring**: given a graph, the a coloring of the graph means assigning each of its vertices a color such that no pair of vertices connected by an edge have the same color
  - in general, this is a very hard problem, e.g. determining if a graph can be colored with 3 colors is NP-hard
  - many problems boil down to graph coloring, or related problems
- **job shop scheduling**: e.g. suppose you need to complete a set of tasks, each of which have a duration, and constraints upon when they start and stop (e.g. task c can't start until both task a and task b are finished)
  - CSPs are a natural way to express such problems
- **cryptarithmetic puzzles**: e.g. suppose you are told that TWO + TWO = FOUR, and each of the letters corresponds to a *different* digit from 0 to 9, and that a number can't start with 0 (so T and F are not 0); what, if any, are the possible values for the letters?
  - while these are not directly useful problems, they are a simple test case for CSP solvers

CSP consists of three main components:

- XX: a set of **variables** {X1,…,Xn}{X1,…,Xn}
- DD: a set of **domains** {D1,…,Dn}{D1,…,Dn}, one domain per variable
  - we're only going to consider **finite domains**; you can certainly have CSPs with infinite domains (e.g. real numbers), but we won't consider such problems here
- CC is a set of **constraints** that specify allowable assignments of values to variables
  - for example, a **binary constraint** consists of a pair of different variables, (Xi,Xj)(Xi,Xj), and a set of pairs of values that XiXi and XjXj can take on at the same time

  - we will usually only deal with binary constraints; constraints between three or more variables are possible (e.g. Xi,Xj,XkXi,Xj,Xk are all different), but they don’t occur too frequently, and can be decomposed into binary constraints

PROGRAM:

```
public class Sudoku {
     // N is the size of the 2D matrix   N*N
    static int N = 9;
     /* Takes a partially filled-in grid and attempts
    to assign values to all unassigned locations in
    such a way to meet the requirements for
    Sudoku solution (non-duplication across rows,
    columns, and boxes) */
    static boolean solveSudoku(int grid[][], int row,
                               int col)
    {

        /*if we have reached the 8th
          row and 9th column (0
          indexed matrix) ,
          we are returning true to avoid further
          backtracking       */
        if (row == N - 1 && col == N)
            return true;

        // Check if column value  becomes 9 ,
        // we move to next row
        // and column start from 0
        if (col == N) {
            row++;
            col = 0;
        }
         // Check if the current position
        // of the grid already
        // contains value >0, we iterate
        // for next column
        if (grid[row][col] != 0)
            return solveSudoku(grid, row, col + 1);
         for (int num = 1; num < 10; num++) {
             // Check if it is safe to place
            // the num (1-9)  in the
            // given row ,col ->we move to next column
            if (isSafe(grid, row, col, num)) {
                 /*  assigning the num in the current
                (row,col)  position of the grid and
                assuming our assigned num in the position
                is correct */
                grid[row][col] = num;

                // Checking for next
                // possibility with next column
                if (solveSudoku(grid, row, col + 1))
                    return true;
            }
            /* removing the assigned num , since our
               assumption was wrong , and we go for next
               assumption with diff num value   */
            grid[row][col] = 0;
        }
        return false;
    }
```

```
/* A utility function to print grid */
static void print(int[][] grid)
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            System.out.print(grid[i][j] + " ");
        System.out.println();
    }
}

// Check whether it will be legal
// to assign num to the
// given row, col
static boolean isSafe(int[][] grid, int row, int col,
                      int num)
{

    // Check if we find the same num
    // in the similar row , we
    // return false
    for (int x = 0; x <= 8; x++)
        if (grid[row][x] == num)
            return false;

    // Check if we find the same num
    // in the similar column ,
    // we return false
    for (int x = 0; x <= 8; x++)
        if (grid[x][col] == num)
```

```
            return false;
     // Check if we find the same num
    // in the particular 3*3
    // matrix, we return false
    int startRow = row - row % 3, startCol
                                = col - col % 3;
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            if (grid[i + startRow][j + startCol] == num)
                return false;
     return true;
}
  // Driver Code
public static void main(String[] args)
{
    int grid[][] = { { 3, 0, 6, 5, 0, 8, 4, 0, 0 },
                     { 5, 2, 0, 0, 0, 0, 0, 0, 0 },
                     { 0, 8, 7, 0, 0, 0, 0, 3, 1 },
                     { 0, 0, 3, 0, 1, 0, 0, 8, 0 },
                     { 9, 0, 0, 8, 6, 3, 0, 0, 5 },
                     { 0, 5, 0, 0, 9, 0, 6, 0, 0 },
                     { 1, 3, 0, 0, 0, 0, 2, 5, 0 },
                     { 0, 0, 0, 0, 0, 0, 0, 7, 4 },
                     { 0, 0, 5, 2, 0, 6, 3, 0, 0 } };
     if (solveSudoku(grid, 0, 0))
        print(grid);
    else
        System.out.println("No Solution exists");
}
// This is code is contributed by Pradeep Mondal P
```

```
}
```

Output:

```
3 1 6 5 7 8 4 9 2
5 2 9 1 3 4 7 6 8
4 8 7 6 2 9 5 3 1
2 6 3 4 1 5 9 8 7
9 7 4 8 6 3 1 2 5
8 5 1 7 9 2 6 4 3
1 3 8 9 4 7 2 5 6
6 9 2 3 5 1 8 7 4
7 4 5 2 8 6 3 1 9
```

EX NO:8 Implement local search algorithms for CSP

Local search algorithms for CSPs use a complete-state formulation: the initial state assigns a value to every variable, and the search change the value of one variable at a time

PROGRAM:

```
import java.sql.Array;

import java.sql.Time;

import java.util.ArrayList;

import java.util.Collection;

import java.util.Collections;

import java.util.HashMap;

import java.util.Map;

import java.util.Scanner;

public class CryptArithmetic

{

        static Scanner sc = new Scanner(System.in);

        static ArrayList<Character> uniquechar = new ArrayList<Character>();

        static int nos[] = {0,1,2,3,4,5,6,7,8,9};

        static HashMap<Character, Integer> hm = new HashMap<Character, Integer>();

        static int no1,no2,no3,count=0;

        static boolean solutionfound=false;

        static ArrayList<ArrayList<Integer>> permuts = new ArrayList<ArrayList<Integer>>();

        static String s1,s2,s3;


        public static void main(String[] args) throws InterruptedException

        {

                getInput();

                System.out.println("Calculating. Please wait...");

                long start = System.currentTimeMillis();
```

```
		calculate();
		long end = System.currentTimeMillis();
		double time = (end-start)/1000.0;
		System.out.println("Time required for execution: "+time+" seconds");
	}
	public static void getInput()
	{
		System.out.println("Enter string 1:");
		s1 = sc.nextLine();
		System.out.println("Enter string 2:");
		s2 = sc.nextLine();
		System.out.println("Enter string 3:");
		s3 = sc.nextLine();

		addToArrayList(s1);
		addToArrayList(s2);
		addToArrayList(s3);
	}
	public static void calculate()
	{
		Collections.sort(uniquechar);
		permute(nos, 0);
		for(int i=0;i<permuts.size();i++)
		{
			for(int j=0;j<uniquechar.size();j++)
			{
				hm.put(uniquechar.get(j),permuts.get(i).get(j));

			}
```

```
                    no1 = getNumber(s1);
                    no2 = getNumber(s2);
                    no3 = getNumber(s3);

                    if(no3==no1+no2 && getLengthOfInt(no1)==s1.length() &&
getLengthOfInt(no2)==s2.length() && getLengthOfInt(no3)==s3.length() && count<1)
                    {
                              solutionfound=true;
                              System.out.println(s1+":"+no1+"  "+s2+":"+no2+"
"+s3+":"+no3);
                              count++;
                    }
          }

          if(!solutionfound)
                    System.out.println("No solution found!");

     }

     public static void permute(int []a,int k )
     {
          if(k==a.length)
          {
                    ArrayList<Integer> perm = new ArrayList<Integer>();
                    for(int i=0;i<a.length;i++)
                    {
                              perm.add(a[i]);
                    }
                    permuts.add(perm);
          }
          else
```

```
        {
                for (int i = k; i < a.length; i++)
                {
                        int temp=a[k];
                        a[k]=a[i];
                        a[i]=temp;
                        permute(a,k+1);
                        temp=a[k];
                        a[k]=a[i];
                        a[i]=temp;
                }
        }
}

public static boolean found(char c)
{
        boolean flag=false;
        for(int i=0;i<uniquechar.size();i++)
        {
                if(uniquechar.get(i)==c)
                        flag=true;
        }
        if(flag)
                return true;
        else
                return false;
}

public static void addToArrayList(String s)
{
        for(int i=0;i<s.length();i++)
        {
                if(!found(s.charAt(i)))
                {
                        uniquechar.add(s.charAt(i));
                }
        }
}

public static void iterateHashMap()
{
        for (Map.Entry<Character, Integer> entry : hm.entrySet())
        {
           char key = entry.getKey();
           int value = entry.getValue();

           System.out.println("Key:"+key+" Value:"+value);
        }
}

public static int getNumber(String s)
{

        String temp="";
        for(int i=0;i<s.length();i++)
        {
                temp=temp+hm.get(s.charAt(i));
        }
        return Integer.parseInt(temp);
}
```

```
	public static int getLengthOfInt(int n)
	{
		return String.valueOf(n).length();
	}
}
```

OUTPUT:

```
Enter string 1:
no
Enter string 2:
no
Enter string 3:
yes
Calculating. Please wait...
no:52  no:52  yes:104
```

EX NO :9 . Implement propositional logic inferences for AI tasks

Propositional logic (PL) is the simplest form of logic where all the statements are made by propositions. A proposition is a declarative statement which is either true or false. It is a technique of knowledge representation in logical and mathematical form.

Example:

a) It is Sunday.
b) The Sun rises from West (False proposition)
c) 3+3= 7(False proposition)
d) 5 is a prime number.

Following are some basic facts about propositional logic:

- Propositional logic is also called Boolean logic as it works on 0 and 1.
- In propositional logic, we use symbolic variables to represent the logic, and we can use any symbol for a representing a proposition, such A, B, C, P, Q, R, etc.
- Propositions can be either true or false, but it cannot be both.
- Propositional logic consists of an object, relations or function, and **logical connectives**.
- These connectives are also called logical operators.
- The propositions and connectives are the basic elements of the propositional logic.
- Connectives can be said as a logical operator which connects two sentences.
- A proposition formula which is always true is called **tautology**, and it is also called a valid sentence.
- A proposition formula which is always false is called **Contradiction**.
- A proposition formula which has both true and false values is called
- Statements which are questions, commands, or opinions are not propositions such as "**Where is Rohini**", "**How are you**", "**What is your name**", are not propositions.

Syntax of propositional logic:

The syntax of propositional logic defines the allowable sentences for the knowledge representation. There are two types of Propositions:

Atomic Propositions

Compound propositions

Atomic Proposition: Atomic propositions are the simple propositions. It consists of a single proposition symbol. These are the sentences which must be either true or false

a) 2+2 is 4, it is an atomic proposition as it is a true fact.

b) "The Sun is cold" is also a proposition as it is a false fact.

Compound proposition: Compound propositions are constructed by combining simpler or atomic propositions, using parenthesis and logical connectives.

Example:

a) "It is raining today, and street is wet."

b) "Ankit is a doctor, and his clinic is in Mumbai."

Logical Connectives:

Logical connectives are used to connect two simpler propositions or representing a sentence logically. We can create compound propositions with the help of logical connectives. There are mainly five connectives, which are given as follows:

Negation: A sentence such as ¬ P is called negation of P. A literal can be either Positive literal or negative literal.

Conjunction: A sentence which has ∧ connective such as, P ∧ Q is called a conjunction.

Example: Rohan is intelligent and hardworking. It can be written as,

P= Rohan is intelligent,

Q= Rohan is hardworking. → P∧ Q.

Disjunction: A sentence which has ∨ connective, such as P ∨ Q. is called disjunction, where P and Q are the propositions.

Example: "Ritika is a doctor or Engineer",

Here P= Ritika is Doctor. Q= Ritika is Doctor, so we can write it as P ∨ Q.

I**mplication**: A sentence such as P → Q, is called an implication. Implications are also known as if-then rules. It can be represented as

If it is raining, then the street is wet.

Let P= It is raining, and Q= Street is wet, so it is represented as P → Q

Biconditional: A sentence such as P⇔ Q is a Biconditional sentence, example If I am breathing, then I am alive

P= I am breathing, Q= I am alive, it can be represented as P ⇔ Q.

Following is the summarized table for Propositional Logic Connectives:

Propositional logic in Artificial intelligence

Truth Table:

In propositional logic, we need to know the truth values of propositions in all possible scenarios. We can combine all the possible combination with logical connectives, and the representation of these combinations in a tabular format is called Truth table. Following are the truth table for all logical connectives:

For Negation:

P	¬ P
True	**False**
False	**True**

For Conjunction:

P	Q	P∧ Q
True	True	**True**
True	False	**False**
False	True	**False**
False	False	**False**

For disjunction:

P	Q	P V Q.
True	True	**True**
False	True	**True**
True	False	**True**
False	False	**False**

For Implication:

P	Q	P→ Q
True	True	**True**
True	False	**False**
False	True	**True**
False	False	**True**

EX NO :10. Implement resolution based first order logic inferences for AI tasks

Resolution in the Propositional Calculus

In its simplest form Resolution is the inference rule:

{A OR C, B OR (NOT C)}

A OR B

More in general the Resolution Inference Rule is:

• Given as premises the clauses C1 and C2, where C1 contains the literal L and C2 contains the literal (NOT L), infer the clause C, called the Resolvent of C1 and C2, where C is the union of (C1 - {L}) and (C2 -{(NOT L)})

In symbols:

{C1, C2}

(C1 - {L}) UNION (C2 - {(NOT L)})

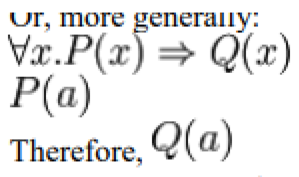
In first-order logic, resolution condenses the traditional syllogisms of logical inference down to a single rule.

To understand how resolution works, consider the following example syllogism of term logic:

All Greeks are Europeans.

Homer is a Greek.

Therefore, Homer is a European



Or, more generally:

$\forall x.P(x) \Rightarrow Q(x)$

$P(a)$

Therefore, $Q(a)$

To recast the reasoning using the resolution technique, first the clauses must be converted to conjunctive normal form. In this form, all quantification becomes implicit: universal quantifiers on variables (X, Y, …) are simply omitted as understood, while existentially quantified variables

$\neg P(x) \vee Q(x)$
$P(a)$
Therefore,
$Q(a)$.

So, the question is, how does the resolution technique derive the last clause from the first two? The rule is simple:

Find two clauses containing the same predicate, where it is negated in one clause but not in the other.

Perform unification on the two predicates. (If the unification fails, you made a bad choice of predicates. Go back to the previous step and try again.)

If any unbound variables which were bound in the unified predicates also occur in other predicates in the two clauses, replace them with their bound values (terms) there as well.

Discard the unified predicates, and combine the remaining ones from the two clauses into a new clause, also joined by the "∨" operator.

To apply this rule to the above example, we find the predicate P occurs in negated form

¬P(X)

in the first clause, and in non-negated form

P(a)

in the second clause. X is an unbound variable, while a is bound value (term). Unifying the two produces the substitution

X ↦ a

Discarding the unified predicates, and applying this substitution to the remaining predicates (just Q(X), in this case), produces the conclusion:

Q(a)

EX NO 11. IMPLEMENT CLASSICAL PLANNING ALGORITHMS

Language of Planning Problem:

What is STRIPS?

The Stanford Research Institute Problem Solver (STRIPS) is an automated planning technique

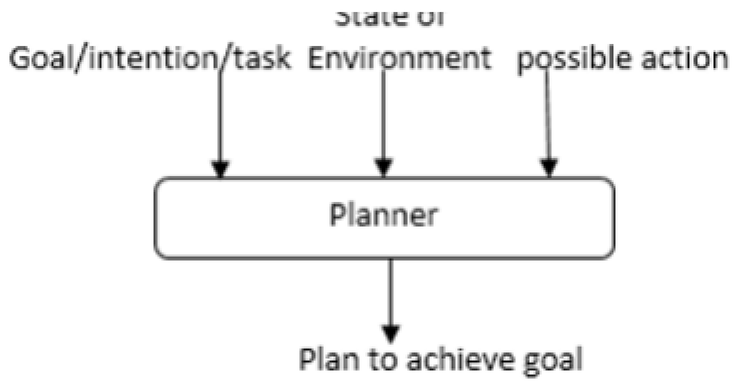
that works by executing a domain and problem to find a goal. With STRIPS, you first describe the world. You do this by providing objects, actions, preconditions, and effects. These are all the types of things you can do in the game world.

Once the world is described, you then provide a problem set. A problem consists of an initial state and a goal condition. STRIPS can then search all possible states, starting from the initial one, executing various actions, until it reaches the goal.

A common language for writing STRIPS domain and problem sets is the Planning Domain Definition Language (PDDL). PDDL lets you write most of the code with English words, so that it can be clearly read and (hopefully) well understood. It's a relatively easy approach to writing simple AI planning problems.

Problem statement

Design a planning agent for a Blocks World problem. Assume suitable initial state and final state for the problem.



Designing the Agent

Idea is to give an agent:

• Representation of goal/intention to achieve

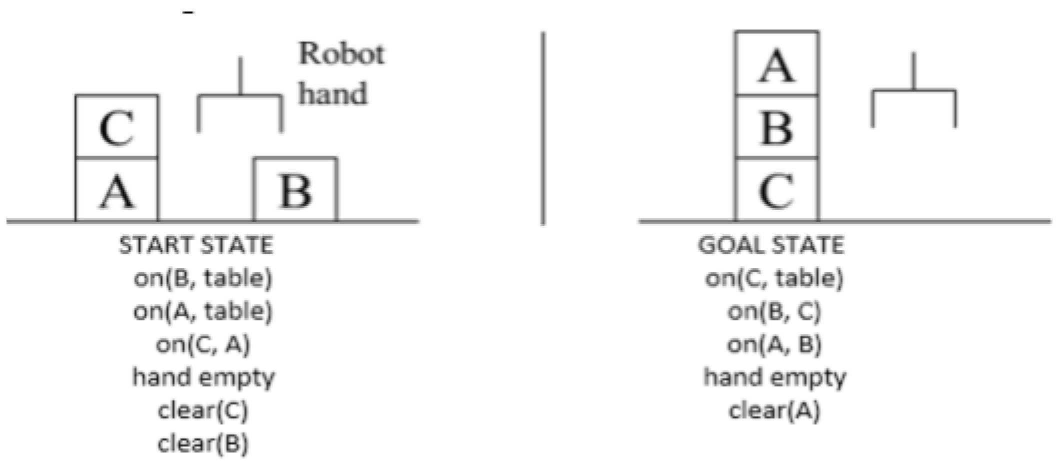
• Representation of actions it can perform; and

• Representation of the environment;

Then have the agent generate a plan to achieve the goal.

The plan is generated entirely by the planning system, without human intervention.

Assume start & goal states as below



a. STRIPS : A planning system – Has rules with precondition deletion list and addition list

Sequence of actions :

b. Grab C

c. Pickup C

d. Place on table C

e. Grab B

f. Pickup B

g. Stack B on C

h. Grab A

i. Pickup A

j. Stack A on B

Rules:

k. R1 : pickup(x)

1. Precondition & Deletion List : hand empty, on(x,table), clear(x)

2. Add List : holding(x)

l. R2 : putdown(x)

1. Precondition & Deletion List : holding(x)

2. Add List : hand empty, on(x,table), clear(x)

m. R3 : stack(x,y)

1. Precondition & Deletion List :holding(x), clear(y)

2. Add List : on(x,y), clear(x)

n. R4 : unstack(x,y)

1. Precondition & Deletion List : on(x,y), clear(x)

2. Add List : holding(x), clear(y)

Plan for the assumed blocks world problem

For the given problem, Start $\rightarrow$ Goal can be achieved by the following sequence:

1. Unstack(C,A)

2. Putdown(C)

3. Pickup(B)

4. Stack(B,C)

5. Pickup(A)

6. Stack(A,B)