Appendix A Lab Experiments

LABORATORY OUTCOMES

After reading this appendix, you will be able to perform the following list of experiments.

List of Experiments

1. Design of Intelligent System Using PEAS

- 1.1 Automated Taxi Driver
- 1.2 Vacuum Cleaner Agent
- 1.3 A Music Composer
- 1.4 An Aircraft Autolander
- 1.5 An Essay Evaluator
- 1.6 A Robotic Sentry Gun for the Keck Lab
- 1.7 Medical Diagnosis System

2. Problem Definition with State Space Representation

- **2.1** Implement Water Jug Problem Using Problem Formulation
- **2.2** Implement Wumpus World Problem Using Problem Formulation

3. Uniformed Search Techniques

- **3.1** Implement Path Finding in Maze Using Depth-First Search (DFS)
- **3.2** Implement Water Jug Problem Using Breadth-First Search (BFS)

4. Informed Search Technique

- **4.1** Implement 8-Puzzle Problem Using Hill Climbing
- **4.2** Implement 8-Puzzle Problem Using Best-First Search
- **4.3** Implement Tic-Tac-Toe Using A* Algorithm
- **4.4** Implement 8-Puzzle Problem Using A* Algorithm
- **4.5** Implement Travelling Salesman Problem (TSP) Using A* Algorithm
- **4.6** Implement 8-Queen Problem with Heuristic Function (Informed Search)

5. Adversarial Search

    5.1 Implement Minimax Algorithm

6. Constraint Satisfaction Problem

    6.1 Implement 8-Queen Problem

    6.2 Implement Map Colouring Problem

    6.3 Implement Crypt Arithmetic Problem

7. Design of a Planning System Using STRIPS (Block World Problem)

8. Implementation of Bayes' Belief Network (Probabilistic Reasoning in an Uncertain Domain)

9. Implement Resolution Inference Rule Using Prolog

10. Ontology Creating, Editing and Authoring Using Protégé Tool.

11. Inductive Learning Using Weka Tool

    11.1 Implement Decision Tree Learning

12. Study of Seiko DTRANS RT 3200 Robot

13. Mini Expert System Using PROLOG.

14. Programming Using Python

    14.1 Water Jug Problem Using Python

    14.2 Wumpus World Problem Using Python

    14.3 Eight Puzzle Problem Using Python

    14.4 Tic-Tak-Toe Game Using Python

    14.5 Eight/N- Queen Problem Using Python

    14.6 Minimax &Alpha Beta Pruning AI Algorithm in Tic-Tac-Toe Using Python

    14.7 Constraint Satisfaction Problem Using Python

15. Construction of a Domain-Specific Chatbot Using Natural Language Processing Techniques

Experiment No. 1

Design of Intelligent System Using PEAS

1.1 Automated Taxi Driver
1.2 Vacuum Cleaner Agent
1.3 A Music Composer
1.4 An Aircraft Autolander
1.5 An Essay Evaluator
1.6 A Robotic Sentry Gun for the Keck Lab
1.7 Medical Diagnosis System

Aim: To understand the concept of PEAS.

PEAS: PEASstands for *performance environment actuators sensors*.

a. Automated Taxi Driver

In designing an agent, the first step must always be to specify the task environment as fully observable.

To understand PEAS in a better way, let us try to analyse the complex problem of automatic taxi driver which is currently beyond the capabilities of existing technology. We would consider characteristics of PEAS for description of taxi's task environment.

Performance measure is the first to which we would like an automatic driver to Aspire. Desirable measures include getting correct destination, minimising fuel consumption, no wear and tear, minimising trip time and cost, minimising violation of traffic laws and disturbance to other

drivers, minimising safety and passenger comfort and maximizing profit. But in this scenario, some of the goals may conflict, so there will be some trade off involved.

Environment: The basic question that comes in the mind is what is the driving environment that a taxi will face? A taxi driver will face with a variety of roads, ruler lines and urban Valley to 12 Lane Freeway. The roads contain other traffic, pedestrians, stray animal, roads work, police potholes and cars. A taxi must also interact with potential and actual passengers. There might be some restriction on driving, such as left-hand side driving as in India, Japan, etc., or right-hand side driving. Otherwise the roads may be soaring temperature, desert areas and all snowfall regions like Kashmir. Thus, more restricted the environment, easier the design problem.

Actuators: The actuators available to an automated taxi will be more or less same as those available to human driver(i.e., control over engine through the accelerator and control over steering and breaking). In addition, it will output to a display screen or voice synthesizer talk back to passengers and perhaps some way to communicate with other drivers or vehicle politely or otherwise.

Sensors: The sensors will play a crucial role in determining where the taxi actually is, what else is on the road and how fast it is going. The basic sensors should therefore include one or more TV cameras, the tachometer and the odometer. To control the vehicle properly, especially on curves, it will also need to know the mechanical state of vehicle so it will need the usual array of engine and electrical system sensors. It might have instruments that are not available to average human driver, a satellite global positioning system (GPS) to give accurate position information with respect to an electronic map and infrared solar sensors to detect distance to other cars and obstacles. Finally, it will require keyboard or microphone for passenger to request a destination.

- Performance measure safe fast legal comfortable trip maximize profit
- Environment roads are the traffic pedestrian customers
- Activators steering accelerator break horn display
- Sensors, camera, sonar speedometer, odometer, GPS, etc.

b. Vacuum Cleaner Agent

Figure 1 iRobot Roomba series.

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.

c. Music Composer

- Performance Measures - number of measures composed per unit time, number of instruments considered, ease of play by a human, range of frequencies within human audible zone, melodic, harmonic and rhythmic criteria, ...
- Environment Software
- Actuator None required, this can be a pure softbot
- Sensors Code that reads in basic parameters

d. Aircraft Autolander

- Performance Measure: Lack of damage to plane, other aircraft or ground structures, lack of injuries to passengers or ground crew or other innocent observers, cargo remains intact, fuel economy, lands at correct airport on correct runway, doesn't take too long
- Environment: Lower atmosphere and surface of planet Earth.
- Actuators: Throttle, landing gear, rudders, ailerons, flaps ...
- Sensors: Cameras, Altimeter, Speedometer, other meters, ...

e. Essay Evaluator

- Performance Measures : awards scores for quality, penalizes crap, detection of plagiarism, impartiality, usefulness of explanation of grading, ...
- Environment: Software
- Actuator: None, this can be a pure softbot
- Sensors: File reading software, (perhaps even OCR)

f. Robotic Sentry Gun for the Keck Lab

- Performance Measures Percentage of correct targets hit, lack of hitting friends, minimal energy consumption, ...
- Environment: The Keck Lab
- Actuator: Gun, trigger, motors, camera, ...
- Sensors: Camera, sonar, bump sensors, ...

g. Medical diagnosis system

- Performance measure: Healthy patient, minimize costs, lawsuits
- Environment: Patient, hospital, staff
- Actuators: Screen display (questions, tests, diagnoses, treatments, referrals)
- Sensors: Keyboard (entry of symptoms, findings, patient's answers)

Experiment No. 2

Problem Definition with State Space Representation

2.1 Implement Water Jug Problem Using Problem Formulation

Aim: Implement water jug problem using BFS or DFS (Un-Informed Search).

Theory:

Problem Statement
For further explanation read Section 3.5 of Chapter 3.

In the **water jug problem in Artificial Intelligence**, we are provided with two jugs: one having the capacity to hold 3 gallons of water and the other has the capacity to hold 4 gallons of water. There is no other measuring equipment available and the jugs also do not have any kind of marking on them. So, the agent's task here is to fill the 4-gallon jug with 2 gallons of water by using only these two jugs and no other material. Initially, both our jugs are empty.

So, to solve this problem, following set of rules were proposed:

Production rules for solving the water jug problem

Here, let ***x*** denote the 4-gallon jug and ***y*** denote the 3-gallon jug.

S.No.	Initial State	Condition	Final state	Description of action taken
1.	(x,y)	If x<4	(4,y)	Fill the 4 gallon jug completely
2.	(x,y)	if y<3	(x,3)	Fill the 3 gallon jug completely
3.	(x,y)	If x>0	(x-d,y)	Pour some part from the 4 gallon jug
4.	(x,y)	If y>0	(x,y-d)	Pour some part from the 3 gallon jug
5.	(x,y)	If x>0	(0,y)	Empty the 4 gallon jug
6.	(x,y)	If y>0	(x,0)	Empty the 3 gallon jug
7.	(x,y)	If (x+y)<7	(4, y-[4-x])	Pour some water from the 3 gallon jug to fill the four gallon jug
8.	(x,y)	If (x+y)<7	(x-[3-y],y)	Pour some water from the 4 gallon jug to fill the 3 gallon jug.

9.	(x,y)	If (x+y)<4	(x+y,0)	Pour all water from 3 gallon jug to the 4 gallon jug
10.	(x,y)	if (x+y)<3	(0, x+y)	Pour all water from the 4 gallon jug to the 3 gallon jug

The listed production rules contain all the actions that could be performed by the agent in transferring the contents of jugs. But, to solve the water jug problem in a minimum number of moves, following set of rules in the given sequence should be performed:

Solution of water jug problem according to the production rules:

S.No.	4 gallon jug contents	3 gallon jug contents	Rule followed
1.	0 gallon	0 gallon	Initial state
2.	0 gallon	3 gallons	Rule no.2
3.	3 gallons	0 gallon	Rule no. 9
4.	3 gallons	3 gallons	Rule no. 2
5.	4 gallons	2 gallons	Rule no. 7
6.	0 gallon	2 gallons	Rule no. 5
7.	2 gallons	0 gallon	Rule no. 9

On reaching the 7th attempt, we reach a state which is our goal state. Therefore, at this state, our problem is solved.

Conclusion: Thus, we have successfully implemented water jug problem.

Program:
Problem Statement: There are two jugs (suppose capacity of 3 and 5) and we need to fill the jug in such a way that 5 liters capacity jug should contain 4 liters of water.

```
import java.util.*;
class WaterJug{
   public static void main(String sap[]){
   Scanner sc = new Scanner(System.in);

   // j1 is capacity of small tank
   System.out.print("\nEnter odd capacity of small tank: ");
   int j1 = sc.nextInt();

   // j2 is capacity of large tank
   System.out.print("\nEnter odd capacity of large tank: ");
   int j2 = sc.nextInt();
```

```
    // count takes care of number of iterations
    int count = j1 + j2;

    /* jug1 array would hold the values for smaller tank and jug2 array would hold the values for
larger     tank */
    int jug1[] = new int[count];
    int jug2[] = new int[count];

    int i=0;

    // initialzing jug1 and jug2 array
    jug1[i] = j1;
    jug2[i] = 0;
    i++;

    jug1[i] = 0;
    jug2[i] = j1;
    i++;

    while(i < count){
        if(jug1[i-1] > 0){
            // if jug1 has any amount of water i.e. it is not empty
            jug1[i] = jug1[i-1];
            jug2[i] = 0;
        }
        else{
            // jug1 is fully empty
            jug1[i] = j1;
            jug2[i] = jug2[i-1];
        }
        i++;

        if(jug2[i-1] > 0){
            // if jug2 has any amount of water i.e. it is not empty
            if(jug1[i-1] + jug2[i-1] < j2){
                // final result obtained
                jug2[i] = jug1[i-1] + jug2[i-1];
                jug1[i] = 0;
            }
            else{
                int temp = jug2[i-1];
                temp = j2 - temp;
                jug2[i] = temp + jug2[i-1];

                jug1[i] = jug1[i-1] - temp;
            }
        }
        else{
            // jug2 is fully empty
```

```
            jug2[i] = jug1[i-1];
            jug1[i] = 0;
        }
        i++;
    }

    // display final result
    for(i=0; i<count; i++){
            System.out.print("\nJUG1: "+jug1[i]+"\tJUG2: "+jug2[i]);
    }

    System.out.println();
    }
}
```

Output:

Enter odd capacity of small tank: 3

Enter odd capacity of large tank: 5

JUG1: 3 JUG2: 0
JUG1: 0 JUG2: 3
JUG1: 3 JUG2: 3
JUG1: 1 JUG2: 5
JUG1: 1 JUG2: 0
JUG1: 0 JUG2: 1
JUG1: 3 JUG2: 1
JUG1: 0 JUG2: 4

2.1 Implement Wumpus World Problem Using Problem Formulation

Aim: Implement Wumpus world problem (knowledge and reasoning).

For further explanation read Section 3.7 of Chapter 3.

Theory: A variety of 'worlds' are getting used as examples for knowledge representation, reasoning, and planning. Among them are the Vacuum World, the Block World, and also the Wumpus world. We'll examine the Wumpus world and during this context introduce the situation Calculus, the Frame drawback, and a range of axioms. The Wumpus world was introduced by Genesereth and is mentioned in Russell-Norvig. The Wumpus world could be an easy world (as is that the Block World) to represent knowledge and to reason.

It is a cave with a number of rooms, represented as a 4x4 square.

Stench		Breeze	Pit
Wumpus	Stench breeze gold	Pit	Breeze
Stench		Breeze	
Start	Breeze	Pit	Breeze

Rules of the Wumpus World: The neighbourhood of a node consists of the four squares north, south, east and west of the given square.

In a square, the agent gets a vector of percepts, with components Stench, Breeze, Glitter, Bump, Scream.

For example [Stench,None,Glitter,None,None]

1. Stench is perceived at a square if the Wumpus is at this square or in its neighbourhood.
2. Breeze is perceived at a square if a pit is in the neighbourhood of this square.
3. Glitter is perceived at a square if gold is in this square
4. Bump is perceived at a square if the agent goes Forward into a wall
5. Scream is perceived at a square if the Wumpus is killed anywhere in the cave

An agent can do the following actions (one at a time):

Turn(Right), Turn(Left), Forward, Shoot, Grab, Release, Climb

1. The agent can go Forward in the direction it is currently facing, or Turn Right, or Turn Left. Going Forward into a wall will generate a Bump percept.
2. The agent has a single arrow that it can Shoot. It will go straight in the direction faced by the agent until it hits (and kills) the Wumpus, or hits (and is absorbed by) a wall.
3. The agent can Grab a portable object at the current square or it can Release an object that it is holding.
4. The agent can Climb out of the cave if at the Start square.

The Start square is (1,1) and initially the agent is facing east. The agent dies if it is in the same square as the Wumpus.

The objective of the game is to kill the Wumpus, to pick up the gold, and to climb out with it.

The Situation Calculus: With the name situation, we have a tendency to mean a state of the world. The Situation Calculus is employed to reason concerning actions and their impact on the possible states of the world. We'll limit our discussion of the Situation Calculus to the case of the Wumpus world.

The Frame Problem: We are reasoning about possible states of the world, wherever the states are known by the actions by which we got to that state from the initial state. The Frame problem worries with the question of what happens to the truth-value of the statements that describe the world as we have a tendency to go from one world to the world resulting by application of an action. We have a tendency to contend with the frame problem by introducing a certain variety of axioms that go by names like effect Axioms, Frame Axioms, and Successor-State Axioms.

Representing our Knowledge about the Wumpus World

Percept (*x, y*)

Where *x* must be a percept vector and y must be a situation. It means that at situation y the agent perceives *x*.

For convenience we introduce the following definitions:

1. Percept ([Stench, *y, z, w, v*], *t*) ≥ Stench(t)
2. Percept ([*x,* Breeze, *z, w, v*], *t*) ≥ Breeze(t)
3. Percept ([*x, y,* Glitter, *w, v*], *t*) ≥AtGold(t)

Holding (*x, y*)

Where *x* is an object and *y* is a situation. It means that the agent is holding the object *x* in situation *y*.

Action (*x,y*)

Where *x* must be an action (i.e., Turn(Right), Turn(Left), Forward) and *y* must be a situation. It means that at situation *y* the agent takes action *x*.

At(*x,y,z*)

Where *x* is an object, *y* is a Location, i.e. a pair [*u,v*] with *u* and *v* in {1,2,3,4}, and *z* is a situation. It means that the agent *x* in situation *z* is at location *y*.

Present(*x,s*)

Means that object *x* is in the current room in the situation *s*.

Result(*x,y*)

It means that the result of applying action *x* to the situation *y* is the situation Result(*x,y*). Note that Result(*x,y*) is a term, not a statement.

For example, we can say

1. Result(Forward, S0) = S1
2. Result(Turn(Right),S1) = S2

These definitions could be created more general. Since within the Wumpus world there is one agent, there is no reason for us to form predicates and functions relative to a selected agent. In alternative 'worlds', we must always change things appropriately.

Another generalisation is feasible, to put in writing all assertions regarding the world as terms, and then add the predicates.

T(situation assertion-term)
To mean that in the specified situation the specified assertion is true.
Belief(agent situation assertion-term)
To mean that the specified agent in the specified situation believes the specified assertion to be true. Here are a series of statements about the Wumpus world.

Effect Axioms: Effect axioms characterize what is changed because of an action. For example:

1. Present(*x*,*s*) & Portable(*x*) ≥ Holding(*x*, Result(Grab,*s*))
2. Not Holding(*x*, Result(Release,*s*))

Frame Axioms: Frame axioms characterize what has remained the same because of an action. For example:

1. Holding(*x*,*s*) & (a/=Release) ≥ Holding(*x*,Result(a,s))
2. NOT Holding(*x*,*s*) & ((a/=Grab)) | NOT(Present(*x*,*s*) & Portable(x)) = > NOT Holding(x,Result(a,s))

Successor–State Axioms: For each predicate that can change over time they characterize the actions under which it changes and the actions under which it remains the same. For example:

Holding(x,Result(a,s)) IFF [(a=Grab & Present(x,s) & Portable(x)) OR (Holding(x,s) & (a/=Release))

More Definitions and Axioms

1. Orientation(Agent,s0) = 0
2. At(Agent,[1,1],s0)
3. Portable(Gold)
4. AtGold(s) ≥ Present(Gold, s)
5. LocationToward([*x*,*y*],0) = [*x*+1,*y*]
6. LocationToward([*x*,*y*],90) = [*x*,*y*+1]
7. LocationToward([*x*,*y*],180) = [*x*-1,*y*]
8. LocationToward([*x*,*y*],270) = [*x*,*y*-1]
9. At(p,l,s) ≥ LocationAhead(p,s) = LocationToward(l, Orientation(p,s))
10. Adjacent(l1,l2) IFF EXISTS d (l1 = LocationToward(l2,d))
11. Wall([*x*,*y*]) IFF (*x*=0 OR *x*=5 OR *y*=0 OR *y*=5)
12. At(p,l,Result(a,s)) IFF [(a=Forward & l=LocationAhead(p,s) & NOT Wall(l)) OR (At(p,l,s) & a/=Forward)]
13. Orientation(p,Result(a,s))=d IFF [(a=Turn(Right) & d=Mod(Orientation(p,s)-90,360)) OR (a=Turn(Left) & d=Mod(Orientation(p,s)+90,360)) OR (Orientation(p,s)=d & NOT(a=Turn(Right) & a=Turn(Left)))]
14. At(Wumpus.l1,s) & Adjacent(l1,l2) ≥ Smelly(l2)
15. At(Pit,l1,s) & Adjacent(l1,l2) ≥ Breezy(l2)

Model-Based and Diagnostic Reasoning: Causal Rules specify how aspects of the state of the world determine our percepts. Model-Based Reasoning is what we do when we use causal rules. Here are some causal rules:

1. At(Wumpus,l1,s) & Adjacent(l1,l2) ≥ Smelly(
2. l2)
3. At(Pit,l1,s) & Adjacent(l1,l2) ≥ Breezy(l2)

Diagnostic rules specify how to go from percepts to assertions about the state of the world. Diagnostic reasoning is what we do when we use diagnostic rules. Here are some diagnostic rules:

1. At(Agent,l,s) & Breeze(s) ≥ Breezy(l)
2. At(Agent,l,s) & Stench(s) ≥ Smelly(l)
3. Smelly(l1) ≥ (EXISTS l2 At(Wumpus,l2,s) & (l2=l1 OR Adjacent(l1,l2)))
4. At(Wumpus,x,t) & NOT Pit(x) IFF OK(x)

PEAS Description:

1. Performance measure:
   (a) +1000 points for picking up the gold — this is the goal of the agent
   (b) −1000 points for dying = entering a square containing a pit or a live Wumpus Monster
   (c) −1 point for each action taken, and
   (d) −10 points for using the arrow trying to kill the Wumpus -- so that the agent should avoid performing unnecessary actions.

2. Environment: A 4 × 4 grid of squares with. . .
   (a) the agent starting from square [1, 1] facing right
   (b) the gold in one square
   (c) the initially live Wumpus in one square, from which it never moves
   (d) maybe pits in some squares.
   The starting square [1, 1] has no Wumpus, no pit, and no gold – so the agent neither dies nor succeeds straight away.

3. **Actuators:**The agent can turn 90° left or right walk one square forward in the current direction, grab an object in this square, shoot the single arrow in the current direction, which flies in a straight line until it hits a wall or the Wumpus.

4. **Sensors:**The agent has 5 true/false sensors which report a stench when the Wumpus is in an adjacent square — directly, not diagonally breeze when an adjacent square has a pit glitter, when the agent perceives the glitter of the gold in the current square bump, when the agent walks into an enclosing wall (and then the action had no effect) scream, when the arrow hits the Wumpus, killing it.

Program:

```
import java.util.*;
class Environment
{Scanner scr=new Scanner(System.in);
int np;     //number of pits
int wp,gp; // wumpus position   gold position
int pos[]; // position of pits
int b_pos[]=new int[20];
int s_pos[]=new int[20];
void accept(String w[][])
{for(int i=0;i<20;++i)
{b_pos[i]=-1;
s_pos[i]=-1;}
```

```
for(int i=0;i<5;++i)
for(int j=0;j<5;++j)
w[i][j]="";
int count=1;
System.out.println("\n\n********* Wumpus World Problem *********\n");
System.out.println("The positions are as follows.");
for(int i=1;i<=4;++i) {System.out.println("\n----------
---"); System.out.print("|\t");
for(int j=1;j<=4;++j)
System.out.print((count++)+"\t|\t");}
System.out.println("\n----------");
System.out.println("\nAgent start position: 13");
w[4][1]="A";
System.out.println("\nEnter the number of pits.");
np=scr.nextInt();
pos=new int[np];
System.out.println("Positions of pit, gold and wumpus should not overlap.");
System.out.println("Enter the position of pits.");
for(int i=0;i<np;++i)
{pos[i]=scr.nextInt();
show_sense(pos[i],1,w);}
System.out.println("Enter the position of wumpus.");
wp=scr.nextInt();
show_sense(wp,2,w);
System.out.println("Enter the position of gold.");
gp=scr.nextInt();
insert(w);}
void insert(String w[][])
{int temp=0;
int count=0;
int flag1=0,flag2=0;
for(int i=0;i<np;++i)
{temp=pos[i];
count=0;
for(int j=1;j<=4;++j)
{for(int k=1;k<=4;++k)
{++count;
if(count==temp)
w[j][k]+="P";
else if(count==gp&& flag1==0)
{w[j][k]+="G";
flag1=1;}
Else if(count==wp&& flag2==0)
{w[j][k]+="W"; flag2=1;}}}}
display(w);}
void show_sense(int a,intb,String w[][])
{int t1,t2,t3,t4;
t1=a-1;
t2=a+1;
t3=a+4;
```

```
t4=a-4;
if(a==5 || a==9)
t1=0;
if(a==8 || a==12)
t2=0;
if(a==4)
t2=0;
if(a==13)
t1=0;
if(t3>16)
t3=0;
if(t4<0)
t4=0;
if(b==1)
{b_pos[0]=t1;
b_pos[1]=t2;
b_pos[2]=t3;
b_pos[3]=t4; }
else if(b==2)
{s_pos[0]=t1;
s_pos[1]=t2;
s_pos[2]=t3;
s_pos[3]=t4;}
int temp1,count;
for(int i=0;i<4;++i)
{if(b==1)
temp1=b_pos[i];
else
temp1=s_pos[i];
count=0;
for(int j=1;j<=4;++j)
{for(int k=1;k<=4;++k)
{++count;
if(count==temp1 && b==1 && !w[j][k].contains("B"))
{w[j][k]+="B";}
Else
if(count==temp1 && b==2 && !w[j][k].contains("S"))
w[j][k]+="S";}}}}
void display(String w[][])
{System.out.println("\nThe environment for problem is as follows.\n");
for(int i=1;i<=4;++i)
{System.out.println("\n-----------------------------------------------------------------");
System.out.print("|\t");
for(int j=1;j<=4;++j)
System.out.print(w[i][j]+"\t|\t");
}
System.out.println("\n-----------------------------------------------------------------");
}}
class tiles
```

```
{int safe=0;int unsafe=0;int wump=0;int pit=0;int gold=0;int doubt_pit=0;int
doubt_wump=0;String env;
int num=0;int br=0;int bl=0;int bu=0;int bd=0;
int visited=0;int l,r,u,d;
String back="";
tiles(String s,int n)
{env=s;
num=n;
l=r=u=d=0;
if(n==9 || n==5) bl=1;
if(n==8 || n==12)
br=1;
if(n==1)
{bu=1;bl=1;}
if(n==13)
{bd=1;bl=1;}
if(n==4)
{bu=1;br=1;}
if(n==16)
{bd=1;br=1;}}
int sense()
{if(env.contains("B"))
return 1;
else if
(env.contains("S"))
return 2;
else if(env.contains("G"))
return 3;
if(env.contains("W"))
return 4;
else
return 0;}}
class wumpus
{static int scream=0;static int score=0;static int complete=0;
static boolean check(tiles t)
{int temp=t.sense();
if(temp==1 || temp==2)
return false;
return true;}
public static void main(String args[])
{Scanner scr=new Scanner(System.in);
Environment e=new Environment();
String w[][]=new String[5][5];
e.accept(w);
System.out.println("\n\nFinding the solution...");
tiles t[]=new tiles[17];
int c=1;
out:for(int i=1;i<5;++i)
{for(int j=1;j<5;++j)
{if(c>16)
```

```
break out;
t[c]=new tiles(w[i][j],c);
++c;}}
t[13].safe=1;
t[13].visited=1;
int pos=13;
int condition;
int limit=0;
String temp1,temp2;
do
{++limit; condition=-1;
if(t[pos].env.contains("G"))
{complete=1;
System.out.println("Gold Found!!");
break;}
if(t[pos].br!=1 && t[pos].r!=1 &&
t[pos+1].doubt_pit<1
&& t[pos+1].doubt_wump<1 && t[pos+1].pit!=1 && t[pos+1].wump!=1 &&
!(t[pos].back.contains("r") && (t[pos].l!=1 || t[pos].u!=1 || t[pos].d!=1) && check(t[pos]) ))
{temp1="l";
t[pos].r=1;
++pos;
System.out.println("\nfrontpos="+pos);
++score;
t[pos].back+=temp1;
condition=t[pos].sense();
if(condition==3)
{complete=1;break;}
else
if(condition==1 && t[pos].visited==0)
{if(t[pos].br!=1 && t[pos+1].safe!=1)
t[pos+1].doubt_pit+=1;
if(t[pos].bu!=1 && (pos-4)>=1 && t[pos-4].safe!=1)
t[pos-4].doubt_pit+=1;
if(t[pos].bl!=1 && t[pos-1].safe!=1 )
t[pos-1].doubt_pit+=1;
if(t[pos].bd!=1 && (pos+4)<=16 && t[pos+4].safe!=1)
t[pos+4].doubt_pit+=1;
t[pos].safe=1;}
else if(condition==2 && t[pos].visited==0)
{if(t[pos].br!=1 && t[pos+1].safe!=1)
t[pos+1].doubt_wump+=1;
if(t[pos].bu!=1 && (pos-4)>=1 &&  t[pos-4].safe!=1)
t[pos-4].doubt_wump+=1;
if(t[pos].bl!=1 && t[pos-1].safe!=1)
t[pos-1].doubt_wump+=1;
if(t[pos].bd!=1 && (pos+4)<=16 &&  t[pos+4].safe!=1)
t[pos+4].doubt_wump+=1;
t[pos].safe=1;}
else
```

```
if(condition==0)
t[pos].safe=1; t[pos].visited=1;}
else
1].doubt_pit<1 && t[pos-1].doubt_wump<1 &&
t[pos-1].pit!=1 && t[pos-1].wump!=1 &&
!(t[pos].back.contains("l") && (t[pos].r!=1 || t[pos].u!=1 || t[pos].d!=1)
&& check(t[pos]) ))
{temp1="r";
t[pos].l=1;
pos=pos-1;
System.out.println("\nbackpos= "+pos);
++score;
t[pos].back+=temp1;
condition=t[pos].sense();
if(condition==3)
{complete=1;break;}
else
if(condition==1 && t[pos].visited==0)
{if(t[pos].br!=1 && t[pos+1].safe!=1)
t[pos+1].doubt_pit+=1;
if(t[pos].bu!=1 && (pos-4)>=1 &&  t[pos-4].safe!=1)
t[pos-4].doubt_pit+=1;
if(t[pos].bl!=1 && t[pos-1].safe!=1)
t[pos-1].doubt_pit+=1;
if(t[pos].bd!=1 && (pos+4)<=16 &&  t[pos+4].safe!=1)
t[pos+4].doubt_pit+=1; t[pos].safe=1;}
else
if(condition==2 && t[pos].visited==0)
{if(t[pos].br!=1 && t[pos+1].safe!=1)
t[pos+1].doubt_wump+=1;
if(t[pos].bu!=1 && (pos-4)>=1 &&  t[pos-4].safe!=1)
t[pos-4].doubt_wump+=1;
if(t[pos].bl!=1 && t[pos-1].safe!=1)
t[pos-1].doubt_wump+=1;
if(t[pos].bd!=1 && (pos+4)<=16 &&  t[pos+4].safe!=1)
t[pos+4].doubt_wump+=1;
t[pos].safe=1;}
else
if(condition==0)
t[pos].safe=1;
t[pos].visited=1;}
else
if(t[pos].bu!=1 && t[pos].u!=1 && (pos-4)>=1 &&
t[pos-4].doubt_pit<1 && t[pos-4].doubt_wump<1
&& t[pos-4].pit!=1 && t[pos-1].wump!=1 &&
!(t[pos].back.contains("u") && (t[pos].l!=1 || t[pos].r!=1 || t[pos].d!=1)
&& check(t[pos])  ))
{temp1="d"; t[pos].u=1; pos=pos-4;
System.out.println("\nUppos= "+pos);
++score;
```

```
t[pos].back+=temp1;
condition=t[pos].sense();
if(condition==3)
{complete=1;break;}
else
if(condition==1 && t[pos].visited==0)
{if(t[pos].br!=1 && t[pos+1].safe!=1)
t[pos+1].doubt_pit+=1;
if(t[pos].bu!=1 && (pos-4)>=1 &&  t[pos-4].safe!=1)
t[pos-4].doubt_pit+=1;
if(t[pos].bl!=1 && t[pos-1].safe!=1)
t[pos-1].doubt_pit+=1;
if(t[pos].bd!=1 && (pos+4)<=16 &&  t[pos+4].safe!=1)
t[pos+4].doubt_pit+=1;t[pos].safe=1;}
else
if(condition==2 && t[pos].visited==0)
{if(t[pos].br!=1 && t[pos+1].safe!=1)
t[pos+1].doubt_wump+=1;
if(t[pos].bu!=1 && (pos-4)>=1 &&  t[pos-4].safe!=1)
t[pos-4].doubt_wump+=1;
if(t[pos].bl!=1 && t[pos-1].safe!=1)
t[pos-1].doubt_wump+=1;
if(t[pos].bd!=1 && (pos+4)<=16 &&  t[pos+4].safe!=1)
t[pos+4].doubt_wump+=1;
t[pos].safe=1;}
else
if(condition==0)
t[pos].safe=1; t[pos].visited=1;}
else
if(t[pos].bd!=1 && t[pos].d!=1 && (pos+4)<=16 &&
t[pos+4].doubt_pit<1 && t[pos+4].doubt_wump<1 &&
t[pos+4].pit!=1 && t[pos+4].wump!=1)
{temp1="u";
t[pos].d=1;
pos=pos+4;
System.out.println("\ndownpos= "+pos);
++score;
t[pos].back+=temp1;
condition=t[pos].sense();
if(condition==3)
{complete=1;break;}
else
if(condition==1 && t[pos].visited==0)
{if(t[pos].br!=1 && t[pos+1].safe!=1)
t[pos+1].doubt_pit+=1;
if(t[pos].bu!=1 && (pos-4)>=1 &&  t[pos-4].safe!=1)
t[pos-4].doubt_pit+=1;
if(t[pos].bl!=1 && t[pos-1].safe!=1)
t[pos-1].doubt_pit+=1;
if(t[pos].bd!=1 && (pos+4)<=16 &&
```

```
t[pos+4].safe!=1) t[pos+4].doubt_pit+=1; t[pos].safe=1;}
else
if(condition==2 && t[pos].visited==0)
{if(t[pos].br!=1 && t[pos+1].safe!=1)
t[pos+1].doubt_wump+=1;
if(t[pos].bu!=1 && (pos-4)>=1 &&
t[pos-4].safe!=1) \t[pos-4].doubt_wump+=1;
if(t[pos].bl!=1 && t[pos-1].safe!=1)
t[pos-1].doubt_wump+=1;
if(t[pos].bd!=1 && (pos+4)<=16 &&  t[pos+4].safe!=1)
t[pos+4].doubt_wump+=1;
t[pos].safe=1;}
else
if(condition==0)
t[pos].safe=1; t[pos].visited=1;}
else
if(limit>50)
{int temp3=pos; int flag_1=0,flag2=0,flag3=0,flag4=0;
System.out.println("\nCurrently at position "+temp3+".\nThinking....");
while(t[pos].visited==1 && t[pos].br!=1)
{++pos;++score;}
if(t[pos].pit==1 || t[pos].wump==1 ||
(t[pos].br==1 && t[pos].visited==1 && t[pos].safe!=1))
{pos=temp3;flag_1=1;}
if(flag_1==0)
t[pos].back+="l";
while(pos+4>=1 && t[pos].bu!=1
&& t[pos].visited==1)
{pos-=4;++score;}
if(t[pos].pit==1 || t[pos].wump==1 ||
(t[pos].bu==1 && t[pos].visited==1
&& t[pos].safe!=1)) {pos=temp3;flag3=1;}
if(flag3==0)
t[pos].back+="d";
while(t[pos].visited==1 && t[pos].bl!=1)
{--pos;++score;}
if(t[pos].pit==1 || t[pos].wump==1 ||
(t[pos].bl==1 && t[pos].visited==1
&& t[pos].safe!=1))
{pos=temp3;flag2=1;}
if(flag2==0) t[pos].back+="r";
while(pos+4<=16 && t[pos].bd!=1 &&
t[pos].visited==1)
{pos+=4;++score;}
if(t[pos].pit==1 ||
t[pos].wump==1 || (t[pos].bd==1 &&
t[pos].visited==1 &&
t[pos].safe!=1))
{pos=temp3;flag4=1;}
if(flag4==0) t[pos].back+="u";t[pos].safe=1;t[pos].visited=1;
```

```
System.out.println("reached at position "+pos);
limit=0;}
if(t[pos].env.contains("W") &&
scream!=1) {score+=100; scream=1;
t[pos].safe=1;
System.out.println("\n\nWumpus killed >--0-->");
t[pos].env.replace("W"," ");
for(int l=1;l<=16;++l)
{t[l].doubt_wump=0;t[l].env.replace("S"," ");}}
if(t[pos].env.contains("P"))
{score+=50;t[pos].pit=1;
System.out.println("\n\nFallen in pit of position "+pos+".");}
for(int k=1;k<=16;++k)
{if(t[k].doubt_pit==1 &&
t[k].doubt_wump==1)
{t[k].doubt_pit=0;
t[k].doubt_wump=0;
t[k].safe=1;}}
for(int y=1;y<=16;++y)
{if(t[y].doubt_wump>1)
{t[y].wump=1;
for(int h=1;h<=16;++h)
{if(h!=y)
{t[h].doubt_wump=0;
t[h].env.replace("S"," ");}}}}
for(int y=1;y<=16;++y)
{if(t[y].doubt_pit>1)
{t[y].pit=1;}}
try{Thread.sleep(200);}catch(Exception p){}}
while(complete==0);
if(complete==1)
{score*=-1;score+=1000;}
System.out.println("The score of the agent till he
reaches gold is "+score+".\nNow he will return
back following the best explored path.");}}
```

Output

```
********** Wumpus World Problem ********** The positions are as follows.
|     1     |     2     |     3     |     4     |
|     5     |     6     |     7     |     8     |
|     9     |     10    |     11    |     12    |
|     13    |     14    |     15    |     16    |

Agent start position: 13
Enter the number of pits.          3
Positions of pit, gold and wumpus
should not overlap.
Enter the position of pits.        4 7 15
Enter the position of wumpus.              5
Enter the position of gold.        6
```

The environment for problem is as follows.

S		*B*	*P*
W	*BSG*	*P*	*B*
S		*B*	
A	*B*	*P*	*B*

Finding the solution...
front pos=14
back pos= 13
Up pos= 9
front pos=10
front pos=11
back pos= 10
Up pos= 6
Gold Found!! The score of the agent till he reaches gold is 993.
Now he will return back following the best explored path.

Experiment No. 3

Uniformed Search Techniques

3.1 Implement Path Finding in Maze Using Depth-First Search

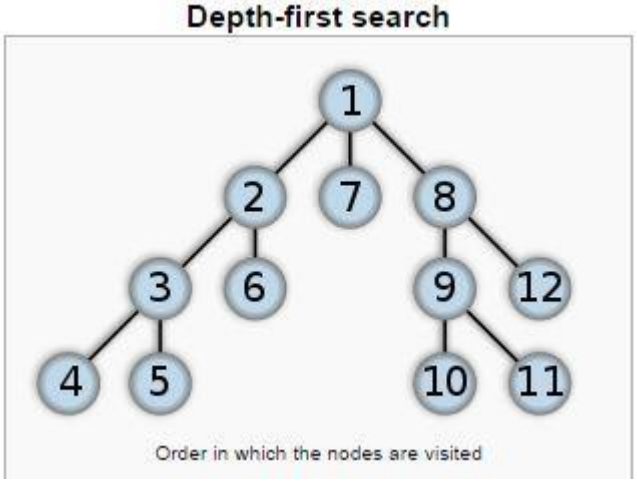
Aim: Path finding in maze using depth-first search (DFS).

Theory:

1. Maze generation algorithms are automated methods for the creation of mazes.
2. A maze can be generated by starting with a predetermined arrangement of cells (most commonly a rectangular grid but other arrangements are possible) with wall sites between them.
3. This predetermined arrangement can be considered as a connected graph with the edges representing possible wall sites and the nodes representing cells.
4. The purpose of the maze generation algorithm can then be considered to be making a sub graph, where it is challenging to find a route between two particular nodes.

Depth-First Search:

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



Program:

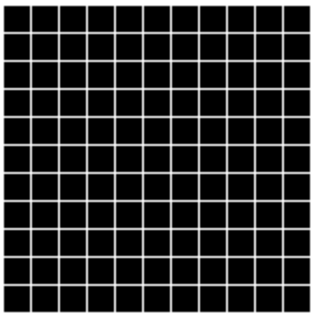
Depth-first search is an algorithm that can be used to generate a maze. The idea is really simple and easy to implement using recursive method or stack.

Basically, you start from a random point and keep digging paths in one of 4 directions(up, right, down, left) until you can't go any further. Once you are stuck, you take a step back until you find an open path. You would continue digging from there. It's just the repetition of these.

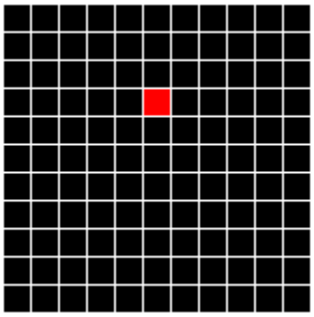
First of all, I would like to explain the general idea a little deeper which you can apply using your choice of programming language. After you have the picture in your mind, you can take a look at the sample code and applet in java.

Explanation

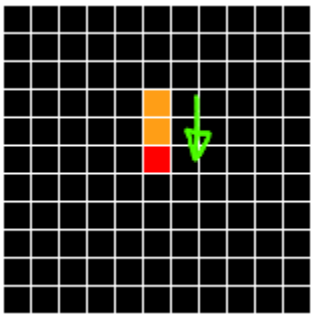
Create a 2-dimensional int array with odd row and column size. 0 represents paths(orange cell) and 1 would be walls(black cell).



Set all cells to 1(wall). There are no paths right now.

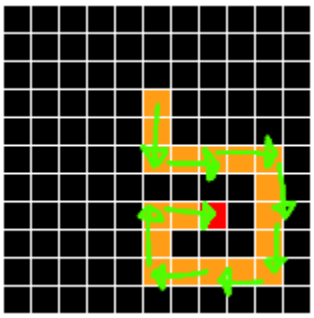


Next, let's set the starting point. Generate odd numbers for row and col. Set that cell to 0. Use row and col variables to keep track of current location. On the picture above, it would be row = 3, col = 5. For clarity, I will be filling the current cell with red.

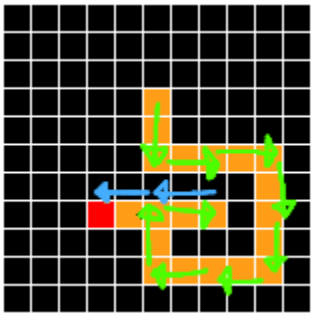


Now, choose a random direction(up, right, down, or left) you are moving to. You will always be moving by 2 cells. The picture above illustrates the current cell moving down. There are couple things you need to check when you move. First, you need to check if 2 cells ahead of that direction is outside of the maze. Then, you check if 2 cells ahead is a path(0) or wall(1). If it's a wall, you

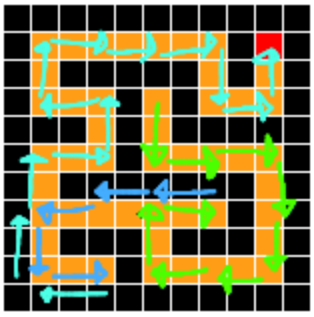
can move by setting these 2 cells to 0(path). Update your current location which is row=5, col=5 at this moment.



As you keep digging as above, you notice that you get to a dead end. In this case, keep moving your current cell to previous cells until you are able to move to a new direction. This is called backtracking. Current location is at row=7, col=7, so you would be moving back to row=7, col=5 on the picture above. You can implement this logic using recursive method or stack.



So you keep digging as the picture demonstrates. For better visual, I changed the color of arrow every time it hits a dead end.



Lastly, this is the final result. With this size, it's just natural that the maze gets too simple. The bigger the maze, the more complicated it will get.

Sample Applet

Using Recursive Method

After you choose your starting point, pass that information to the recursive method. In the recursive method, you can do the following…

1. Generate an int array with 4 random numbers to represent directions.
2. Start a for loop to go for 4 times.
3. Set up a switch statement to take care of 4 directions.
4. For that direction, check if the new cell will be out of maze or if it's a path already open. If so, do nothing.
5. If the cell in that direction is a wall, set that cell to path and call recursive method passing the new current row and column.
6. Done.

Sample code

```
1   public int[][] generateMaze() {
2     int[][] maze = new int[height][width];
3     // Initialize
4     for (int i = 0; i < height; i++)
5        for (int j = 0; j < width; j++)
6           maze[i][j] = 1;
7
8     Random rand = new Random();
9     // r for row、 c for column
10    // Generate random r
11    int r = rand.nextInt(height);
12    while (r % 2 == 0) {
13       r = rand.nextInt(height);
14    }
15    // Generate random c
16    int c = rand.nextInt(width);
17    while (c % 2 == 0) {
18       c = rand.nextInt(width);
19    }
20    // Starting cell
21    maze[r][c] = 0;
22
23    //    Allocate the maze with recursive method
24    recursion(r, c);
25
26    return maze;
27  }
28
29  public void recursion(int r, int c) {
30    // 4 random directions
31    int[] randDirs = generateRandomDirections();
32    // Examine each direction
33    for (int i = 0; i < randDirs.length; i++) {
```

```
34
35          switch(randDirs[i]){
36          case 1: // Up
37             //    Whether 2 cells up is out or not
38             if (r - 2 <= 0)
39                continue;
40             if (maze[r - 2][c] != 0) {
41                maze[r-2][c] = 0;
42                maze[r-1][c] = 0;
43                recursion(r - 2, c);
44             }
45             break;
46          case 2: // Right
47             // Whether 2 cells to the right is out or not
48             if (c + 2 >= width - 1)
49                continue;
50             if (maze[r][c + 2] != 0) {
51                maze[r][c + 2] = 0;
52                maze[r][c + 1] = 0;
53                recursion(r, c + 2);
54             }
55             break;
56          case 3: // Down
57             // Whether 2 cells down is out or not
58             if (r + 2 >= height - 1)
59                continue;
60             if (maze[r + 2][c] != 0) {
61                maze[r+2][c] = 0;
62                maze[r+1][c] = 0;
63                recursion(r + 2, c);
64             }
65             break;
66          case 4: // Left
67             // Whether 2 cells to the left is out or not
68             if (c - 2 <= 0)
69                continue;
70             if (maze[r][c - 2] != 0) {
71                maze[r][c - 2] = 0;
72                maze[r][c - 1] = 0;
73                recursion(r, c - 2);
74             }
75             break;
76          }
77       }
78
79    }
80
81    /**
82    * Generate an array with random directions 1-4
```

```
83  * @return Array containing 4 directions in random order
84  */
85  public Integer[] generateRandomDirections() {
86      ArrayList<Integer> randoms = new ArrayList<Integer>();
87      for (int i = 0; i < 4; i++)
88          randoms.add(i + 1);
89      Collections.shuffle(randoms);
90
91     return randoms.toArray(new Integer[4]);
92  }
```

3.2 Implement Water Jug Problem Using Breadth-First Search

Aim: Implement water jug problem using Breadth-First Search (BFS).

Theory:

You are given a m litre jug and a n litre jug . Both the jugs are initially empty. The jugs don't have markings to allow measuring smaller quantities. You have to use the jugs to measure d litres of water where d is less than n.
(X, Y) corresponds to a state where X refers to amount of water in Jug1 and Y refers to amount of water in Jug2
Determine the path from initial state (xi, yi) to final state (xf, yf), where (xi, yi) is (0, 0) which indicates both Jugs are initially empty and (xf, yf) indicates a state which could be (0, d) or (d, 0).

The operations you can perform are:

1. Empty a Jug, (X, Y)->(0, Y) Empty Jug 1
2. Fill a Jug, (0, 0)->(X, 0) Fill Jug 1
3. Pour water from one jug to the other until one of the jugs is either empty or full, (X, Y) -> (X-d, Y+d)

Examples:

Input : 4 3 2

Output : {(0, 0), (0, 3), (4, 0), (4, 3),

(3, 0), (1, 3), (3, 3), (4, 2),

(0, 2)}

Algorithm:

```
#include <bits/stdc++.h>
#define pii pair<int, int>
#define mp make_pair
using namespace std;

void BFS(int a, int b, int target)
{
   // Map is used to store the states, every
   // state is hashed to binary value to
   // indicate either that state is visited
   // before or not
   map<pii, int> m;
   bool isSolvable = false;
   vector<pii> path;

   queue<pii> q; // queue to maintain states
   q.push({ 0, 0 }); // Initialing with initial state

   while (!q.empty()) {
```

```
pii u = q.front(); // current state

q.pop(); // pop off used state

// if this state is already visited
if (m[{ u.first, u.second }] == 1)
    continue;

// doesn't met jug constraints
if ((u.first > a || u.second > b ||
    u.first < 0 || u.second < 0))
    continue;

// filling the vector for constructing
// the solution path
path.push_back({ u.first, u.second });

// marking current state as visited
m[{ u.first, u.second }] = 1;

// if we reach solution state, put ans=1
if (u.first == target || u.second == target) {
    isSolvable = true;
    if (u.first == target) {
        if (u.second != 0)

            // fill final state
            path.push_back({ u.first, 0 });
    }
    else {
        if (u.first != 0)

            // fill final state
            path.push_back({ 0, u.second });
    }

    // print the solution path
    int sz = path.size();
    for (int i = 0; i < sz; i++)
        cout << "(" << path[i].first
             << ", " << path[i].second << ")\n";
    break;
}

// if we have not reached final state
// then, start developing intermediate
// states to reach solution state
q.push({ u.first, b }); // fill Jug2
q.push({ a, u.second }); // fill Jug1
```

```
        for (int ap = 0; ap <= max(a, b); ap++) {

            // pour amount ap from Jug2 to Jug1
            int c = u.first + ap;
            int d = u.second - ap;

            // check if this state is possible or not
            if (c == a || (d == 0 && d >= 0))
                q.push({ c, d });

            // Pour amount ap from Jug 1 to Jug2
            c = u.first - ap;
            d = u.second + ap;

            // check if this state is possible or not
            if ((c == 0 && c >= 0) || d == b)
                q.push({ c, d });
        }

        q.push({ a, 0 }); // Empty Jug2
        q.push({ 0, b }); // Empty Jug1
    }

    // No, solution exists if ans=0
    if (!isSolvable)
        cout << "No solution";
}

// Driver code
int main()
{
    int Jug1 = 4, Jug2 = 3, target = 2;
    cout << "Path from initial state "
            "to solution state :\n";
    BFS(Jug1, Jug2, target);
    return 0;
}
```

Output:

Path from initial state to solution state ::

(0, 0)

(0, 3)

(4, 0)

(4, 3)

(3, 0)

(1, 3)

(3, 3)

(4, 2)

(0, 2)

Experiment No. 4

Informed Search Technique

4.1 Implement 8-Puzzle Problem Using Hill Climbing

Aim: Implement 8-puzzle problem with heuristic function using hill climbing (informed search)

Theory:
In an 8-puzzle game, we need to rearrange some tiles to reach a predefined goal state. Consider the following 8-puzzle board.

This is the goal state where each tile is in correct place. In this game, you will be given a board where the tiles aren't in the correct places. You need to move the tiles using the gap to reach the goal state.

Suppose f (n) can be defined as: the number of misplaced tiles.

In the above figure, tiles 6, 7 and 8 are misplaced. So f (n) = 3 for this case.

For solving this problem with hill climbing search, we need to set a value for the heuristic. Suppose the heuristic function h (n) is the lowest possible f (n) from a given state. First, we need to know all the possible moves from the current state. Then we have to calculate f(n) (number of misplaced tiles) for each possible move. Finally we need to choose the path with lowest possible f (n) (which is our h (n) or heuristic).

Consider the figure above. Here, 3 moves are possible from the current state. For each state we have calculated f (n). From the current state, it is optimal to move to the state with f (n) = 3 as it is closer to the goal state. So we have our h (n) = 3.

However, do you really think we can guarantee that it will reach the goal state? What will you do if you reach on a state(Not the goal state) from which there are no better neighbour states! This condition can be called a local maxima and this is the problem of hill climbing search. Therefore, we may get stuck in local maxima. In this scenario, you need to backtrack to a previous state to perform the search again to get rid of the path having local maxima.

What will happen if we reach to a state where all the f (n) values are equal? This condition is called a plateau. You need to select a state at random and perform the hill climbing search again!

Input:

You will be given the initial state of the board. The input will be given by row by row of the 3x3 grid of the 8-puzzle game. A digit from 1 to 9 will denote a tiles number. A 0 will denote the gap of the board.

For the above board, the input will be:

1 2 3

7 8 4

6 0 5

Task 1: Print the total cost (number of steps) needed to reach the goal state (if possible). Report if you reach a local maxima and get stuck. Print the board state in this scenario.

Task 2: You need get rid of any local maxima and reach the goal state anyway. Print the total cost needed to reach the goal state. Mention where you have backtracked to avoid local maxima (if any).

Program:

Main.java

```
public
class
Main
{
        public static void main(String[] args) {
            Eight_Puzzle eight_Puzzle = new Eight_Puzzle();
            eight_Puzzle.initializations();
        }
    }
```

Priority.java

```
import java.util.Arrays;

Public
class
Priority
{
```

```
   static int[][] preState;//keeps the previous state
    static Node neighbors_nodeArray[];
    //takes an node array, sort nodes based on distance of fn
    //and return lowest fn node
    public static Node sort(Node[] nodeArray) {

        if(preState!=null){//parent exists
            nodeArray = getParentRemovedNodeArray(nodeArray, preState);//remove
parent
        }

        //sorting nodes based on fn
        for (int i = 0; i < nodeArray.length; i++) {
            for (int j = nodeArray.length - 1; j > i; j--) {
                if (nodeArray[j].fn < nodeArray[j - 1].fn) {
                    Node temp = nodeArray[j];
                    nodeArray[j] = nodeArray[j - 1];
                    nodeArray[j - 1] = temp;
                }
            }
        }
        Priority.neighbors_nodeArray = nodeArray;
        return nodeArray[0];
    }
    //takes node array and prestate
    //remove the neighbor which same as prestate and return parent removed node
array
    public static Node[] getParentRemovedNodeArray(Node []nodeArray, int[][]
preState) {
        Node[] parentRemovedNodeArray = new Node[nodeArray.length - 1];
        int j = 0;
        for (int i = 0; i < nodeArray.length; i++) {
            if (Arrays.deepEquals(nodeArray[i].state, preState)) {
                //System.out.println("removed parent");
            } else {
                parentRemovedNodeArray[j] = nodeArray[i];
                j++;
            }
        }
        return parentRemovedNodeArray;
    }
}
//Node class
class Node {
    int fn;//fn value
    int[][] state;//states
    int [][] parent;
    public Node(int fn, int[][] state, int[][]parent) {
        this.fn = fn;
        this.state = state;
```

```
        this.parent = parent;
    }
}
```

Eight_Puzzle.java

```
import
java.util.Rando
m;
                    import java.util.Stack;
                    public class Eight_Puzzle {
                        //solution state of the 8-puzzle game
                        int goal_state[][] = {
                            {1, 2, 3},
                            {8, 0, 4},
                            {7, 6, 5}
                        };
                        //problem board of 8-puzzle game
                        int game_board[][] = {
                            {2, 6, 3},
                            {1, 0, 4},
                            {8, 7, 5}
                        };
                        /* one local maxima input example input
                         {2, 8, 3},
                         {1, 6, 4},
                         {7, 0, 5}
                         */
                        /* one solved input example
                         {1, 3, 4},
                         {8, 2, 5},
```

{0, 7, 6}

{2, 0, 6},

{1, 4, 3},

{8, 7, 5}

//nice backtrack not solved in local maxima

{2, 6, 3},

{1, 0, 4},

{8, 7, 5}

**/*

/ one no backtrack local maxima test input example*

{1, 4, 0},

{8, 3, 2},

{7, 6, 5}

**/*

/ one impossible local maxima test input example*

{1, 2, 0},

{8, 3, 4},

{7, 6, 5}

using best solved

{8, 2, 3},

{0, 6, 4},

{7, 1, 5}

not using best

{1, 0, 2},

{8, 3, 6},

{7, 4, 5}

**/*

//initial empty tile position

```
    int emptyTile_row = 0;
    int emptyTile_col = 0;
    int stepCounter = 0;
    int min_fn;
    Node min_fn_node;
    Random random = new Random();
    Stack<Node> stack_state = new Stack<>();//for backtracking
    //initializations
    public void initializations() {
        locateEmptyTilePosition();//set empty tile position
        min_fn = get_fn(game_board);//-? min fn

System.out.println("================================
=========");
        printState(game_board, "initial problem state");
        System.out.println("initial empty tile position: " + emptyTile_row + ",
" + emptyTile_col);
        System.out.println("initial fn (number of misplaced tiles): " + min_fn);

System.out.println("================================
=========");
        //start hill climbing search
        try {
            hill_climbing_search();
        } catch (Exception e) {
            System.out.println("Goal can not be reached, found closest solution
state");
            printState(min_fn_node.state, "---------solution state------with min
fn " + min_fn);
        }
    }
    //start hill climbing search for 8-puzzle problem
    public void hill_climbing_search() throws Exception {
        while (true) {
```

```
System.out.println(">==============================
==========<");
            System.out.println("cost/steps: " + (++stepCounter));
            System.out.println("-------------");
            //Priority.preState = game_board;//change pre state
            Node lowestPossible_fn_node = getLowestPossible_fn_node();
            addToStackState(Priority.neighbors_nodeArray);//add    neighbors
to stack in high to low order fn
            printState(lowestPossible_fn_node.state, "-------new state");
            //print all fn values
//            System.out.print("all sorted fn of current state: ");
//            for (int i = 0; i < Priority.neighbors_nodeArray.length; i++) {
//                System.out.print(Priority.neighbors_nodeArray[i].fn + " ");
//            }
//            System.out.println();
            //check for local maxima
            int fnCounter = 1;
            for (int i = 1; i < Priority.neighbors_nodeArray.length; i++) {
                if       (Priority.neighbors_nodeArray[i       -       1].fn       ==
Priority.neighbors_nodeArray[i].fn) {//fns are equal
                    fnCounter++;
                }
            }
            if (Priority.neighbors_nodeArray.length != 1 && fnCounter ==
Priority.neighbors_nodeArray.length) {//all fns are equal, equal chances to
choose
                System.out.println("---fn's are equal, found in local maxima---");
                //backtracking
                for (int i = 0; i < Priority.neighbors_nodeArray.length; i++) {
                    if (stack_state != null) {
                        System.out.println("pop " + (i + 1));
                        stack_state.pop();
                    } else {
```

```
                System.out.println("empty stack inside loop");
            }
        }
        if (stack_state != null) {
            Node gameNode = stack_state.pop();
            game_board = gameNode.state;//update game board
            Priority.preState = gameNode.parent;//update prestate
            locateEmptyTilePosition();//locate empty tile for updated state
            printState(game_board, "popped state from all equal fn");
            System.out.println("empty tile position: " + emptyTile_row + ", " + emptyTile_col);
        } else {
            System.out.println("stack empty inside first lm check");
        }
    } else {//for backtracking
        System.out.println("lowest fn: " + lowestPossible_fn_node.fn);
        if (lowestPossible_fn_node.fn == 0) {//no misplaced found
            System.out.println("-------------------------");
            System.out.println("8-Puzzle has been solved!");
            System.out.println("-------------------------");
            System.out.println("Total cost/steps to reach the goal: " + stepCounter);
            System.out.println("-----------------------------------");
            break;
        }
        if (lowestPossible_fn_node.fn <= min_fn) {
            min_fn = lowestPossible_fn_node.fn;
            min_fn_node = lowestPossible_fn_node;//store lowest fn solution
            if (stack_state != null) {
                Node gameNode = stack_state.pop();
                game_board = gameNode.state;//update game board
```

```
                    Priority.preState = gameNode.parent;//update prestate
                    locateEmptyTilePosition();//locate empty tile for updated state
                    printState(game_board, "-------new state as going deeper");
                    System.out.println("empty tile position: " + emptyTile_row + ", " + emptyTile_col);
                } else {
                    System.out.println("stack empty");
                }
            } else {
                System.out.println("---stuck in local maxima---");
                System.out.println("getting higher, not possible");
            //break;
                //backtracking
                for (int i = 0; i < Priority.neighbors_nodeArray.length; i++) {
                    if (stack_state != null) {
                        //System.out.println("pop " + (i + 1));
                        stack_state.pop();
                    } else {
                        System.out.println("empty stack inside loop");
                    }
                }
                if (stack_state != null) {
                    Node gameNode = stack_state.pop();
                    game_board = gameNode.state;//update game board
                    Priority.preState = gameNode.parent;//update prestate
                    locateEmptyTilePosition();//locate empty tile for updated state
                    printState(game_board, "popped state from getting higher");
                    System.out.println("empty tile position: " + emptyTile_row + ", " + emptyTile_col);
                } else {
```

```
                    System.out.println("stack empty inside second lm check");
                }
            }//end of if cond: new fn<=pre min fn
        }//end of if cond: all fn equal
    }//while end
}
private Node getLowestPossible_fn_node() {
    if (emptyTile_row == 0 && emptyTile_col == 0) {//0,0 position is empty tile
        //System.out.println("Empty 0,0");
        Node fn_array[] = {get_fn_down(), get_fn_right()};
        Node lowest_fn_node = Priority.sort(fn_array);
        return lowest_fn_node;
    } else if (emptyTile_row == 0 && emptyTile_col == 1) {//0,1 position is empty tile
        //System.out.println("Empty 0,1");
        Node fn_array[] = {get_fn_left(), get_fn_down(), get_fn_right()};
        Node lowest_fn_node = Priority.sort(fn_array);
        return lowest_fn_node;
    } else if (emptyTile_row == 0 && emptyTile_col == 2) {//0,2 position is empty tile
        //System.out.println("Empty 0,2");
        Node fn_array[] = {get_fn_left(), get_fn_down()};
        Node lowest_fn_node = Priority.sort(fn_array);
        return lowest_fn_node;
    } else if (emptyTile_row == 1 && emptyTile_col == 0) {//1,0 position is empty tile
        //System.out.println("Empty 1,0");
        Node fn_array[] = {get_fn_down(), get_fn_right(), get_fn_up()};
        Node lowest_fn_node = Priority.sort(fn_array);
        return lowest_fn_node;
    } else if (emptyTile_row == 1 && emptyTile_col == 1) {//1,1 position is empty tile
```

```
            //System.out.println("Empty 1,1");
            Node fn_array[] = {get_fn_left(), get_fn_down(), get_fn_right(), get_fn_up()};
            Node lowest_fn_node = Priority.sort(fn_array);
            return lowest_fn_node;
        } else if (emptyTile_row == 1 && emptyTile_col == 2) {//1,2 position is empty tile
            //System.out.println("Empty 1,2");
            Node fn_array[] = {get_fn_left(), get_fn_down(), get_fn_up()};
            Node lowest_fn_node = Priority.sort(fn_array);
            return lowest_fn_node;
        } else if (emptyTile_row == 2 && emptyTile_col == 0) {//2,0 position is empty tile
            //System.out.println("Empty 2,0");
            Node fn_array[] = {get_fn_right(), get_fn_up()};
            Node lowest_fn_node = Priority.sort(fn_array);
            return lowest_fn_node;
        } else if (emptyTile_row == 2 && emptyTile_col == 1) {//2,1 position is empty tile
            //System.out.println("Empty 2,1");
            Node fn_array[] = {get_fn_left(), get_fn_right(), get_fn_up()};
            Node lowest_fn_node = Priority.sort(fn_array);
            return lowest_fn_node;
        } else if (emptyTile_row == 2 && emptyTile_col == 2) {//2,2 position is empty tile
            //System.out.println("Empty 2,2");
            Node fn_array[] = {get_fn_left(), get_fn_up()};
            Node lowest_fn_node = Priority.sort(fn_array);
            return lowest_fn_node;
        }
        return null;
    }
    //----------------------------
```

```
    //return number of misplaced tiles for left state
    private Node get_fn_left() {
        int left_state[][] = new int[game_board.length][game_board[0].length];
        for (int i = 0; i < game_board.length; i++) {
            for (int j = 0; j < game_board[0].length; j++) {
                if (i == emptyTile_row && j == emptyTile_col) {//empty tile, swap left
                    left_state[i][j] = game_board[i][j - 1];
                    left_state[i][j - 1] = game_board[i][j];
                } else {//normal copy
                    left_state[i][j] = game_board[i][j];
                }
            }
        }
        printState(left_state, "left state");//print left state
        Node node = new Node(get_fn(left_state), left_state, game_board);
        return node;
    }
    //return number of misplaced tiles for right state
    private Node get_fn_right() {
        int right_state[][] = new int[game_board.length][game_board[0].length];
        for (int i = 0; i < game_board.length; i++) {
            for (int j = 0; j < game_board[0].length; j++) {
                if (i == emptyTile_row && j == emptyTile_col) {//empty tile, swap right
                    right_state[i][j] = game_board[i][j + 1];
                    right_state[i][j + 1] = game_board[i][j];
                    j++;//as j++ position already copied/updated
                } else {//normal copy
                    right_state[i][j] = game_board[i][j];
                }
```

```
            }
        }
        printState(right_state, "right state");//print right state
        Node node = new Node(get_fn(right_state), right_state, game_board);
        return node;
    }
    //return number of misplaced tiles for up state
    private Node get_fn_up() {
        int up_state[][] = new int[game_board.length][game_board[0].length];
        for (int i = 0; i < game_board.length; i++) {
            for (int j = 0; j < game_board[0].length; j++) {
                if (i == emptyTile_row && j == emptyTile_col) {//empty tile, swap up
                    up_state[i][j] = game_board[i - 1][j];
                    up_state[i - 1][j] = game_board[i][j];
                } else {//normal copy
                    up_state[i][j] = game_board[i][j];
                }
            }
        }
        printState(up_state, "up state");//print up state
        Node node = new Node(get_fn(up_state), up_state, game_board);
        return node;
    }
    //return number of misplaced tiles for down state
    private Node get_fn_down() {
        int down_state[][] = new int[game_board.length][game_board[0].length];
        for (int i = 0; i < game_board.length; i++) {
            for (int j = 0; j < game_board[0].length; j++) {
```

```
            if ((i - 1) == emptyTile_row && j == emptyTile_col) {//down pos of empty tile, swap down
                down_state[i][j] = game_board[i - 1][j];
                down_state[i - 1][j] = game_board[i][j];
            } else {//normal copy
                down_state[i][j] = game_board[i][j];
            }
        }
    }
    printState(down_state, "down state");//print down state
    Node node = new Node(get_fn(down_state), down_state, game_board);
    return node;
  }
  //takes a game state and returns number of misplaced tiles
  private int get_fn(int[][] game_state) {
    int fn_count = 0;
    for (int i = 0; i < game_state.length; i++) {
      for (int j = 0; j < game_state[0].length; j++) {
        if (game_state[i][j] != goal_state[i][j] && game_state[i][j] != 0) {//found misplaced tiles
          fn_count++;
        }
      }
    }
    return fn_count;
  }
  //takes parent removed,  sorted node array and add states to stack in high to low order
  private void addToStackState(Node nodeArray[]) {
    for (int i = nodeArray.length - 1; i >= 0; i--) {
      stack_state.add(nodeArray[i]);//highest fn to lowest fn
    }
```

```
    }
    //find out the new empty tile position for current state
    private void locateEmptyTilePosition() {
        nestedloop://to break inner and outer loop
        for (int i = 0; i < game_board.length; i++) {
            for (int j = 0; j < game_board[0].length; j++) {
                if (game_board[i][j] == 0) {
                    emptyTile_row = i;
                    emptyTile_col = j;
                    break nestedloop;
                }
            }
        }
    }
    //print the current state of the game board
    private void printState(int[][] state, String message) {
        System.out.println(message);
        for (int i = 0; i < state.length; i++) {
            for (int j = 0; j < state[0].length; j++) {
                System.out.print(state[i][j] + "  ");
            }
            System.out.println();
        }
        System.out.println("--------");
    }
}
```

4.2 Implement 8-Puzzle Problem Using Best-First Search

Aim: Implement 8-puzzle problem with heuristic function – best-first search (informed search)

Theory:

The problem. The 8-puzzle problem is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. Your goal is to rearrange the blocks so that they are in order. You are permitted to slide blocks horizontally or vertically into the blank square. The following shows a sequence of legal moves from an initial board position (left) to the goal position (right).

```
    1  3        1     3        1  2  3        1  2  3        1  2  3
4   2  5   =>   4  2  5   =>   4     5   =>   4  5     =>   4  5  6
7  8  6         7  8  6         7  8  6         7  8  6         7  8

initial                                                      goal
```

Best-first search. We now describe an algorithmic solution to the problem that illustrates a general artificial intelligence methodology known as the A* search algorithm. We define a *state* of the game to be the board position, the number of moves made to reach the board position, and the previous state. First, insert the initial state (the initial board, 0 moves, and a null previous state) into a priority queue. Then, delete from the priority queue the state with the minimum priority, and insert onto the priority queue all neighboring states (those that can be reached in one move). Repeat this procedure until the state dequeued is the goal state. The success of this approach hinges on the choice of *priority function* for a state. We consider two priority functions:

- *Hamming priority function.* The number of blocks in the wrong position, plus the number of moves made so far to get to the state. Intuitively, a state with a small number of blocks in the wrong position is close to the goal state, and we prefer a state that have been reached using a small number of moves.
- *Manhattan priority function.* The sum of the distances (sum of the vertical and horizontal distance) from the blocks to their goal positions, plus the number of moves made so far to get to the state.

For example, the Hamming and Manhattan priorities of the initial state below are 5 and 10, respectively.

```
8  1  3        1  2  3     1  2  3  4  5  6  7  8     1  2  3  4  5  6  7  8
4     2        4  5  6     ----------------------     ----------------------
7  6  5        7  8        1  1  0  0  1  1  0  1     1  2  0  0  2  2  0  3

initial          goal          Hamming = 5 + 0           Manhattan = 10 + 0
```

Program :

```
import
java.util.*;
                public class BDSearch {
                   public static State initialState;
                   public static void search(int[] board) {
                      int[] goalBoard = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 0};
                      initialState = new EightPuzzleState(board);
                      State goalState = new EightPuzzleState(goalBoard);
                      SearchNode root = new SearchNode(initialState);
                      SearchNode goal = new SearchNode(goalState);
                      Queue<SearchNode> forwardQueue = new LinkedList<>();
                      Queue<SearchNode> backwardQueue = new LinkedList<>();
                      forwardQueue.add(root);
                      backwardQueue.add(goal);
                      performSearch(forwardQueue, backwardQueue);
                   }
                   private static boolean checkRepeats(SearchNode n) {
                      boolean retValue = false;
                      SearchNode checkNode = n;
                      // While n's parent isn't null, check to see if it's equal to the node
                      // we're looking for.
                      while (n.getParent() != null && !retValue) {
                         if (n.getParent().getCurState().equals(checkNode.getCurState())) {
                            retValue = true;
                         }
                         n = n.getParent();
                      }
                      return retValue;
                   }
                   private static boolean isInitialState(State state) {
                      return state.equals(initialState);
                   }
                   private static SearchNode queueContainsNode(Queue<SearchNode> q,
                SearchNode targetNode) {
                      for (SearchNode n :
                            q) {
                         if (n.getCurState().equals(targetNode.getCurState())) {
                            return n;
                         }
                      }
                      return null;
                   }
                   private static void bfs(SearchNode n, Queue<SearchNode> q) {
                      ArrayList<State> tempSuccessors = n.getCurState()
                            .genSuccessors(); // generate tempNode's immediate
                      // successors
                      /*
                       * Loop through the successors, wrap them in a SearchNode, check
                       * if they've already been evaluated, and if not, add them to
                       * the queue
```

```
         */
        for (int i = 0; i < tempSuccessors.size(); i++) {
            // second parameter here adds the cost of the new node to
            // the current cost total in the SearchNode
            SearchNode newNode = new SearchNode(n,
                    tempSuccessors.get(i), n.getCost()
                    + tempSuccessors.get(i).findCost(), 0);
            if (!checkRepeats(newNode)) {
                q.add(newNode);
            }
        }
    }
    // The goal state has been found. Print the path it took to get to
    // it.
    private static void success(SearchNode node, int searchCount) {
        // Use a stack to track the path from the starting state to the
        // goal state
        Stack<SearchNode> solutionPath = new Stack<SearchNode>();
        solutionPath.push(node);
        node = node.getParent();
        while (node.getParent() != null) {
            solutionPath.push(node);
            node = node.getParent();
        }
        solutionPath.push(node);
        // The size of the stack before looping through and emptying it.
        int loopSize = solutionPath.size();
        for (int i = 0; i < loopSize; i++) {
            node = solutionPath.pop();
            node.getCurState().printState();
            System.out.println();
            System.out.println();
        }
        System.out.println("The cost was: " + node.getCost());
    }
    private static void performSearch(Queue<SearchNode> fq,
                                      Queue<SearchNode> bq) {
        int searchCount = 1;
        while (!fq.isEmpty() && !bq.isEmpty()) {
            if (!fq.isEmpty()) {
                SearchNode tempNode = fq.poll();
                SearchNode nodeExistOnBackwardQueue =
queueContainsNode(bq, tempNode);
                if (tempNode.getCurState().isGoal() ||
nodeExistOnBackwardQueue != null) {
                    if (nodeExistOnBackwardQueue != null) {
                        success(nodeExistOnBackwardQueue, searchCount);
                    }
                    success(tempNode, searchCount);
                    System.exit(0);
```

```
            } else {
                bfs(tempNode, fq);
                searchCount++;
            }
        }
        if (!bq.isEmpty()) {
            SearchNode tempNode = bq.poll();
            SearchNode nodeExistOnForwardQueue = queueContainsNode(fq,
tempNode);
            if              (isInitialState(tempNode.getCurState())          ||
nodeExistOnForwardQueue != null) {
                success(tempNode, searchCount);
                if (nodeExistOnForwardQueue != null) {
                    success(nodeExistOnForwardQueue, searchCount);
                }
                System.exit(0);
            } else {
                bfs(tempNode, bq);
                searchCount++;
            }
        }
    }
  }
}
```

4.3 Implement Tic-Tac-Toe Using A* algorithm

Aim: Tic-Tac-Toe using A* algorithm.

Theory:A board game (such as tic-tac-toe) is usually programmed as a state machine. looking on the current-state and therefore the player's move, the game goes into the next-state. tit-tat-toe (or Noughts and crosses, Xs and Os) could be a paper and pencil for 2 players, X and O, who take turns marking the areas in an exceedingly 3×3 grid. The player who succeeds in putting 3 individual marks in an exceedingly horizontal, vertical or diagonal row wins the game. Players shortly discover that best play from each party ends up in a draw(often said as cat or cat's game). Hence, tit-tat-toe is most frequently competed by young children. The simplicity of tit-tat-toe makes it ideal as a pedagogical tool for teaching the ideas of fine sportsmanship and therefore the branch of artificial intelligence that deals with the searching of game trees. It is simple to write down a computer program to play tit-tat-toe perfectly, to enumerate the 765 essentially completely different positions (the space highlighting), or the 26,830 possible games up to rotations and reflections (the game tree complexity) on this space. the game is generalized to an m,n,k-game during which 2 players alternate putting stones of their own colour on an m×n board, with the goal of obtaining k of their own colour in a row. tit-tat-toe is the (3,3,3)-game.

Algorithm:

```
public class TripleT {
enum State{Blank, X, O};
 int n = 3;
 State[ ][ ] board = new State[n][n];
   int moveCount;
```

```
 void Move(int x, int y, State s){
        if (board[x][y] == State.Blank){
                board[x][y] = s;
}
moveCount++;
//check end conditions
        //check col
        for(int i = 0; i< n; i++){
                if(board[x][i] != s)
                        break;
                if(i == n-1){
                        //report win for s
                }
}
        //check row
        for (int  i = 0; i< n; i++){
                if (board[i][y] != s)
                        break ;
                if(i == n-1){
                        //report win for s
                }
        }
//check diag
        if(x == y){
                //we're on a diagonal
                for(int i = 0; i< n; i++){
                        if(board[i][i] != s)
                                break ;
                        if(i == n-1){
                                //report win for s
                        }
                }
        }

        //check anti diag (thanks rampion)
        for (int  i=0; i<n; i++){
                if (board[i][(n-1)-i] != s)
                        break  ;
                if (i == n-1){
                        //report win for s
                }
        }

        //check draw
        if (moveCount == (n^2 - 1)){
                //report draw
        }
   }
}
Conclusion:
```

Thus, a winning move will solely happen when X or O has created their most up-to-date move, therefore, one will solely search row/column with optional diag that are contained in this move to limit their search space when trying to work out a winning board. Also, since there are a set range of moves in a draw tit-tat-toe game once the last move is created, if it wasn't a winning move, it is by default a draw game.

Program:

```
import java.util.Scanner;
/**
 * Tic-Tac-Toe: Two-player console, non-graphics, non-OO version.
 * All variables/methods are declared as static (belong to the class)
 *  in the non-OO version.
 */
public class TicTacToe {
   // Name-constants to represent the seeds and cell contents
   public static final int EMPTY = 0;
   public static final int CROSS = 1;
   public static final int NOUGHT = 2;

   // Name-constants to represent the various states of the game
   public static final int PLAYING = 0;
   public static final int DRAW = 1;
   public static final int CROSS_WON = 2;
   public static final int NOUGHT_WON = 3;

   // The game board and the game status
   public static final int ROWS = 3, COLS = 3; // number of rows and columns
   public static int[][] board = new int[ROWS][COLS]; // game board in 2D array
                                       //  containing (EMPTY, CROSS, NOUGHT)
   public static int currentState;  // the current state of the game
                             // (PLAYING, DRAW, CROSS_WON, NOUGHT_WON)
   public static int currentPlayer; // the current player (CROSS or NOUGHT)
   public static int currntRow, currentCol; // current seed's row and column

   public static Scanner in = new Scanner(System.in); // the input Scanner

   /** The entry main method (the program starts here) */
   public static void main(String[] args) {
      // Initialize the game-board and current status
initGame();
      // Play the game once
      do {
playerMove(currentPlayer); // update currentRow and currentCol
updateGame(currentPlayer, currntRow, currentCol); // update currentState
printBoard();
         // Print message if game-over
         if (currentState == CROSS_WON) {
System.out.println("'X' won! Bye!");
         } else if (currentState == NOUGHT_WON) {
```

```
System.out.println("'O' won! Bye!");
        } else if (currentState == DRAW) {
System.out.println("It's a Draw! Bye!");
        }
        // Switch player
currentPlayer = (currentPlayer == CROSS) ? NOUGHT : CROSS;
     } while (currentState == PLAYING); // repeat if not game-over
   }

   /** Initialize the game-board contents and the current states */
   public static void initGame() {
      for (int row = 0; row < ROWS; ++row) {
         for (int col = 0; col < COLS; ++col) {
            board[row][col] = EMPTY;  // all cells empty
         }
      }
currentState = PLAYING; // ready to play
currentPlayer = CROSS;  // cross plays first
   }

   /** Player with the "theSeed" makes one move, with input validation.
       Update global variables "currentRow" and "currentCol". */
   public static void playerMove(int theSeed) {
booleanvalidInput = false;  // for input validation
      do {
         if (theSeed == CROSS) {
System.out.print("Player 'X', enter your move (row[1-3] column[1-3]): ");
         } else {
System.out.print("Player 'O', enter your move (row[1-3] column[1-3]): ");
         }
         int row = in.nextInt() - 1;  // array index starts at 0 instead of 1
         int col = in.nextInt() - 1;
         if (row >= 0 && row < ROWS && col >= 0 && col < COLS && board[row][col] ==
EMPTY) {
currntRow = row;
currentCol = col;
            board[currntRow][currentCol] = theSeed;  // update game-board content
validInput = true;  // input okay, exit loop
         } else {
System.out.println("This move at (" + (row + 1) + "," + (col + 1)
               + ") is not valid. Try again...");
         }
      } while (!validInput);  // repeat until input is valid
   }

   /** Update the "currentState" after the player with "theSeed" has placed on
       (currentRow, currentCol). */
   public static void updateGame(int theSeed, int currentRow, int currentCol) {
      if (hasWon(theSeed, currentRow, currentCol)) {  // check if winning move
currentState = (theSeed == CROSS) ? CROSS_WON : NOUGHT_WON;
```

```
    } else if (isDraw()) {  // check for draw
currentState = DRAW;
    }
    // Otherwise, no change to currentState (still PLAYING).
  }

  /** Return true if it is a draw (no more empty cell) */
  // TODO: Shall declare draw if no player can "possibly" win
  public static booleanisDraw() {
    for (int row = 0; row < ROWS; ++row) {
      for (int col = 0; col < COLS; ++col) {
        if (board[row][col] == EMPTY) {
          return false;  // an empty cell found, not draw, exit
        }
      }
    }
    return true;  // no empty cell, it's a draw
  }

  /** Return true if the player with "theSeed" has won after placing at
     (currentRow, currentCol) */
  public static booleanhasWon(int theSeed, int currentRow, int currentCol) {
    return (board[currentRow][0] == theSeed         // 3-in-the-row
&& board[currentRow][1] == theSeed
&& board[currentRow][2] == theSeed
           || board[0][currentCol] == theSeed      // 3-in-the-column
&& board[1][currentCol] == theSeed
&& board[2][currentCol] == theSeed
           || currentRow == currentCol            // 3-in-the-diagonal
&& board[0][0] == theSeed
&& board[1][1] == theSeed
&& board[2][2] == theSeed
           || currentRow + currentCol == 2  // 3-in-the-opposite-diagonal
&& board[0][2] == theSeed
&& board[1][1] == theSeed
&& board[2][0] == theSeed);
  }

  /** Print the game board */
  public static void printBoard() {
    for (int row = 0; row < ROWS; ++row) {
      for (int col = 0; col < COLS; ++col) {
printCell(board[row][col]); // print each of the cells
        if (col != COLS - 1) {
System.out.print("|");   // print vertical partition
        }
      }
System.out.println();
      if (row != ROWS - 1) {
System.out.println("-----------"); // print horizontal partition
```

```
      }
    }
System.out.println();
  }

  /** Print a cell with the specified "content" */
  public static void printCell(int content) {
    switch (content) {
      case EMPTY:  System.out.print("   "); break;
      case NOUGHT: System.out.print(" O "); break;
      case CROSS:  System.out.print(" X "); break;
    }
  }
}
```

Output:

```
Player 'X', enter your move (row[1-3] column[1-3]): 2
3
   |   |
-----------
   |   | X
-----------
   |   |

Player 'O', enter your move (row[1-3] column[1-3]): 3
1
   |   |
-----------
   |   | X
-----------
 O |   |

Player 'X', enter your move (row[1-3] column[1-3]): 3
3
   |   |
-----------
   |   | X
-----------
 O |   | X

Player 'O', enter your move (row[1-3] column[1-3]): 1
3
   |   | O
-----------
   |   | X
-----------
 O |   | X

Player 'X', enter your move (row[1-3] column[1-3]): 1
```

1
X | | O

| | X

O | | X

Player 'O', enter your move (row[1-3] column[1-3]): 2
2
X | | O

| O | X

O | | X
'O' won! Bye!

4.4 Implement 8-Puzzle Problem Using A* Algorithm

Aim: Implement 8-puzzle problem with heuristic function – A* (informed search)

Theory:

N Puzzle or **sliding puzzle** is a popular puzzle that consists of N tiles where N can be 8, 15, 24 and so on. In our example N = 8. The puzzle is divided into sqrt(N+1) rows and sqrt(N+1) columns. For example, 15-Puzzle will have 4 rows and 4 columns and an 8-Puzzle will have 3 rows and 3 columns. The puzzle consists of N tiles and one empty space where the tiles can be moved. Start and Goal configurations (also called state) of the puzzle are provided. The puzzle can be solved by moving the tiles one by one in the single empty space and thus achieving the Goal configuration.

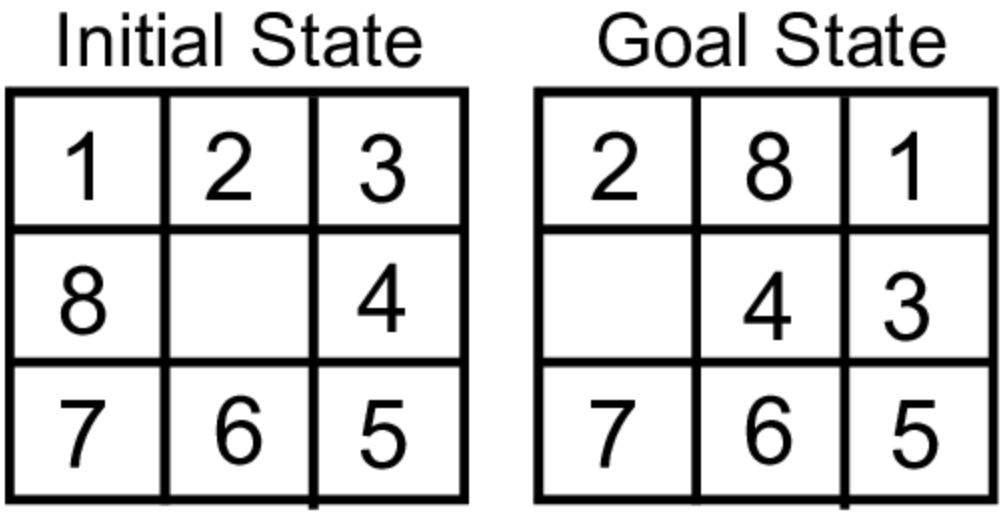


Fig 1. Start and Goal configurations of an 8-Puzzle.

The tiles in the initial(start) state can be moved in the empty space in a particular order and thus achieve the goal state.

Rules for solving the puzzle.

Instead of moving the tiles in the empty space we can visualize moving the empty space in place of the tile, basically swapping the tile with the empty space. The empty space can only move in four directions viz.,

1. Up
2. Down
3. Right or
4. Left

The empty space cannot move diagonally and can take **only one step at a time** (i.e. move the empty space one position at a time).

A* is a computer algorithm that is widely used in pathfinding and graph traversal, the process of plotting an efficiently traversable path between multiple points, called nodes. Noted for its performance and accuracy, it enjoys widespread use.
The key feature of the A* algorithm is that it keeps a track of each visited node which helps in ignoring the nodes that are already visited, saving a huge amount of time. It also has a list that holds all the nodes that are left to be explored and it chooses the most optimal node from this list, thus saving time not exploring unnecessary or less optimal nodes.
So we use two lists namely ‘open list‘ and ‘closed list‘ the open list contains all the nodes that are being generated and are not existing in the closed list and each node explored after its neighboring nodes are discovered is put in the closed list and the neighbors are put in the open list this is how the nodes expand. Each node has a pointer to its parent so that at any given point it can retrace the path to the parent. Initially, the open list holds the start(Initial) node. The next node chosen from the open list is based on its **f score**, the node with the least f score is picked up and explored.

f-score = h-score + g-score

A* uses a combination of heuristic value (h-score: how far the goal node is) as well as the g-score (i.e. the number of nodes traversed from the start node to current node).

In our 8-Puzzle problem, we can define the **h-score** as the number of misplaced tiles by comparing the current state and the goal state or summation of the Manhattan distance between misplaced nodes.
g-score will remain as the number of nodes traversed from start node to get to the current node.

From Fig 1, we can calculate the **h-score** by comparing the initial(current) state and goal state and counting the number of misplaced tiles.
Thus, **h-score** = 5 and **g-score** = 0 as the number of nodes traversed from the start node to the current node is 0.

How A* solves the 8-Puzzle problem.

We first move the empty space in all the possible directions in the start state and calculate **f-score** for each state. This is called expanding the current state.
After expanding the current state, it is pushed into the **closed** list and the newly generated states are pushed into the **open** list. A state with the least f-score is selected and expanded again. This process continues until the goal state occurs as the current state. Basically, here we are providing the algorithm a measure to choose its actions. The algorithm chooses the best possible action and proceeds in that path.
This solves the issue of generating redundant child states, as the algorithm will expand the node with the least **f-score**.

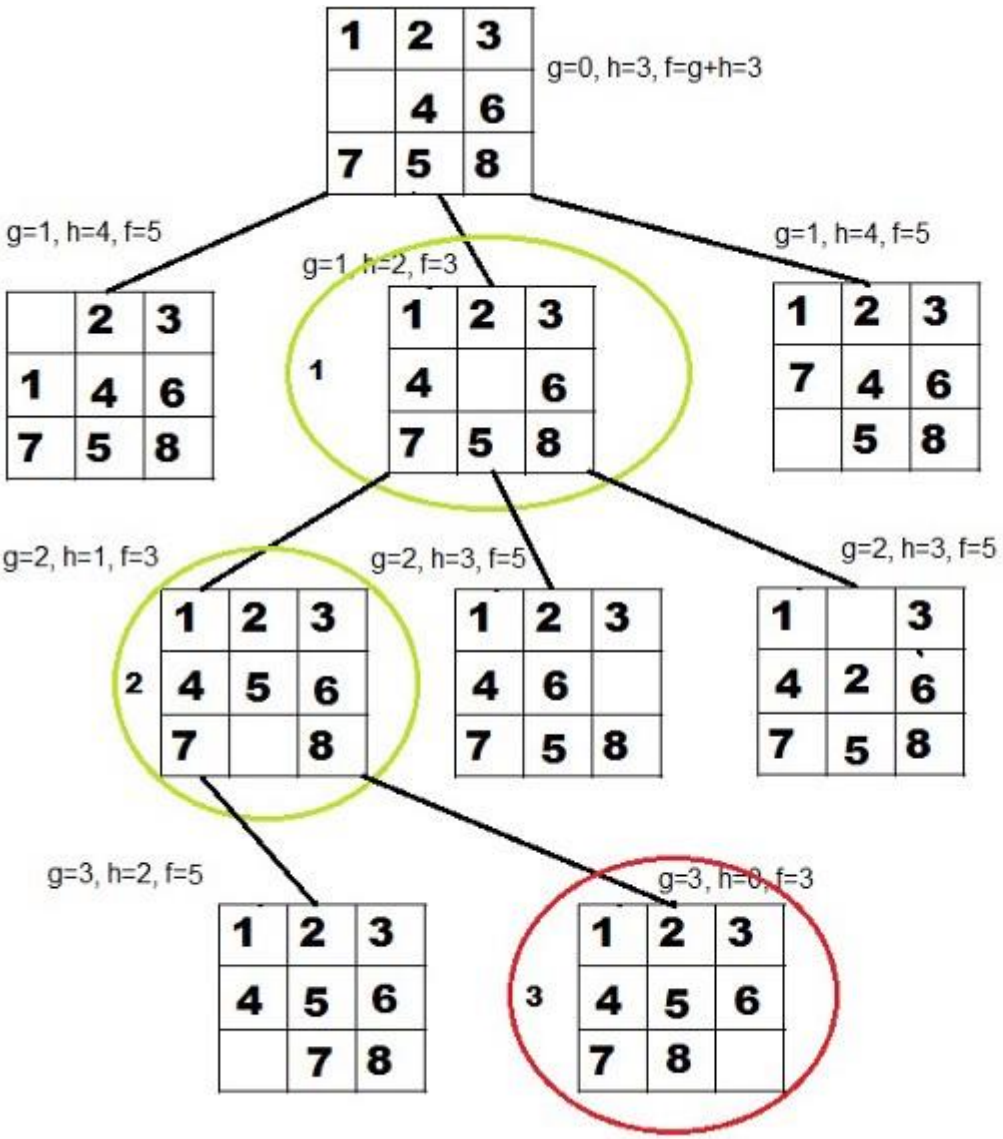


Fig 2. A* algorithm solves 8-puzzle

Program:

```
import
java.util.ArrayList;
                        import java.util.LinkedList;
                        import java.util.Queue;
                        import java.util.Stack;
                        /**
                         * Defines an A* search to be performed on a qualifying puzzle.
                        Currently
                        public class AStarSearch
                        {
                                /**
                                 * Initialization function for 8puzzle A*Search
                                 *
                                 * @param board
```

```
         *          - The starting state, represented as a linear array of length
         *          9 forming 3 meta-rows.
         */
        public static void search(int[] board, char heuristic)
        {
                SearchNode root = new SearchNode(new EightPuzzleState(board));
                Queue<SearchNode> q = new LinkedList<SearchNode>();
                q.add(root);
                int searchCount = 1; // counter for number of iterations
                while (!q.isEmpty()) // while the queue is not empty
                {
                        SearchNode tempNode = q.poll();
                        // if the tempNode is not the goal state
                        if (!tempNode.getCurState().isGoal())
                        {
                                // generate tempNode's immediate successors
                                ArrayList<State> tempSuccessors = tempNode.getCurState()
                                                .genSuccessors();
                                ArrayList<SearchNode> nodeSuccessors = new ArrayList<SearchNode>();
                                /*
                                 * Loop through the successors, wrap them in a SearchNode, check
                                 * if they've already been evaluated, and if not, add them to
                                 * the queue
                                 */
                                for (int i = 0; i < tempSuccessors.size(); i++)
                                {
                                        SearchNode checkedNode;
                                        // make the node
                                        if (heuristic == 'o')
                                        {
                                                /*
                                                 * Create a new SearchNode, with tempNode as the parent,
                                                 * tempNode's cost + the new cost (1) for this state,
                                                 * and the Out of Place h(n) value
                                                 */
                                                checkedNode = new SearchNode(tempNode, tempSuccessors.get(i),
```

```
tempNode.getCost() + tempSuccessors.get(i).findCost(),
((EightPuzzleState) tempSuccessors.get(i)).getOutOfPlace());
                                        }
                                        else
                                        {
                                                // See previous
comment
                                                checkedNode = new
SearchNode(tempNode,

        tempSuccessors.get(i), tempNode.getCost()

        + tempSuccessors.get(i).findCost(),

        ((EightPuzzleState) tempSuccessors.get(i))

        .getManDist());
                                        }
                                        // Check for repeats before
adding the new node
                                        if
(!checkRepeats(checkedNode))
                                        {

        nodeSuccessors.add(checkedNode);
                                        }
                                }
                                // Check to see if nodeSuccessors is
empty. If it is, continue
                                // the loop from the top
                                if (nodeSuccessors.size() == 0)
                                        continue;
                                SearchNode lowestNode =
nodeSuccessors.get(0);
                                /*
                                 * This loop finds the lowest f(n) in a
node, and then sets that
                                 * node as the lowest.
                                 */
                                for (int i = 0; i <
nodeSuccessors.size(); i++)
                                {
                                        if (lowestNode.getFCost() >
nodeSuccessors.get(i)
                                                        .getFCost())
                                        {
                                                lowestNode =
nodeSuccessors.get(i);
                                        }
                                }
```

```
                    int lowestValue = (int) lowestNode.getFCost();
                    // Adds any nodes that have that same lowest value.
                    for (int i = 0; i < nodeSuccessors.size(); i++)
                    {
                        if (nodeSuccessors.get(i).getFCost() == lowestValue)
                        {
    q.add(nodeSuccessors.get(i));
                        }
                    }
                    searchCount++;
                }
                else
                // The goal state has been found. Print the path it took to get to
                // it.
                {
                    // Use a stack to track the path from the starting state to the
                    // goal state
                    Stack<SearchNode> solutionPath = new Stack<SearchNode>();
                    solutionPath.push(tempNode);
                    tempNode = tempNode.getParent();
                    while (tempNode.getParent() != null)
                    {
    solutionPath.push(tempNode);
                        tempNode = tempNode.getParent();
                    }
                    solutionPath.push(tempNode);
                    // The size of the stack before looping through and emptying it.
                    int loopSize = solutionPath.size();
                    for (int i = 0; i < loopSize; i++)
                    {
                        tempNode = solutionPath.pop();
    tempNode.getCurState().printState();
                        System.out.println();
                        System.out.println();
                    }
```

```
                                        System.out.println("The cost was: "
+ tempNode.getCost());
                                        System.exit(0);
                                }
                        }
                        // This should never happen with our current
puzzles.
                        System.out.println("Error! No solution found!");
                }
                /*
                 * Helper method to check to see if a SearchNode has
already been evaluated.
                 * Returns true if it has, false if it hasn't.
                 */
                private static boolean checkRepeats(SearchNode n)
                {
                        boolean retValue = false;
                        SearchNode checkNode = n;
                        // While n's parent isn't null, check to see if it's
equal to the node
                        // we're looking for.
                        while (n.getParent() != null && !retValue)
                        {
                                if
(n.getParent().getCurState().equals(checkNode.getCurState()))
                                {
                                        retValue = true;
                                }
                                n = n.getParent();
                        }
                        return retValue;
                }
}
```

4.5 Implement Travelling Salesman Problem (TSP) Using A* Algorithm

Aim: Implement travelling salesman problem using A* Algorithm (informed search)

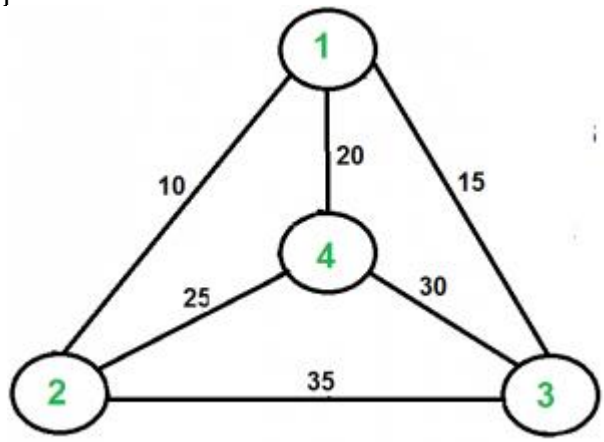
Theory:

Travelling Salesman Problem (TSP): Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns back to the starting point.

Note the difference between Hamiltonian Cycle and TSP. The Hamiltonian cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.

For example, consider the graph shown in figure on right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.

The problem is a famous NP hard problem. There is no polynomial time know solution for this problem.



Examples:

Output of Given Graph:

minimum weight Hamiltonian Cycle :

10 + 25 + 30 + 15 := 80

1. Consider city 1 as the starting and ending point. Since route is cyclic, we can consider any point as starting point.
2. Generate all (n-1)! permutations of cities.
3. Calculate cost of every permutation and keep track of minimum cost permutation.
4. 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 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++]=inputSe

t[j];
}
int
```

```
temp=COST(inputSet[i],setToBePassedOnToNextCa
llOfCOST,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++)//considers each node of
inputSet
{
int k=0;
for(int j=0;j<setSize;j++)
{
if(inputSet[i]!=inputSet[j])
setToBePassedOnToNextCallOfCOST[k++]=inputSe

t[j];
}
int
temp=COST(inputSet[i],setToBePassedOnToNextCa
llOfCOST,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

4.6 Implement 8-Queen Problem with Heuristic Function (Informed Search)
Aim: Implement 8-queen problem with heuristic function (informed search)

Introduction: The problem is to seek out all ways in which of putting N non-attacking queens on an N×N board. A queen attacks all cells in its same row, column, and either diagonal. Therefore, the target is to position N queens on an N×N board in such a way that no 2 queens are on the same row, column or diagonal.

In chess, a queen will move as far as she pleases, horizontally, vertically or diagonally. A chess board has eight rows and eight columns. The standard eight by eight Queens' problem asks the way to place eight queens on an ordinary chess board so none of them will hit the other in one move.

The following constraints need to be satisfied:

To place N queens on an N×N chessboard so that no two queens are attacking one another: i.e.

1.They are not on the same row.

2.They are not on the same column.

3. They are not on the same diagonal.

Algorithm:

Backtracking algorithm: The idea is to place queens one by one in different columns, starting from the left-most column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return false.

1. Start
2. Start in the leftmost column
3. If all queens are placed return true
4. Try all rows in the current column. Do following for every tried row.
(a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
(b) If placing queen in [row, column] leads to a solution then return true.
(c) If placing queen does not lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.
5. If all rows have been tried and nothing worked, return false to trigger backtracking.

6. Stop

Program:

```
import java.io.*;

import java.util.*;
public class Queens
{
//return true if q's placement q[n] does not conflict with other q q[n] through q[n-1]
public static booleanisConsistent(int []q,int n)
{
for(int i=0;i<n;i++)
{
if(q[i]==q[n])
{
return false;
}
if((q[i]-q[n])==(n-i))
{
return false;

return false;

}       }
return true;
}
//print out N-by-N placement of q from permutation q in ASCII
public static void printQueens(int []q)
{
int N=q.length;
System.out.println("Output:");
for(int i=0;i<N;i++)
{
for(int j=0;j<N;j++)
{
if(q[i]==j)
{
}       System.out.print("Q\t");
if((q[n]-q[i])==(n-i))  }
{       else


{

System.out.print("*\t");

}
}
System.out.println();
}
//System.out.println("Output:");
```

```
}
//try all permutation using backtracking
public static void enumerate(int N)
{
int[]a=new int[N];
//System.out.println("Output:");
enumerate(a,0);
}
public static void enumerate(int []q,int n)
{
int N=q.length;
if(n==N)
{
printQueens(q);
}
else
{
for(int i=0;i<N;i++)
{
q[n]=i;
if(isConsistent(q,n))
{

enumerate(q,n+1);

}
}       }       }
public static void main(String args[]) throws
Exception
{
Scanner sc = new Scanner(System.in);
System.out.println("Enter value of N for
N*N matrix:\t");
int N=sc.nextInt();
enumerate(N);
}}
```

Output:

*Enter value of N for N*N matrix:8*

```
Q  *     *  *     *  *    *    *
*  *     *  *     Q  *    *    *
*  *     *  *     *  *    *    Q
```

* * * * * Q * *

* * Q * * * * *

* * * * * * Q *

* Q * * * * * *

* * * Q * * * *

Experiment No. 5

Adversarial Search

5.1 Implement Minimax Algorithm

Aim: Adversarial search minimax algorithm with α-β pruning.
Theory: Minimax may be a decision rule utilised in decision theory, game theory, statistics and philosophy for minimizing the possible loss for a worst case (maximum loss) scenario. as an alternative, it will be thought of as maximizing the minimum gain (maximin). Originally developed for two-player zero-sum game theory, covering both the cases wherever players take alternate moves and people where they create simultaneous moves, it is additionally been extended to additional complicated games and to general decision making within the presence of uncertainty.

In the theory of simultaneous games, a minimax strategy may be a mixed strategy that is part of the answer to a zero-sum game. In zero-sum games, the minimax solution is that the same as the nash equilibrium.

Minimax Theorem: The minimax theorem states the following:
For every two-person, zero-sum game with finitely many strategies, there exists a value V and a mixed strategy for each player, such that
1. Given player 2's strategy, the best payoff possible for player 1 is V, and
2. Given player 1's strategy, the best payoff possible for player 2 is −V.
Equivalently, Player 1's strategy guarantees him a payoff of V regardless of Player 2's strategy, and similarly Player 2 can guarantee himself a payoff of −V. The name minimax arises because each player minimizes the maximum payoff possible for the other—since the game is zero-sum, he also minimizes his own maximum loss (i.e. maximize his minimum payoff).
Example:

	B chooses B1	**B chooses B2**	**B chooses B3**
A chooses A1	+3	−2	+2
A chooses A2	−1	0	+4
A chooses A3	−4	−3	+1

The following example of a game, wherever A and B create simultaneous moves, illustrates minimax solutions. Suppose every player has 3 selections and think about the payoff matrix for A displayed at right. Assume the payoff matrix for B is that the same matrix with the signs reversed (i.e. if the alternatives are A1 and B1 then B pays three to A). Then, the minimax selection for A is A2 since the worst possible result then needs to pay one, whereas the simple minimax selection for B is B2 since the worst attainable result is then no payment. However, this solution is not stable, since if B believes A can opt for A2 then B can opt for B1 to gain 1; then if A believes B can opt for B1 then A can opt for A1 to achieve 3; then B can opt for B2; and eventually each players can realize the difficulty of making a selection. Thus, an additional stable strategy is required.

Some decisions are dominated by others and may be eliminated: A won't select A3 since either A1 or A2 can turn out a stronger result, no matter what B selects; B won't choose B3 since some mixtures of B1 and B2 can turn out a stronger result, no matter what A chooses.

A can avoid having to make an expected payment of more than 1/3 by selecting A1 with probability 1/6 and A2 with probability 5/6, despite what B chooses. B will ensure an expected gain of a minimum of 1/3 by using an irregular strategy of choosing B1 with probability 1/3 and B2 with probability 2/3, no matter what A chooses. These mixed minimax strategies are currently stable and can't be improved.

Minimax-algorithm

```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

Properties of minimax

• Complete? Yes (if tree is finite)
• Optimal? Yes (against an optimal opponent)
• Time complexity? O (bm)
• Space complexity? O (bm) (depth- first exploration)

The α-β algorithm

```
function ALPHA-BETA-SEARCH(state) returns an action
    inputs: state, current state in game
    v ← MAX-VALUE(state, −∞, +∞)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
    inputs: state, current state in game
            α, the value of the best alternative for MAX along the path to state
            β, the value of the best alternative for MIN along the path to state
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s, α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
    inputs: state, current state in game
            α, the value of the best alternative for MAX along the path to state
            β, the value of the best alternative for MIN along the path to state
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← +∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s, α, β))
        if v ≤ α then return v
        β ← MIN(β, v)
    return v
```

Properties of α-β

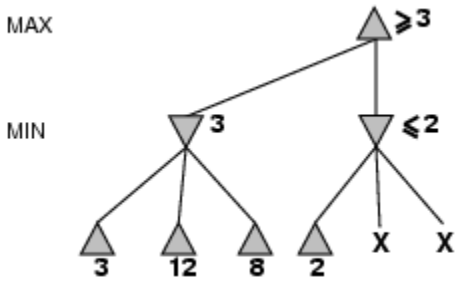
• Pruning does not affect final result

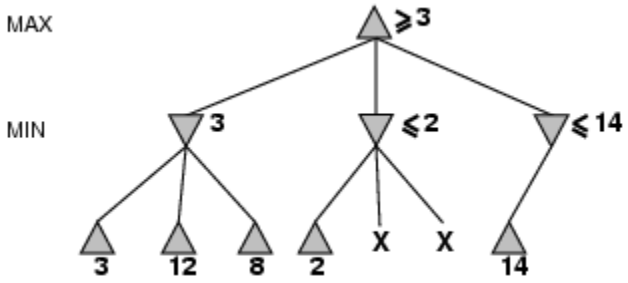
• Good move ordering improves effectiveness of pruning

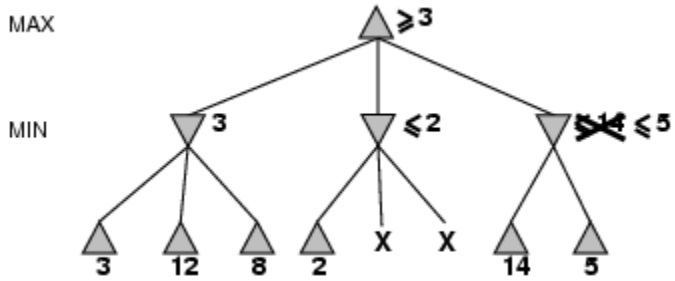
• With "perfect ordering," time complexity = O (bm/2)

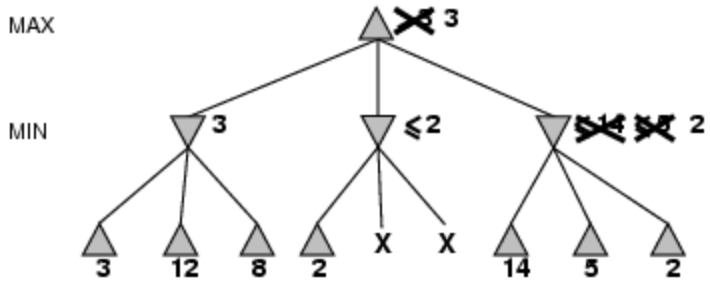
 doubles depth of search

• A simple example of the value of reasoning about which Computations are relevant (a form of meta-reasoning)









Conclusion: Thus, we have enforced a program using alpha-beta pruning in Min-Max algorithm. Thus, Min-Max algorithm prunes the information, thereby using alpha-beta pruning, we reduce the iteration steps.

Program:

```
import java.util.Scanner;
public class minmax {
```

```
   public static int min(int a[][],int n,intsetIndex) {
      int smallest = a[setIndex][0];
      for(int i=1; i<n; i++) {
         if(smallest > a[setIndex][i])
            smallest = a[setIndex][i];
      }
      return smallest;
   }

   public static int max(int a[][],int n,intsetIndex) {
      int greatest = a[setIndex][0];
      for(int i=1; i<n; i++) {
         if(greatest < a[setIndex][i])
            greatest = a[setIndex][i];
      }
      return greatest;
   }

   public static void main(String[] args) {
     Scanner s = new Scanner(System.in);
System.out.println("Enter the no. of nodes in each subtree");
      int n = s.nextInt();
      int set[][] = new int[n][n];
System.out.println("Enter the utility values: ");
      for(int i=0; i<n; i++) {
         for(int j=0; j<n; j++) {
            set[i][j] = s.nextInt();
         }
      }
      int max[][] = new int[1][n];
System.out.print("The min values retured are: ");
      for(int i=0; i<n; i++) {
         max[0][i] = min(set, n, i);
System.out.print(" " +max[0][i]);
      }
System.out.println("");
      int maxValue = max(max, n, 0);
System.out.println("The Max Value is " + maxValue);
   }
}
OUTPUT:
Enter the no. of nodes in each subtree
3
Enter the utility values:
-2
2
1
1
3
```

4
4
3
7
The min values retured are: -2 1 3
The Max Value is 3

Process completed.

Experiment No. 6

Constraint Satisfaction Problem

6.1 Implement 8-Queen Problem

Aim: To solve 8-queen problem using constraint satisfaction.

Theory:

Introduction: The problem is to search out all ways that of placing N non-attacking queens on an N by N board. A queen attacks all cells in its same row, column, and either diagonal. Therefore, the target is to put N queens on an n by n board in such the simplest way that no 2 queens are on identical row, column or diagonal.

In chess, a queen will move as so much as she pleases, horizontally, vertically or diagonally. A chess board has eight rows and eight columns. The quality eight by eight Queens' drawback asks the way to place eight queens on a standard chess board so none of them will hit the other in one move.

The following constraints need to be satisfied:

To place N queens on an N×N chessboard so that no two queens are attacking one another: i.e.

1.They are not on the same row.

2.They are not on the same column.

3. They are not on the same diagonal.

Algorithm:

1. **Backtracking Algorithm:** The idea is to place queens one by one in different columns, starting from the left-most column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return false.
   1.Start
   2.Start in the leftmost column
   3.If all queens are placed, return true
   4.Try all rows in the current column. Do following for every tried row.
   (a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
   (b) If placing queen in [row, column] leads to a solution then return true.
   (c) If placing queen does not lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.
   5.If all rows have been tried and nothing worked, return false to trigger backtracking.
   6.Stop

Conclusion: The constraint satisfaction problem (CSP) consists in finding a solution for a constraint network. This has numerous applications including, e.g. scheduling and timetabling.

Program:

```
import java.util.*;
public class EightQueens {
  public static void main(String args[]) {
    Scanner src=new Scanner (System.in);
System.out.println ("Enter the number of queens you want to place :");
int  N=src.nextInt();
    int[][] board = new int[N][N];
    solve(0, board, N);
for (int i = 0; i< N; i++) {
for (int j = 0; j < N; j++) {
if (board[i][j]==1)
System.out.print ("Q  ");
else System.out.print("* ");
      }
System.out.println();
    }  }
  static boolean solve(int row, int[][] board, int N) {
    if(row>=N) return true;
    for(int position = 0; position < N; position++) {
      if(isValid(board, row, position, N)) {
        board[row][position] = 1;
        if(!solve(row+1, board, N)) {
          board[row][position] = 0;
        } else
          return true;
      }
    }
    return false;
```

```
  }

  static booleanisValid(int[][] board, int x, int y, int N) {
    int i, j;
    for(i = 0; i< x; i++)
      if(board[i][y]==1)
        return false;
i = x - 1;
    j = y - 1;
    while((i>=0)&&(j>=0))
      if(board[i--][j--]==1) return false;
i = x - 1;
    j = y + 1;
    while((i>=0)&&(j<N))
      if(board[i--][j++]==1) return false;
    return true;
  }
}
```

Output: Enter the number of queens you want to place:8.

```
Q * * * * * * *
* * * * Q * * *
* * * * * * * Q
* * * * * Q * *
* * Q * * * * *
* * * * * * Q *
* Q * * * * * *
* * * Q * * * *
```

6.2 Implement Map Colouring Problem

Aim: To implement map colouring problem.

Theory: In AI and research, constraint satisfaction is the method of finding an answer to a collection of constraints that impose conditions that the variables should satisfy. An answer is so a collection of values for the variables that satisfies all constraints—that is, a point within the feasible region.

Program:

```
importjava.io.FileNotFound
Exception;

//This class implements the map colouring problem.

publicclassMapColoringProblem {

    Graph;

    int[] color_config_array;//this array holds the color number for each vertex corresponding to the index

    //For example - if the number of colors is 3 . Then different colours will be represented by numbers 1,2 and 3.

    //this method prints which vertex will be of which color

    publicvoidprintConfiguration()

    {

    for(inti=0;i<color_config_array.length;i++)

        {

            System.out.println("The "+(i+1)+"(th) vertex will be colored in color number "+color_config_array[i]);

        }

    }
```

```
//this method colors the graph using recursive backtracking in all possible combinations and returns true of the graph can

//be colored in the given number of colors

publicbooleancolorGraph(intvertex_num,intnumber_of_colours)

{


if(vertex_num==graph.adjLists.length)//base condition

        {

                returncheckGraph();

        }

        for(inti=1;i<=number_of_colours;i++)

        {

                color_config_array[vertex_num]=i;


if(colorGraph(vertex_num+1,number_of_colours)==true)

                {

                        returntrue;

                }

        }

        returnfalse;

}


//this method  returns true of graph is colored properly and false otherwise

publicbooleancheckGraph()

{

        boolean result=true;

        intno_Of_vertex=graph.adjLists.length;

        for(inti=0;i<no_Of_vertex;i++)

        {
```

```
        intcolor_of_vertex=color_config_array[i];
                        Neighbour
neighbour=graph.adjLists[i].adjList;
                        while(neighbour!=null)
                        {

        if(color_of_vertex==color_config_array[neighbou
r.vertexNum])
                                {
                                        result=false;
                                        return result;
                                }
                                neighbour=neighbour.next;
                        }


                }
                return result;
        }


        /**
         * @paramargs
         * @throwsFileNotFoundException
         */
        publicstaticvoidmain(String[]                   args)
throwsFileNotFoundException {
                // TODO Auto-generated method stub
                String
path="C:\\Users\\sangam\\Desktop\\graph.txt";

        MapColoringProblemmapColoringProblem=new
MapColoringProblem();
                mapColoringProblem.graph=new
Graph(path);
```

```
        intno_Of_vertex=mapColoringProblem.graph.adjLists.length;

        mapColoringProblem.color_config_array=new int[no_Of_vertex];

                intnumber_of_colours=3;//this can be changed as per the user input

                boolean result=mapColoringProblem.colorGraph(0, number_of_colours);//0 is because we have to start the coloring from zeroth vertex

                if(result ==true)

                {

                        System.out.println("The combination is ");

        mapColoringProblem.printConfiguration();

                }

                else

                {

                        System.out.println("the graph cannot be colored in these many given colours");

                }



        }

}
```

6.3 Implement Crypt Arithmetic Problem

Aim: To implement crypt arithmetic problem.

Theory: Cryptarithmetic problems are where numbers are replaced with alphabets. By using standard arithmetic rules, we need to decipher the alphabet.

General rules:
1. Each alphabet takes only one number from 0 to 9 uniquely.
2. Two single digit numbers sum can be maximum 19 with carryover. So, carry over in problems of two number addition is always 1.
3. Try to solve left most digit in the given problem.
4. If a × b = kb, then the following are the possibilities (3 × 5 = 15; 7 × 5 = 35; 9 × 5 = 45) or (2 × 6 = 12; 4 × 6 = 24; 8 × 6 = 48)

Program:

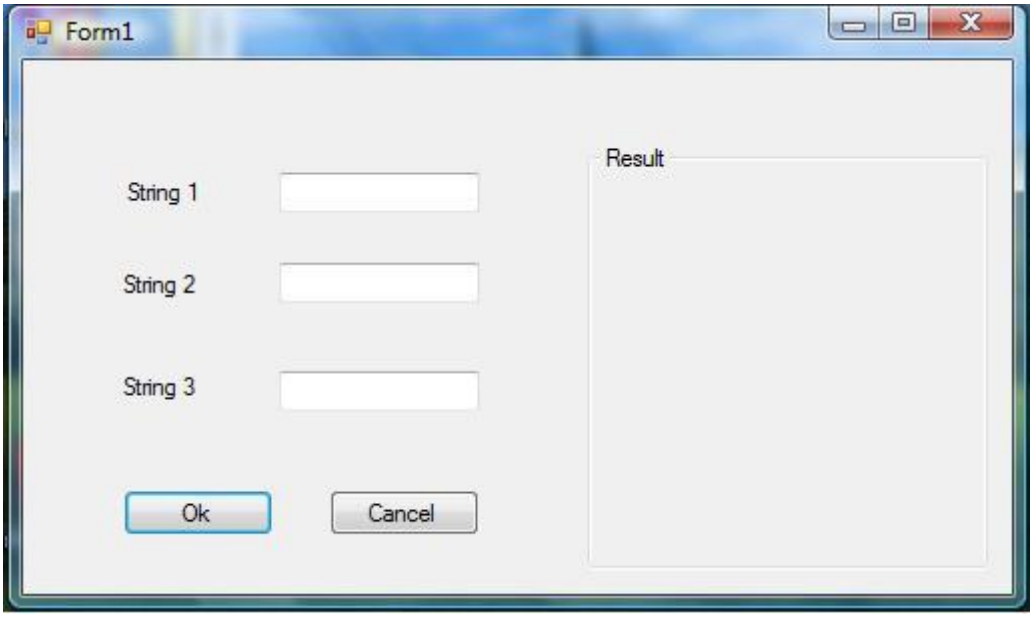
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace CryptArithmetic
{
publicpartialclass Form1 : Form
   {
char[] s1 = newchar[10];
char[] s2 = newchar[10];
char[] s3 = newchar[10];
int[] assinged = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
char[] c = newchar[11];

int[] val = newint[11];
int topc = 0;
public Form1()
      {
InitializeComponent();
      }

privatevoidbtn_ok_Click(object sender, EventArgs e)
      {
         label4.Text = "";

         s1 = textBox1.Text.ToCharArray();
```

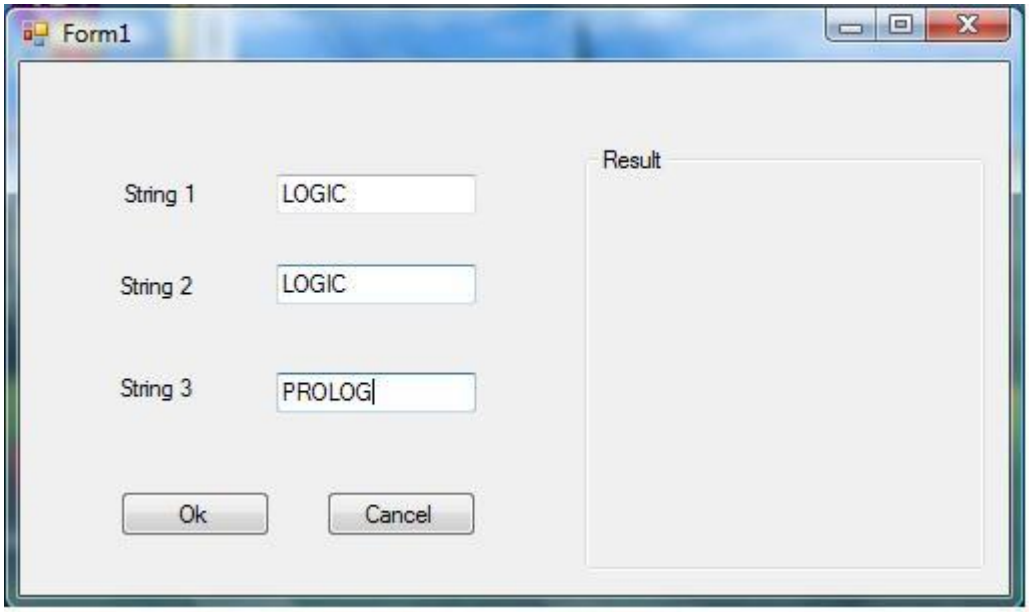
```
        s2 = textBox2.Text.ToCharArray();
        s3 = textBox3.Text.ToCharArray();

        int flag=0;

        for(int i=0;i<s1.Length;i++)
            {
        for(int j=0;j<=topc;j++)
                    {
        if(s1[i]!=c[j])
                        flag=1;
        else
                {
                        flag =0;
            break;
                    }
                }
        if(flag==1)
                    c[topc++] =s1[i];
            }

        for(int i=0;i<s2.Length;i++)
            {
for(int j=0;j<=topc;j++)
                {
        if(s2[i]!=c[j])
                        flag=1;
        else
                {
                        flag =0;
                break;
                    }
                }
if(flag==1)
                 c[topc++] =s2[i];
            }

for(int i=0;i<s3.Length;i++)
                {
for(int j=0;j<=topc;j++)
              {
if(s3[i]!=c[j])
        flag=1;
else
                {
                   flag =0;
break;
                }
             }
                if(flag==1)
```

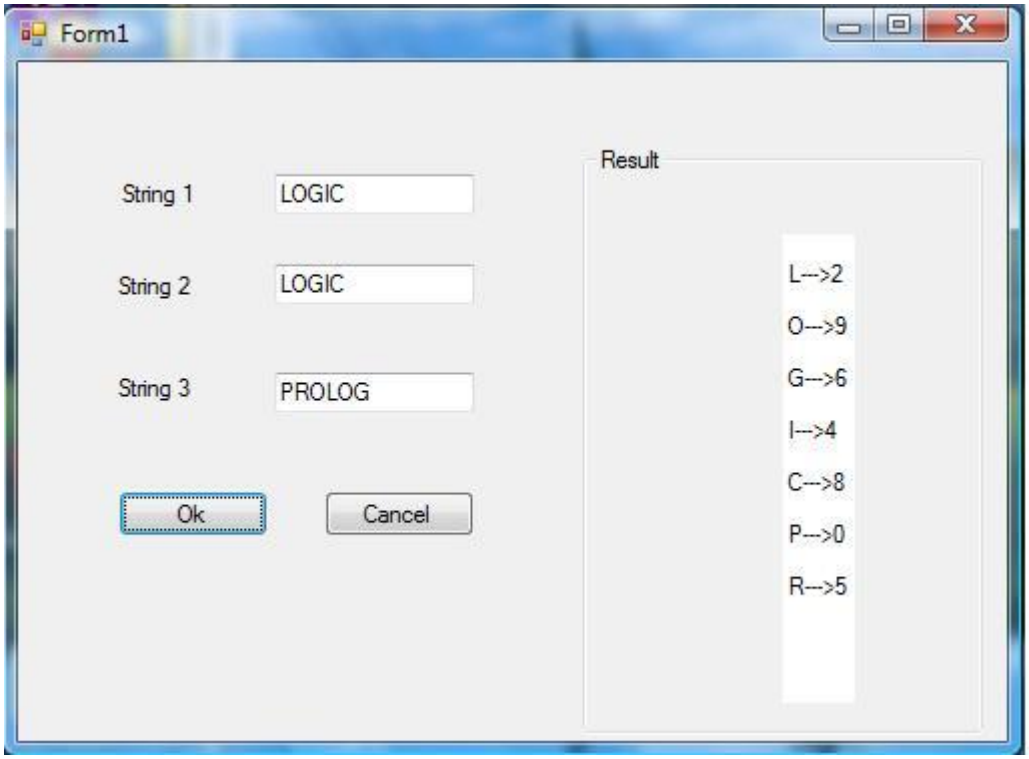
```
                                    c[topc++] =s3[i];
                }

if (solve(0, assinged)==1)
          {
for(int i=0;i<c.Length;i++)
              label4.Text += "\n"+ c[i]+"--->"+val[i].ToString() + "\n";
          }
else
              label4.Text = "Sorry";
       }

//-------------------end of getdata-----------------

int solve(int ind,int []temp1)
       {
          int [] temp2 = newint[10];
          int flag=0;
          for(int i=0;i<10;i++)
                {
                   if(temp1[i]==0)
                        {
                   for(int j=0;j<10;j++)
                                    temp2[j]=temp1[j];
                      temp2[i]=1;
          val[ind]=i;
          if(ind==(topc-1))
                      {
                   if(verify()==1)
                                    {
                                       flag=1;
                            goto exit;
                                    }
                      }
          else
                      {
if(solve(ind+1,temp2)==1)
                   {
                                flag=1;
                   goto exit;
                   }
                }
                   }
          }
exit :
if(flag!=0)
          return1;
else
          return0;
       }
```

```
            int verify()
{
   long n1=0,n2=0,n3=0;
   long power=1;
   char ch;
   int i=s1.Length-1;
   int in1;
   while(i>=0)
                {
            ch=s1[i];
                  in1=0;
            while(in1!=topc)
                  {
            if(c[in1]==ch)
                                    break;
            else
                              in1++;
                  }
                        n1+=power*val[in1];
                        power *=10;
                     i--;
                }
          power=1;
   i=s2.Length-1;
   while(i>=0)
                {
                     ch=s2[i];
                        in1=0;
                     while(in1!=topc)
                        {
                     if(c[in1]==ch)
                                    break;
                     else
                                 in1++;
                        }
                              n2+=power*val[in1];
                              power *=10;
                              i--;
                }
                power=1;
            i=s3.Length-1;
            while(i>=0)
                     {
                              ch=s3[i];
                              in1=0;
                              while(in1!=topc)
                              {
                                    if(c[in1]==ch)
                                          break;
```

```
                    else
                        in1++;
                }
                n3+=power*val[in1];
                power *=10;
                i--;
            }
            if(n1+n2==n3)
                return1;
            else
                return0;
        }

privatevoid Form1_Load(object sender, EventArgs e)
    {

    }

privatevoid textBox1_TextChanged(object sender, EventArgs e)
    {
    }
  }
}
```




Form1
String 1
LOGIC
String 2
LOGIC
String 3
PROLOG
Ok
Cancel
Result


Form1
String 1
LOGIC
String 2
LOGIC
String 3
PROLOG
Ok
Cancel
Result
L--->2
O--->9
G--->6
I--->4
C--->8
P--->0
R--->5

Experiment No. 7

Design of a Planning System Using STRIPS (Block World Problem)

Aim: Study of planning agent.

Theory:

Language of Planning Problem:

What is STRIPS?

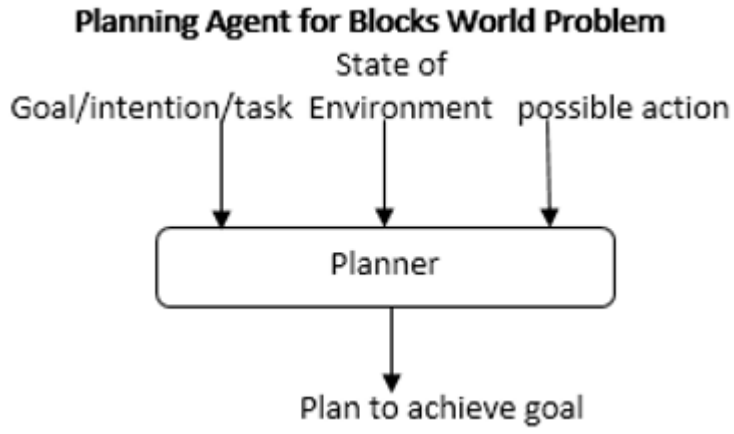
The Stanford Research Institute Problem Solver (**STRIPS**) is an automated planning technique that works by executing a domain and problem to find a goal. With STRIPS, you first describe the world. You do this by providing objects, actions, preconditions, and effects. These are all the types of things 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

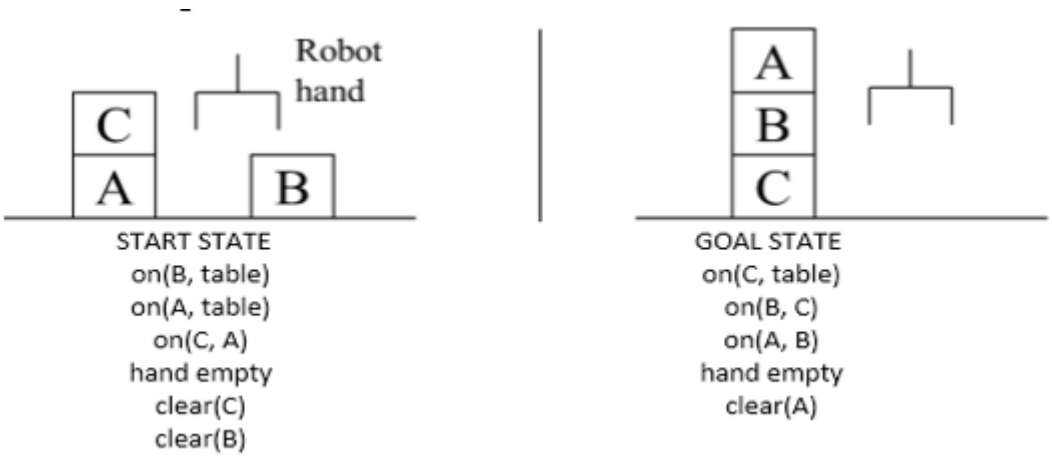
Idea is to give an agent:

- 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)***

Experiment No. 8

Implementation of Bayes' belief network (probabilistic reasoning in an uncertain domain)

Aim: To implementation of Bayes' belief network (probabilistic reasoning in an uncertain domain).

Theory:

Probabilistic reasoning

The aim of a reasoning is to combine the capacity of probability theory to handle uncertainty with the capacity of deductive logic to exploit structure. The result is a richer and more expressive formalism with a broad range of possible application areas. Probabilistic logics attempt to find a natural extension of traditional logic truth tables: the results they define are derived through probabilistic expressions instead. A difficulty with probabilistic logics is that they tend to multiply the computational complexities of their probabilistic and logical components. Other difficulties include the possibility of counter-intuitive results, such as those of Dempster-Shafer theory. The need to deal with a broad variety of contexts and issues has led to many different proposals.

Probabilistic Reasoning Using Bayesian Learning: The idea of Bayesian learning is to compute the posterior probability distribution of the target features of a new example conditioned on its input features and all of the training examples.

Suppose a new case has inputs $X=x$ and has target features, Y; the aim is to compute $P(Y|X=x \wedge e)$, where e is the set of training examples. This is the probability distribution of the target variables given the particular inputs and the examples. The role of a model is to be the assumed generator of the examples. If we let M be a set of disjoint and covering models, then reasoning by cases and the chain rule give

$$\begin{aligned} P(Y|x \wedge e) &= \sum_{m \in M} P(Y \wedge m \,|x \wedge e) \\ &= \sum_{m \in M} P(Y \mid m \wedge x \wedge e) \times P(m|x \wedge e) \\ &= \sum_{m \in M} P(Y \mid m \wedge x) \times P(m|e) \,. \end{aligned}$$

The first two equalities are theorems from the definition of probability. The last equality makes two assumptions: the model includes all of the information about the examples that is necessary for a particular prediction [i.e., $P(Y \mid m \wedge x \wedge e) = P(Y \mid m \wedge x)$], and the model does not change depending on the inputs of the new example [i.e., $P(m|x \wedge e) = P(m|e)$]. This formula says that we average over the prediction of all of the models, where each model is weighted by its posterior probability given the examples.

$P(m|e)$ can be computed using Bayes' rule:

$$P(m|e) = (P(e|m) \times P(m))/(P(e)) \,.$$

Thus, the weight of each model depends on how well it predicts the data (the likelihood) and its prior probability. The denominator, $P(e)$, is a normalizing constant to make sure the

posterior probabilities of the models sum to 1. Computing $P(e)$ can be very difficult when there are many models.

A set $\{e_1,...,e_k\}$ of examples are IID (independent and identically distributed), where the distribution is given by model m if, for all i and j, examples e_i and e_j are independent given m, which means $P(e_i \wedge e_j|m)=P(e_i|m)\times P(e_j|m)$. We usually assume that the examples are i.i.d.

Suppose the set of training examples e is $\{e_1,...,e_k\}$. That is, e is the conjunction of the e_i, because all of the examples have been observed to be true. The assumption that the examples are IID implies

$$P(e|m) = \prod_{i=1}^{k} P(e_i|m)$$

The set of models may include structurally different models in addition to models that differ in the values of the parameters. One of the techniques of Bayesian learning is to make the parameters of the model explicit and to determine the distribution over the parameters.

Example: Consider the simplest learning task under uncertainty. Suppose there is a single Boolean random variable, Y. One of two outcomes, a and $\neg a$, occurs for each example. We want to learn the probability distribution of Y given some examples.

There is a single parameter, φ, that determines the set of all models. Suppose that φ represents the probability of $Y=true$. We treat this parameter as a real-valued random variable on the interval $[0,1]$. Thus, by definition of φ, $P(a|\varphi)=\varphi$ and $P(\neg a|\varphi)=1-\varphi$.

Suppose an agent has no prior information about the probability of Boolean variable Y and no knowledge beyond the training examples. This ignorance can be modelled by having the prior probability distribution of the variable φ as a uniform distribution over the interval $[0,1]$. This is the probability density function labeled$n_0=0$, $n_1=0$ in.

We can update the probability distribution of φ given some examples. Assume that the examples, obtained by running a number of independent experiments, are a particular sequence of outcomes that consists of n_0 cases where Y is false and n_1 cases where Y is true.

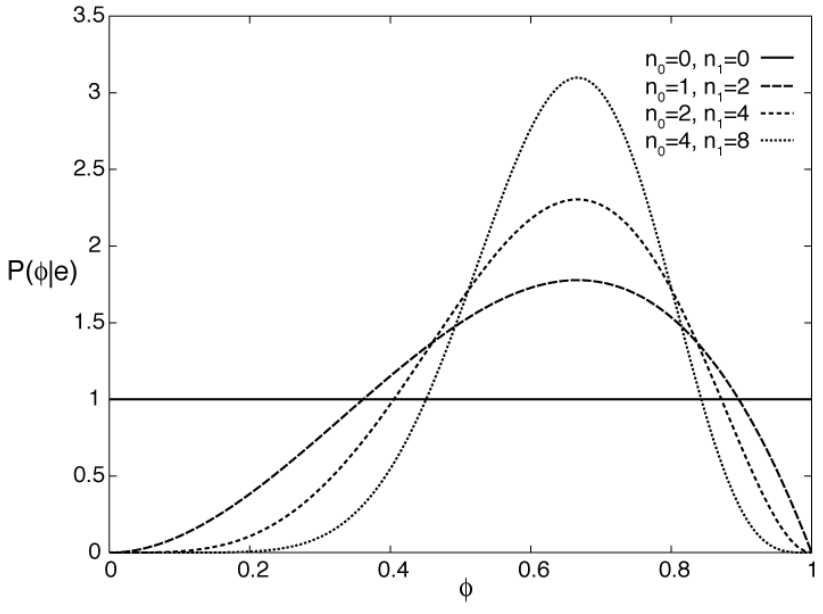


Figure Beta distribution based on different samples.

The posterior distribution for φ given the training examples can be derived by Bayes' rule. Let the examples $\boldsymbol{e}$ be the particular sequence of observation that resulted in n_1 occurrences of $Y=true$ and n_0 occurrences of $Y=false$. Bayes' rule gives us

$P(\varphi|\boldsymbol{e})=(P(\boldsymbol{e}|\varphi)\times P(\varphi))/(P(\boldsymbol{e}))$.

The denominator is a normalizing constant to make sure the area under the curve is 1.

Given that the examples are IID,

$P(e|\varphi)=\varphi^{n}{}_1\times(1-\varphi)^{n}{}_0$

Because there are n_0 cases where $Y=false$, each with a probability of $1-\varphi$, and n_1 cases where $Y=true$, each with a probability of φ.

One possible prior probability, $P(\varphi)$, is a uniform distribution on the interval $[0,1]$. This would be reasonable when the agent has no prior information about the probability.

The figure on “Beta distribution based on different samples” gives some posterior distributions of the variable φ based on different sample sizes, and given a uniform prior. The cases are $(n_0=1, n_1=2)$, $(n_0=2, n_1=4)$, and $(n_0=4, n_1=8)$. Each of these peak at the same place, namely at $(2)/(3)$. More training examples make the curve sharper.

The distribution of this example is known as beta distribution; it is parametrized by two counts, α_0 and α_1, and a probability p. Traditionally, the α_i parameters for the beta distribution are one more than the counts; thus, $\alpha_i=n_i+1$. The beta distribution is

$Beta^{\alpha_0,\alpha_1}(p)=(1)/(K)\ p^{\alpha_1-1}\times(1-p)^{\alpha_0-1}$

where K is a normalizing constant that ensures the integral over all values is 1. Thus, the uniform distribution on $[0,1]$ is the beta distribution $Beta^{1,1}$.

The generalization of the beta distribution to more than two parameters is known as the Dirichlet distribution. The Dirichlet distribution with two sorts of parameters, the "counts" $\alpha_1,\ldots,\alpha_k$, and the probability parameters $p_1,\ldots,p_k$, is

$Dirichlet^{\alpha_1,\ldots,\alpha_k}(p_1,\ldots,p_k) = (1)/(K)\ \prod_{j=1}^{k} p_j^{\alpha_j-1}$

where K is a normalizing constant that ensures the integral over all values is 1; p_i is the probability of the ith outcome (and so $0 \le p_i \le 1$) and α_i is one more than the count of the ith outcome. That is, $\alpha_i=n_i+1$. The Dirichlet distribution looks like as in the figure along each dimension (i.e. as each p_j varies between 0 and 1).

For many cases, summing over all models weighted by their posterior distribution is difficult, because the models may be complicated (e.g., if they are decision trees or even belief networks). However, for the Dirichlet distribution, the expected value for outcome i (averaging over all p_j's) is

$(\alpha_i)/(\sum_j \alpha_j)$.

The reason that the α_i parameters are one more than the counts is to make this formula simple. This fraction is well defined only when the α_j are all non-negative and not all are zero.

Example: Consider Example, which determines the value of φ based on a sequence of observations made up of n_0 cases where Y is false and n_1 cases where Y is true. Consider the posterior. What is interesting about this is that, whereas the most likely posterior value of φ is $(n_1)/(n_0+n_1)$, the expected value of this distribution is $(n_1+1)/(n_0+n_1+2)$.

Thus, the expected value of the $n_0=1$, $n_1=2$ curve is $(3)/(5)$, for the $n_0=2$, $n_1=4$ case the expected value is $(5)/(8)$, and for the $n_0=4$, $n_1=8$ case it is $(9)/(14)$. As the learner gets more training examples, this value approaches $(n)/(m)$.

This estimate is better than $(n)/(m)$ for a number of reasons. First, it tells us what to do if the learning agent has no examples: Use the uniform prior of $(1)/(2)$. This is the expected value of the $n=0$, $m=0$ case. Second, consider the case where $n=0$ and $m=3$. The agent should not use $P(y)=0$, because this says that Y is impossible, and it certainly does not have evidence for this! The expected value of this curve with a uniform prior is $(1)/(5)$.

An agent does not have to start with a uniform prior; it can start with any prior distribution. If the agent starts with a prior that is a Dirichlet distribution, its posterior will be a Dirichlet distribution. The posterior distribution can be obtained by adding the observed counts to the α_i parameters of the prior distribution.

The IID assumption can be represented as a belief network, where each of the e_i are independent given model m. This independence assumption can be represented by the belief network.

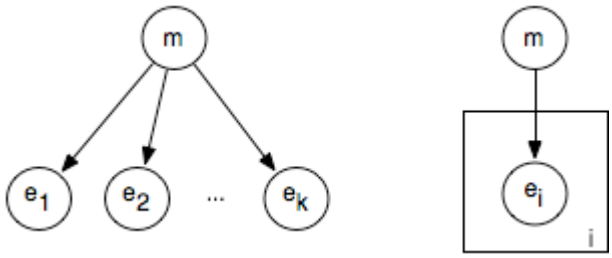


Figure Belief network and plate models of Bayesian learning.

If m is made into a discrete variable, any of the inference methods of the previous chapter can be used for inference in this network. A standard reasoning technique in such a network is to condition on all of the observed e_i and to query the model variable or an unobserved e_i variable.

The problem with specifying a belief network for a learning problem is that the model grows with the number of observations. Such a network can be specified before any observations have been received by using a plate model. A plate model specifies what variables will be used in the model and what will be repeated in the observations. The plate is drawn as a rectangle that contains some nodes, and an index (drawn on the bottom right of the plate).

The nodes in the plate are indexed by the index. In the plate model, there are multiple copies of the variables in the plate, one for each value of the index. The intuition is that there is a pile of plates, one for each value of the index. The number of plates can be varied depending on the number of observations and what is queried. In this figure, all of the nodes in the plate share a common parent. The probability of each copy of a variable in a plate given the parents is the same for each index.

A plate model lets us specify more complex relationships between the variables. In a hierarchical Bayesian model, the parameters of the model can depend on other parameters. Such a model is hierarchical in the sense that some parameters can depend on other parameters.

Example: Suppose a diagnostic assistant agent wants to model the probability that a particular patient in a hospital is sick with the flu before symptoms have been observed for this patient. This prior information about the patient can be combined with the observed symptoms of the patient. The agent wants to learn this probability, based on the statistics about other patients in the same hospital and about patients at different hospitals. This problem can range from the cases where a lot of data exists about the current hospital (in which case, presumably, that data should be used) to the case where there is no data about the particular hospital that the patient is in. A hierarchical Bayesian model can be used to combine the statistics about the particular hospital the patient is in with the statistics about the other hospitals.

Suppose that for patient X in hospital H there is a random variable S_{HX} that is true when the patient is sick with the flu. (Assume that the patient identification number and the hospital uniquely determine the patient.) There is a value φ_H for each hospital H that will be used for the prior probability of being sick with the flu for each patient in H. In a Bayesian model, φ_H is treated as a real-valued random variable with domain $[0,1]$. S_{HX} depends on φ_H, with $P(S_{HX}|\varphi_H)=\varphi_H$. Assume that φ_H is distributed according to a beta distribution. We don't assume that φ_{hi} and φ_{h2} are independent of each other, but depend on hyperparameters. The hyperparameters can be the prior counts α_0 and α_1. The parameters depend on the hyperparameters in terms of the conditional probability $P(\varphi_{hi}|\alpha_0,\alpha_1)= Beta^{\alpha_0}{}_{,}{}^{\alpha_1}(\varphi_{hi})$; α_0 and α_1 are real-valued random variables, which require some prior distribution.

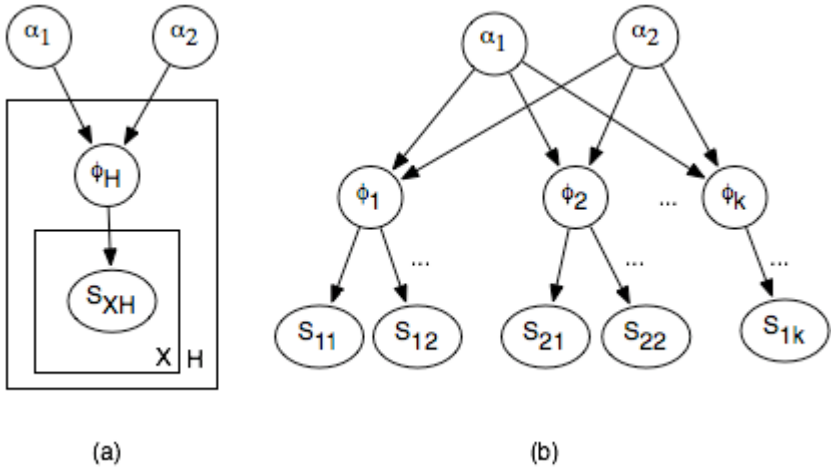


Figure Hierarchical Bayesian model.

Part (a) shows the plate model, where there is a copy of the outside plate for each hospital and a copy of the inside plate for each patient in the hospital. Part of the resulting belief network is shown in part (b). Observing some of the S_{HX} will affect the φ_H and so α_0 and α_1, which will in turn affect the other φ_H variables and the unobserved S_{HX} variables.

Sophisticated methods exist to evaluate such networks. However, if the variables are made discrete, any of the methods of the previous chapter can be used.

In addition to using the posterior distribution of φ to derive the expected value, we can use it to answer other questions such as: What is the probability that the posterior probability of φ is in the range $[a,b]$? In other words, derive $P((\varphi \geq a \wedge \varphi \leq b) \mid e)$.

$$(\int_a^b p^n \times (1-p)^{m-n}) / (\int_0^1 p^n \times (1-p)^{m-n})$$

This kind of knowledge is used in surveys when it may be reported that a survey is correct with an error of at most 5%, 19 times out of 20. It is also the same type of information that is used by probably approximately correct (PAC) learning, which guarantees an error at most ε at least $1-\delta$ of the time. If an agent chooses the midpoint of the range $[a,b]$, namely $(a+b)/(2)$, as its hypothesis, it will have error less than or equal to $(b-a)/(2)$, just when the hypothesis is in $[a,b]$. The value $1-\delta$ corresponds to $P(\varphi \geq a \wedge \varphi \leq b \mid e)$. If $\varepsilon=(b-a)/(2)$ and $\delta=1-P(\varphi \geq a \wedge \varphi \leq b \mid e)$, choosing the midpoint will result in an error at most ε in $1-\delta$ of the time. PAC learning gives worst-case results, whereas Bayesian learning gives the expected number. Typically, the Bayesian estimate is more accurate, but the PAC results give a guarantee of the error. The sample complexity required for Bayesian learning is typically much less than that of PAC learning – many fewer examples are required to *expect to achieve* the desired accuracy than are needed to *guarantee* the desired accuracy.

Experiment No. 9

Implement Resolution Inference Rule Using Prolog

Aim: To study resolution inference rule.

Theory: Inference is the act or method of deriving logical conclusions from premises known or assumed to be true. The conclusion drawn is additionally known as an idiomatic. The laws of valid inference are studied within the field of logic.

Human inference (i.e. how humans draw conclusions) is historically studied inside the sphere of cognitive psychology; artificial intelligence researchers develop machine-driven inference systems to emulate human inference. statistical inference permits for inference from quantitative data.

The process by which a conclusion is inferred from multiple observations is named inductive reasoning. The conclusion is also correct or incorrect, or correct to within a certain degree of accuracy, or correct in certain situations. Conclusions inferred from multiple observations is also tested by additional observations.

A conclusion reached on the basis of proof and reasoning.

The process of reaching such a conclusion: "order, health, and by inference cleanliness".

The validity of an inference depends on the shape of the inference. That is, the word "valid" does not refer to the reality of the premises or the conclusion, but rather to the form of the inference. an inference may be valid though the elements are false, and may be invalid though the elements are true. However, a valid form with true premises can always have a real conclusion.

For example,
All fruits are sweet.
A banana is a fruit.
Therefore, a banana is sweet.
For the conclusion to be necessarily true, the premises need to be true.
To show that this form is invalid, we demonstrate how it can lead from true premises to a false conclusion.
All apples are fruit. (Correct)
Bananas are fruit. (Correct)
Therefore, bananas are apples. (Wrong)
A valid argument with false premises may lead to a false conclusion:
All tall people are Greek.
John Lennon was tall.
Therefore, John Lennon was Greek.
When a valid argument is used to derive a false conclusion from false premises, the inference is valid because it follows the form of a correct inference. A valid argument can also be used to derive a true conclusion from false premises:
All tall people are musicians.
John Lennon was tall.
Therefore, John Lennon was a musician.
In this case we have two false premises that imply a true conclusion.
In mathematical logic and automated theorem proving, resolution could be a rule of inference resulting in a refutation theorem-proving technique for sentences in propositional logic and first-order logic. In alternative words, iteratively applying the resolution rule in an acceptable method allows for telling whether a propositional formula is satisfiable and for proving that a first-order formula is unsatisfiable; this methodology could prove the satisfiability of a first-order satisfiable formula, however not always, because it is the case for all ways for first-order logic. Resolution was introduced by John Alan Robinson in 1965.

Resolution Rule: The resolution rule in propositional logic is a single valid inference rule that produces a new clause implied by two clauses containing complementary literals. A literal is a propositional variable or the negation of a propositional variable. Two literals are said to be complements if one is the negation of the other (in the following, a_i is taken to be the complement to b_j). The resulting clause contains all the literals that do not have complements. Formally:

$$\frac{a_1 \vee \ldots \vee a_i \vee \ldots \vee a_n, \quad b_1 \vee \ldots \vee b_j \vee \ldots \vee b_m}{a_1 \vee \ldots \vee a_{i-1} \vee a_{i+1} \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_{j-1} \vee b_{j+1} \vee \ldots \vee b_m}$$

where
all as and bs are literals,
a_i is the complement to b_j, and

the dividing line stands for entails

The clause produced by the resolution rule is called the resolvent of the two input clauses.

When the two clauses contain more than one pair of complementary literals, the resolution rule can be applied (independently) for each such pair; however, the result is always a tautology.

Modus ponens can be seen as a special case of resolution of a one-literal clause and a two-literal clause.

A Resolution Technique: When coupled with a complete search algorithm, the resolution rule yields a sound and complete algorithm for deciding the satisfiability of a propositional formula, and, by extension, the validity of a sentence under a set of axioms.

This resolution technique uses proof by contradiction and is based on the fact that any sentence in propositional logic can be transformed into an equivalent sentence in conjunctive normal form. The steps are as follows.

All sentences in the knowledge base and the *negation* of the sentence to be proved (the *conjecture*) are conjunctively connected.

The resulting sentence is transformed into a conjunctive normal form with the conjuncts viewed as elements in a set, *S*, of clauses.

For example,

$$(A_1 \vee A_2) \wedge (B_1 \vee B_2 \vee B_3) \wedge (C_1)$$

gives rise to the set

$$S = \{A_1 \vee A_2, B_1 \vee B_2 \vee B_3, C_1\}.$$

Algorithm: The resolution rule is applied to all possible pairs of clauses that contain complementary literals. After each application of the resolution rule, the resulting sentence is simplified by removing repeated literals. If the sentence contains complementary literals, it is discarded (as a tautology). If not, and if it is not yet present in the clause set *S*, it is added to *S*, and is considered for further resolution inferences.

If after applying a resolution rule the *empty clause* is derived, the original formula is unsatisfiable (or *contradictory*), and hence, it can be concluded that the initial conjecture follows from the axioms.

If, on the other hand, the empty clause cannot be derived, and the resolution rule cannot be applied to derive any more new clauses, the conjecture is not a theorem of the original knowledge base.

One instance of this algorithm is the original Davis–Putnam algorithm that was later refined into the DPLL algorithm that removed the need for explicit representation of the resolvents.

This description of the resolution technique uses a set *S* as the underlying data-structure to represent resolution derivations. Lists, Trees and Directed Acyclic Graphs are other possible and common alternatives. Tree representations are more faithful to the fact that the resolution rule is binary. Together with a sequent notation for clauses, a tree representation also makes it clear to see how the resolution rule is related to a special case of the cut-rule, restricted to atomic cut-formulas. However, tree representations are not as compact as set or list representations, because they explicitly show redundant subderivations of clauses that are used more than once in the derivation of the empty clause. Graph representations can be as compact in the number of clauses as list representations and they also store structural information regarding which clauses were resolved to derive each resolvent.

A simple example

$$\frac{a \vee b, \quad \neg a \vee c}{b \vee c}$$

In plain language: Suppose a is false. In order for the premise $a \vee b$ to be true, b must be true. Alternatively, suppose a is true. In order for the premise $\neg a \vee c$ to be true, c must be true. Therefore, regardless of falsehood or veracity of a, if both premises hold, then the conclusion $b \vee c$ is true.

Resolution in First-Order Logic: 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 are replaced by Skolem functions.

$\neg P(x) \vee Q(x)$
$P(a)$
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
$\neg P(X)$
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
$X \mapsto a$

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

For another example, consider the syllogistic form
All Cretans are islanders.
All islanders are liars.
Therefore, all Cretans are liars.

Or more generally,
$\forall X\ P(X) \rightarrow Q(X)$
$\forall X\ Q(X) \rightarrow R(X)$
Therefore, $\forall X\ P(X) \rightarrow R(X)$
In CNF, the antecedents become:
$\neg P(X) \vee Q(X)$
$\neg Q(Y) \vee R(Y)$

(Note that the variable in the second clause was renamed to make it clear that variables in different clauses are distinct.)

Now, unifying $Q(X)$ in the first clause with $\neg Q(Y)$ in the second clause means that X and Y become the same variable anyway. Substituting this into the remaining clauses and combining them gives the conclusion:
$\neg P(X) \vee R(X)$

The resolution rule, as defined by Robinson, also incorporated factoring, which unifies two literals in the same clause, before or during the application of resolution as defined above. The resulting inference rule is refutation complete, in that a set of clauses is unsatisfiable if and only if there exists a derivation of the empty clause using resolution alone.

Program:

```
%% Sam's likes and dislikes in food
%% Considering the following will give some practice
%% in thinking about backtracking.
%% ?- likes(sam,dahl).
%% ?- likes(sam,chop_suey).
%% ?- likes(sam,pizza).
%% ?- likes(sam,chips).
%% ?- likes(sam,curry).
likes(sam,Food) :-
indian(Food),
mild(Food).
likes(sam,Food) :-
chinese(Food).
likes(sam,Food) :-
italian(Food).
likes(sam,chips).
indian(curry).
indian(dahl).
indian(tandoori).
indian(kurma).
mild(dahl).
mild(tandoori).
mild(kurma).
chinese(chow_mein).
chinese(chop_suey).
chinese(sweet_and_sour).
italian(pizza).
italian(spaghetti).
```

Output:

```
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 6.6.6)
Copyright (c) 1990-2013 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.
For help, use ?- help(Topic). or ?- apropos(Word).
1 ?- chdir('C:/Program Files (x86)/swipl/demo/').
true.
2 ?- consult('C:/Program Files (x86)/swipl/demo/likes.pl').
% C:/Program Files (x86)/swipl/demo/likes.pl compiled 0.02 sec, 17 clauses
true.
3 ?- likes(sam,pizza).
true.
4 ?- likes(sam,idle).
false.
5 ?-
```

Experiment No. 10

Ontology Creating, Editing and Authoring Using Protégé Tool

Aim: To study ontology creating, editing and authorizing using Protégétool.
Theory: Protégé is a free, open-source platform that gives a growing user community with a set of tools to construct domain models and knowledge-based applications with ontologies. At its core, Protégé implements a rich set of knowledge-modelling structures and actions that support the creation, visualization, and manipulation of ontologies in varied illustration formats. Protégé are often customized to supply domain-friendly support for making knowledge models and entering data. Further, Protégé can be extended by means of a plug-in design and a Java-based Application Programming Interface (API) for building knowledge-based tools and applications.

An ontology describes the ideas and relationships that are necessary in an exceedingly particular domain, providing a vocabulary for that domain in addition as a computerized specification of the meaning of terms utilized in the vocabulary. Ontologies vary from taxonomies and classifications, database schemas, to totally axiomatized theories. In recent years, ontologies are adopted in several business and scientific communities as some way to share, use and method domain data. Ontologies are currently central to several applications like scientific knowledge portals, data management and integration systems, electronic commerce, and semantic web services.

The Protégé platform supports two main ways of modelling ontologies:

The Protégé-Frames editor permits users to create and populate ontologies that are frame-based, in accordance with the Open knowledge base connectivity protocol (OKBC). in this model, an ontology consists of a group of classes organized during a subsumption hierarchy to represent a domain's salient ideas, a set of slots associated to categories to explain their properties and relationships, and a set of instances of these classes - individual exemplars of the ideas that hold specific values for his or her properties.

The Protégé-OWL editor permits users to create ontologies for the semantic web, especially within the W3C's web ontology Language (OWL). "An owl ontology might embody descriptions of categories, properties and their instances. Given such an ontology, the owl formal semantics specifies a way to derive its logical consequences, i.e. facts not literally present within the ontology, however entailed by the semantics. These entailments is also based on a single document or multiple distributed documents that are combined using defined owl mechanisms"

Ontology: Ontology could be a formal explicit description of ideas in a very domain of discourse (classes (sometimes called concepts)), properties of every concept describing varied options and attributes of the concept (slots (sometimes referred to as roles or properties)), and restrictions on slots (facets (sometimes referred to as role restrictions)). an ontology together with a set of individual instances of classes constitutes a knowledge base. In reality, there is a fine line wherever the ontology ends and therefore the knowledge base begins.

Classes are the main focus of most ontologies. classes describe ideas within the domain. as an example, a class of wines represents all wines. Specific wines are instances of this class. The Bordeaux wine within the glass front of you while you scan this document is an instance of the class of Bordeaux wines. a class will have subclasses that represent ideas that are a lot of specific than the superclass. as an example, we will divide the class of all wines into red,

white and rose. Alternatively, we will divide a class of all wines into sparkling and non-sparkling wines.

Creating Ontology: Ontology creation involves following steps:
Step 1: Determine the domain and scope of the ontology starting the development of an ontology by defining its domain and scope. That is, answer several basic questions:

What is the domain that the ontology will cover?
For what we are going to use the ontology?
For what types of questions the information in the ontology should provide answers?
Who will use and maintain the ontology?

The answers to these questions may change during the ontology-design process, but at any given time they help limit the scope of the model.

Step 2: Consider reusing existing ontologies. It is almost always value considering what somebody else has done and checking if we are able to refine and extend existing sources for our particular domain and task. Reusing existing ontologies may be a requirement if our system needs to move with different applications that have already committed to explicit ontologies or controlled vocabularies. several ontologies are already offered in electronic type and might be foreign into an ontology-development environment that you simply are using. The formalism within which an ontology is expressed usually doesnot matter, since several knowledge-representation systems will import and export ontologies. although a knowledge-representation system cannot work directly with a selected formalism, the task of translating an ontology from one formalism to a different is usually not a troublesome one.

Step 3: Enumerate important terms in the ontology. It is useful to write down a list of all terms we would like either to make statements about or to explain to a user. What are the terms we would like to talk about? What properties do those terms have? What would we like to say about those terms? For example, important wine-related terms will include wine, grape, winery, location, a wine's color, body, flavour and sugar content; different types of food, such as fish and red meat; subtypes of wine such as white wine, and so on. Initially, it is important to get a comprehensive list of terms without worrying about overlap between concepts they represent, relations among the terms, or any properties that the concepts may have, or whether the concepts are classes or slots.

Step 4: Define the classes and the class hierarchy. There are several possible approaches in developing a class hierarchy (Uschold and Gruninger 1996):

A top-down development process starts with the definition of the most general concepts in the domain and subsequent specialization of the concepts. For example, we can start with creating classes for the general concepts of Wine and Food. Then we specialise the wine class by creating some of its subclasses: white, red and rose. We can further categorize the Red wine class, for example, into Syrah, Red Burgundy, Cabernet Sauvignon, and so on.

A bottom-up development process starts with the definition of the most specific classes, the leaves of the hierarchy, with subsequent grouping of these classes into more general concepts. For Example, we start by defining classes for Pauillac and Margaux wines. We then create a common superclass for these two classes—Medoc—which in turn is a subclass of Bordeaux.

A combination development process is a combination of the top-down and bottom-up approaches: We define the more salient concepts first and then generalize and specialize them appropriately. We might start with a few top-level concepts such as Wine, and a few specific concepts, such as Margaux. We can then relate them to a middle-level concept, such as Medoc.

Then we may want to generate all of the regional wine classes from France, thereby generating a number of middle-level concepts.

Step 5: Define the properties of classes—slots. There are several types of object properties that can become slots in ontology:
"Intrinsic" properties such as the flavour of a wine;
"Extrinsic" properties such as a wine's name, and area it comes from
parts, if the object is structured; these can be both physical and abstract "parts" (e.g., the courses of a meal)
Relationships to other individuals; these are the relationships between individual members of the class and other items (e.g., the maker of a wine, representing a relationship between a wine and a winery, and the grape the wine is made from.)

Step 6: Define the facets of the slots. Slots can have different facets describing the value type, allowed values, the number of the values (cardinality), and other features of the values the slot can take.

Step 7: Create instances. The last step is creating individual instances of classes in the hierarchy. Defining an individual instance of a class requires (1) choosing a class, (2) creating an individual instance of that class, and (3) filling in the slot values.

Editing: One of the primary and most vital choices within the design of an owl editor is how to to|a way to} show class expressions in an exceedingly user-friendly however economical way. Syntax planned within the owl specification [8] is clearly too verbose to be of any use here. The owl Abstract Syntax [9] is far more easy, however still quite verbose. For the owl Plug-in, we selected to use an expression syntax based on standard dl symbols, like 8 and t. These symbols permit to the system display even complicated nested expressions in a single row. A trade-off from this syntax is that some characters don't seem to be found on standard keyboards. The owl Plug-in provides a comfortable expression editor that permits users to quickly assemble expressions with either the mouse or the keyboard. The special characters are mapped onto keys known from languages like Java (e.g., owl: intersection of which is entered with the & key). To alter editing, keyboard users will exploit a syntax completion mechanism known from programming environments, that semi-automatically completes partial names after the uses has pressed tab. The expression editor is invoked by double-click on a class expression, then pops up directly below the expression. For extremely complicated expressions, users will open a multi-line editor in an additional window, that formats the expression using indentation.

The OWL Plug-in helps new users to get acquainted with the expression syntax. An English prose text is shown as a "tool tip" when the mouse is moved over the expression. For example, "9 has Pet Cat" is displayed as "Any object which has a cat as its pet". Figure 2Protégé provides a comfortable editor for arbitrary OWL expressions. 4 Editing Class Descriptions Another major design decision for a DL class editor is how to edit the logical class definitions. Protégé users are accustomed to an object-centred view to the interface which has required some effort to adapt to OWL. The distinction between defined and primitive classes simply is not found in frame-style or object-oriented modelling paradigms, and this can compound users' confusion when learning the DL paradigm. In the OWL specification, there is a lack of uniformity between defined classes and primitive classes. Multiple necessary conditions are represented by multiple rdfs: sub Class Of statements whose intersection is implied, whereas sets of multiple necessary and sufficient conditions are represented by an owl equivalent class block containing an explicit intersection class. Although logically consistent, experience has

shown that many users find the difference confusing. As a result, it was decided that the user interface should not simply reflect the structure suggested by the OWL specification but attempt to provide a clearer more uniform presentation to users. During the evolution of the OWL plug-in we experimented with several interface designs, partly based on existing tools such as the Protégé core system and OilEd, partly on suggestions from our colleagues and users.

OilEd has 2 modes: one to “partially” define a class with solely necessary conditions, the other to “completely” define a class with necessary conditions. there is a button to change between these 2 modes. whereas this feature allowed the oiled developers to produce customized widgets for varied forms of class descriptions (e.g. a widget for only restrictions), it has the disadvantage that users have to be compelled to maintain separate class axioms in a separate pane for the required restrictions of classes that also are “completely” defined by a set of necessary conditions. there is nobody pane in oiled within which one will see both the sets of necessary and decent conditions and any further necessary conditions (i.e. axioms taking the defined class as their antecedent.)

As shown in the centre of Figure 5, the owl Plug-in solves this problem by means of a list of conditions, organized into blocks of necessary, and inherited (i.e., inferred) conditions. Each of the necessary blocks represents one equivalent intersection class, and solely those inherited conditions are listed that haven't been further restricted above in the hierarchy

The editor supports drag-and-drop between blocks in the conditions list, and copy-and-paste of expressions. It also supports changing super classes by dragging a class from one parent to another in the class tree on the left-hand side of the window. In addition to the list of conditions, there is also a custom-tailored widget for entering disjoint classes, which has special support for typical design patterns such as making all siblings disjoint. This rather object-centred design of the OWL Classes tab makes it possible to maintain the whole class definition on a single screen.

Authoring Ontology: Change tracking. Collaborative Protégé records all actions made by a user and stores them in a structured log as instances of the Change class of the CHAO ontology. The change log contains

Because the changes are also stored as instances of the CHAO ontology, they can also be annotated to record design rationale, or other information. For projects with large sets of annotations, search and filtering are crucial features. Users of Collaborative Protégé can filter and search in each of the collaboration panels using different criteria, such as author, date, annotation text, and annotation type. Chat. Users connected at the same time to a Protégé server can discuss and exchange live messages. One feature that sets Collaborative Protégé chat functionality apart from other chat clients is the support for sending links to entities in the ontology (e.g. a class).

The user who receives a chat message containing an entity link, can simply click on that link, and she will see the definition for that entity. We designed Collaborative Protégé in such a way that it is very easy to extend the annotation types available in the user interface. It only requires adding a subclass of Annotation in the CHAO ontology and the tool will be able to handle the new type of annotation. Collaborative Protégé is built using a plug-in architecture, meaning that other groups can very easily add their own custom collaboration panel. The extensibility is made possible by exposing all the collaboration information through API calls, so they can be easily used and integrated in other applications.

Experiment No. 11

Inductive Learning Using Weka Tool

11.1 Implement Decision Tree Learning

Aim: Inductive learning using Weka tool.

Theory: Decision tree learning

A decision tree is a classifier expressed as a recursive partition of the instance space. The decision tree consists of nodes that form a rooted tree, that means it is a directed tree with a node known as "root" that has no incoming edges. All alternative nodes have exactly one incoming edge. A node with outgoing edges is termed an internal or test node. All other nodes are known as leaves (also referred to as terminal or decision nodes). in a decision tree, every internal node splits the instance space into 2 or a lot of sub-spaces according to a particular discrete function of the input attributes values. within the simplest and most frequent case, every check considers a single attribute, specified the instance space is partitioned according to the attribute's value. within the case of numeric attributes, the condition refers to a range.

Example: For each feature encountered in the tree, the arc corresponding to the value of the example for that feature is followed. When a leaf is reached, the classification corresponding to that leaf is returned.

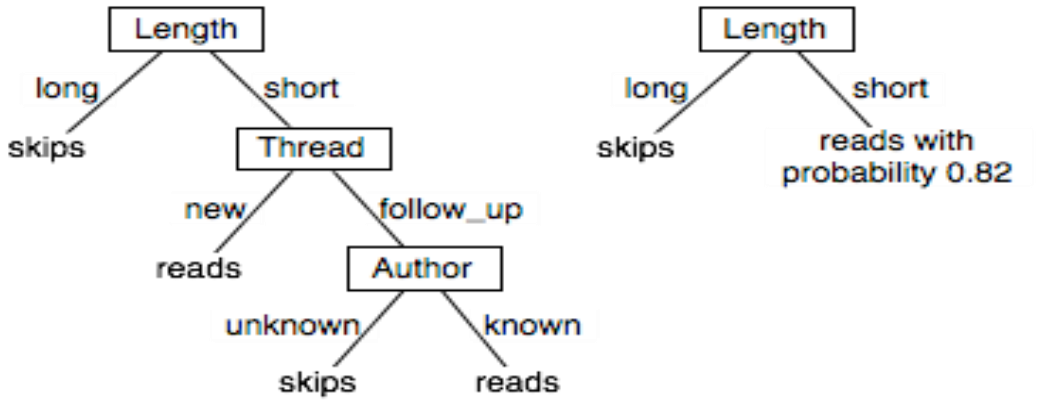


Figure Two decision trees.

A deterministic decision tree, in which all of the leaves are classes, can be mapped into a set of rules, with each leaf of the tree corresponding to a rule. The example has the classification at the leaf if all of the conditions on the path from the root to the leaf are true.

The leftmost decision tree in Figure 5can be represented as the following rules:

Skips ←long.
Reads ←short ∧new
Reads ←short∧followUp∧known
Skips ←short ∧follow Up∧unknown

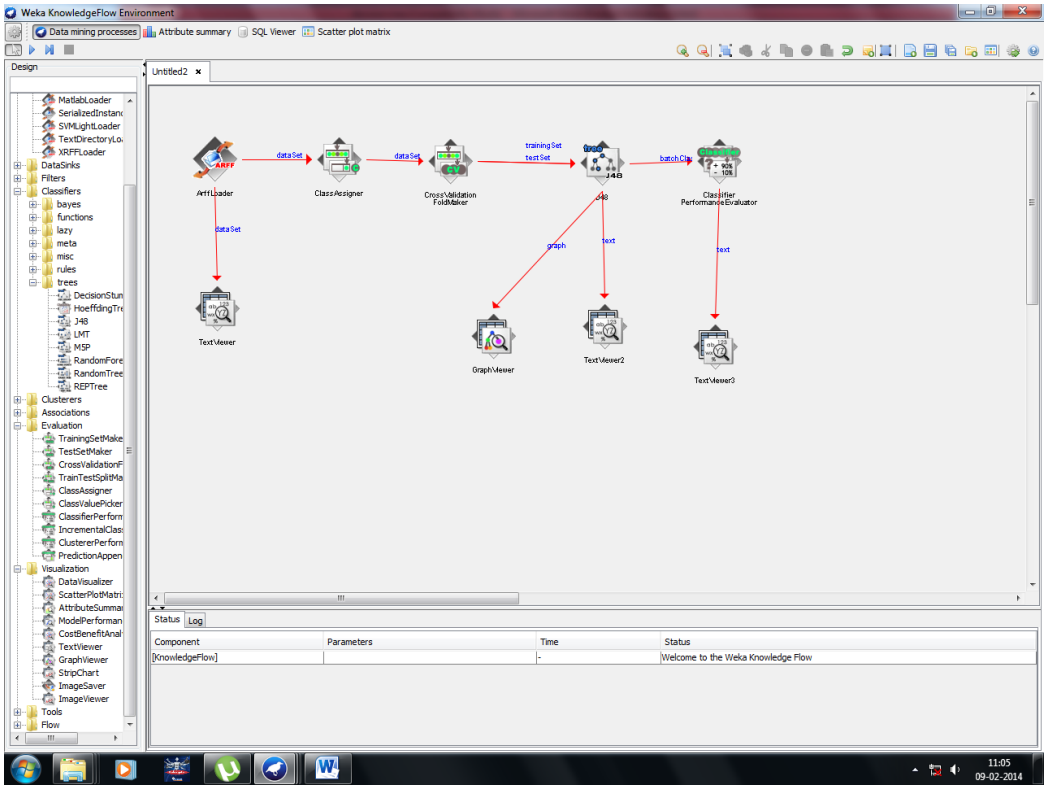
Algorithm:

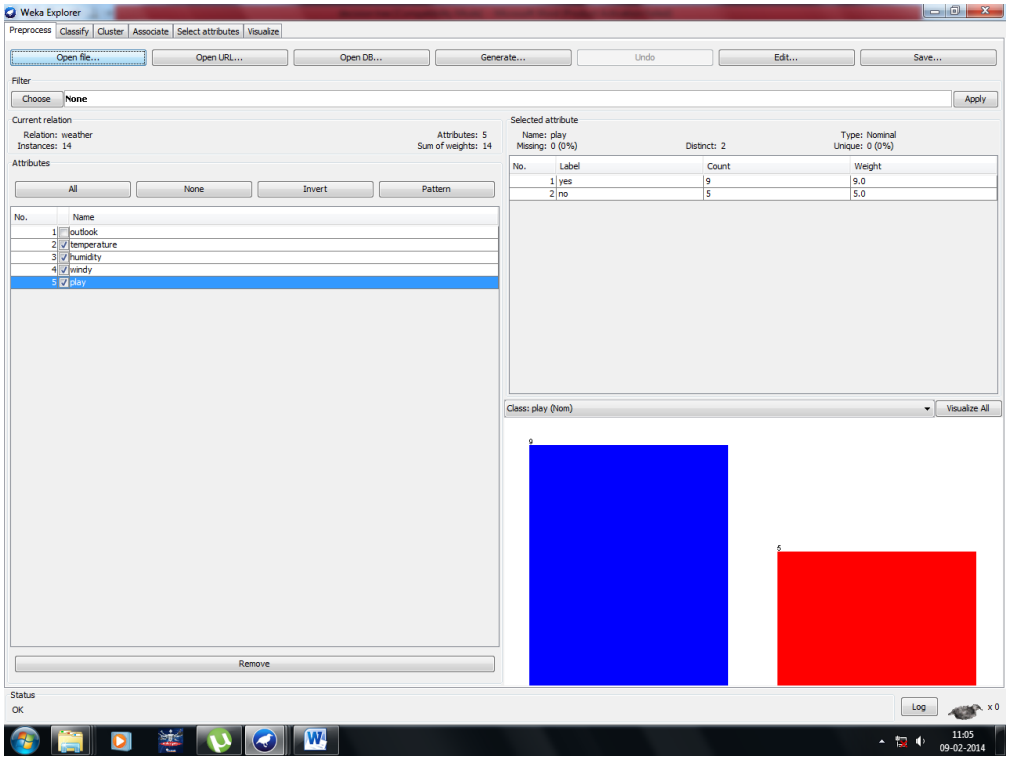
```
1: Procedure DecisionTreeLearner(X,Y,E)
2:      Inputs
3:          X: set of input features, X={X1,...,Xn}
4:          Y: target feature
5:          E: set of training examples
6:      Output
7:          decision tree
8:      if stopping criterion is true then
9:          return pointEstimate(Y,E)
10:      else
11:         Select feature Xi∈X, with domain {v1,v2}
12:         let E1={e∈E: val(e,Xi)=v1}
13:         let T1=DecisionTreeLearner(X \ {Xi},Y,E1)
14:         let E2={e∈E: val(e,Xi)=v2}
15:         let T2=DecisionTreeLearner(X \ {Xi},Y,E2)
16:         return ⟨Xi=v1, T1, T2⟩
```

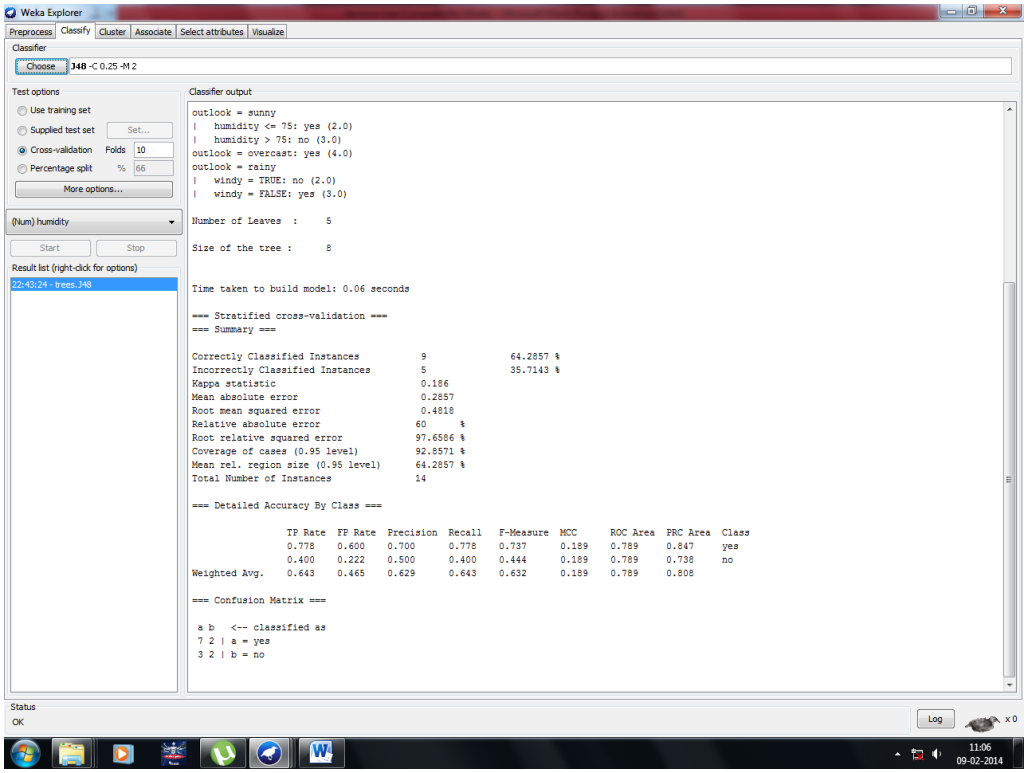
A decision tree can be incrementally built from the top down by recursively selecting a feature to split on and partitioning the training examples with respect to that feature. In above algorithm, the procedure *DecisionTreeLearner* learns a decision tree for binary attributes. The decisions regarding when to stop and which feature to split on are left undefined.

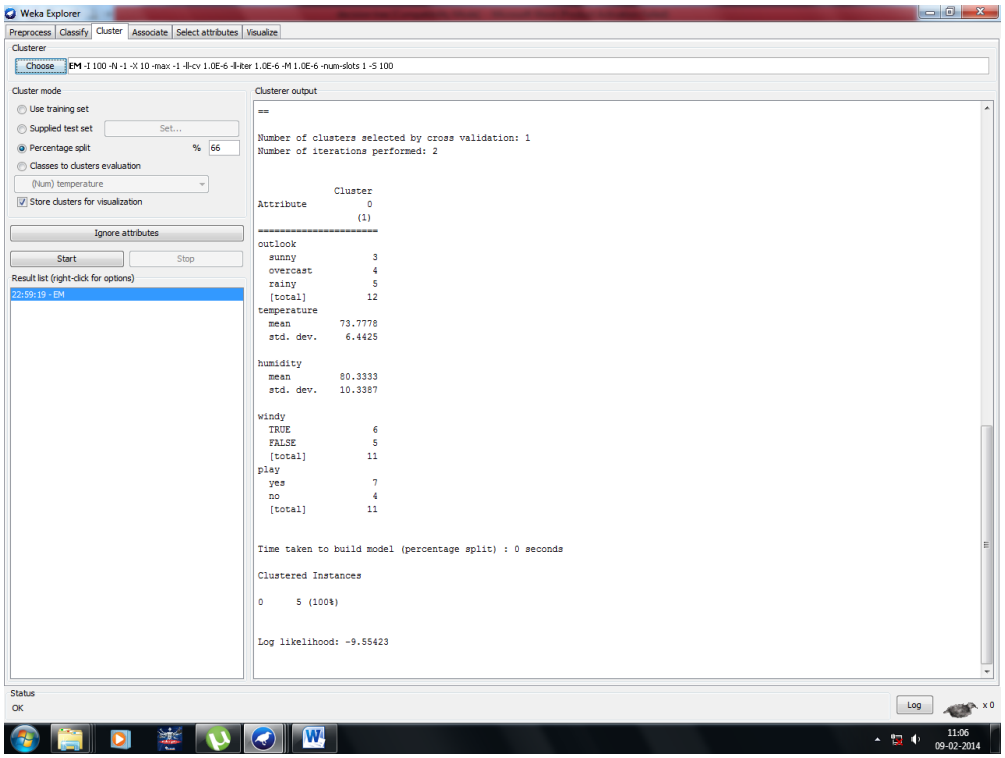
The algorithm DecisionTreeLearner builds a decision tree from the top down as follows: The input to the algorithm is a set of input features, a target feature, and a set of examples. The learner first tests if some stopping criterion is true. If the stopping criterion is true, it returns a point estimate for Y, which is either a value for Y or a probability distribution over the values for Y. If the stopping criterion is not true, the learner selects a feature X_i to split on, and for each value v of this feature, it recursively builds a subtree for those examples with $X_i=v$. The returned tree is represented here in terms of triples representing an if-then-else structure.

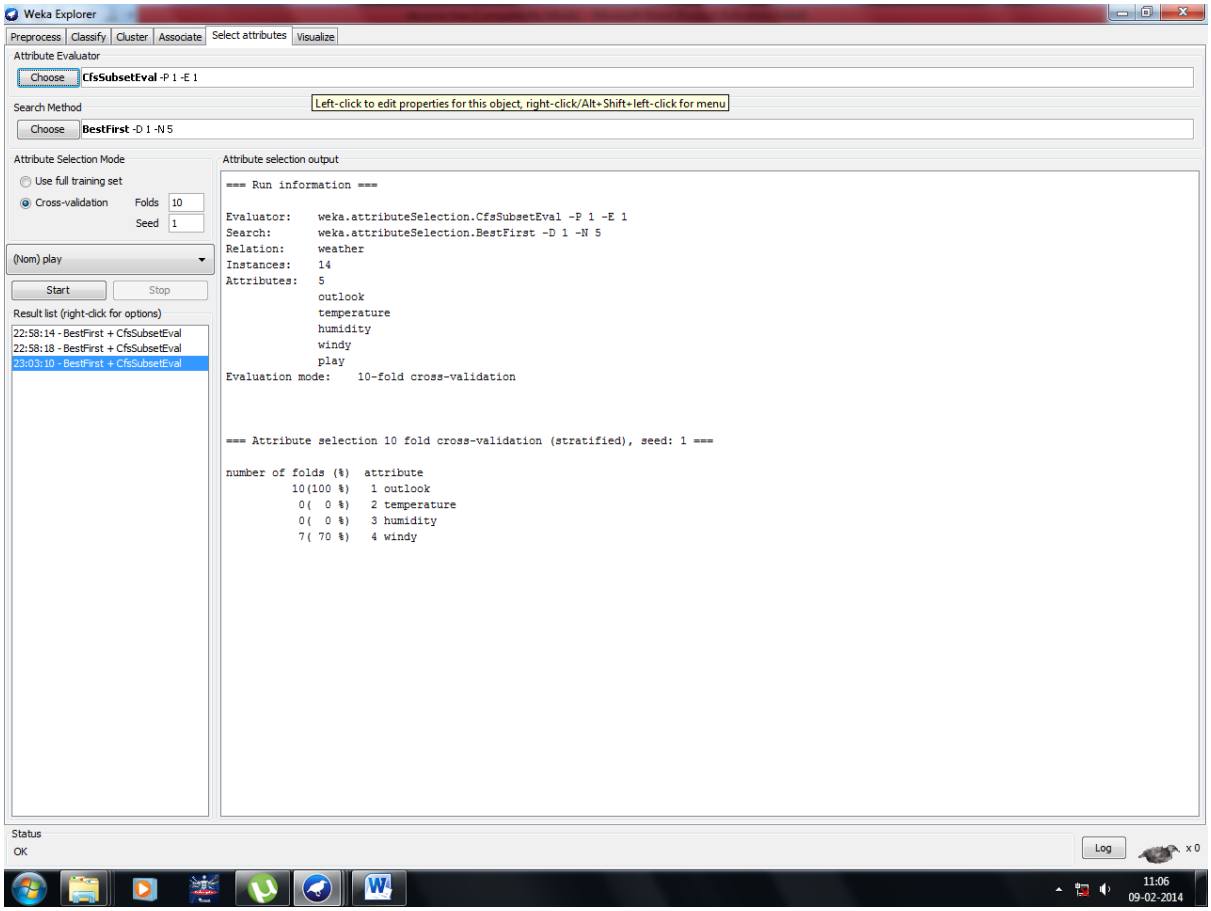
Conclusion: A decision tree or a classification tree is a tree in which each internal (non-leaf) node is labelled with an input feature. The arcs coming from a node labelled with a feature are labelled with each of the possible values of the feature. Each leaf of the tree is labelled with a class or a probability distribution over the classes.


Weka KnowledgeFlow Environment
Data mining processes
Attribute summary
SQL Viewer
Scatter plot matrix
Design
Untitled2
ArffLoader
ClassAssigner
CrossValidationFoldMaker
J48
ClassifierPerformanceEvaluator
TextViewer
GraphViewer
TextViewer2
TextViewer3
Status
Log
Component
Parameters
Time
Status
[KnowledgeFlow]
Welcome to the Weka Knowledge Flow


Weka Explorer
Preprocess
Classify
Cluster
Associate
Select attributes
Visualize
Open file...
Open URL...
Open DB...
Generate...
Undo
Edit...
Save...
Filter
Choose
None
Apply
Current relation
Relation: weather
Instances: 14
Attributes: 5
Sum of weights: 14
Selected attribute
Name: play
Missing: 0 (0%)
Distinct: 2
Type: Nominal
Unique: 0 (0%)
Attributes
All
None
Invert
Pattern
No.
Name
outlook
temperature
humidity
windy
play
Label
Count
Weight
yes
no
Class: play (Nom)
Visualize All
Remove
Status
OK
Log


Weka Explorer
Preprocess
Classify
Cluster
Associate
Select attributes
Visualize
Classifier
Choose
J48 -C 0.25 -M 2
Test options
Use training set
Supplied test set
Set...
Cross-validation
Folds
10
Percentage split
%
66
More options...
(Num) humidity
Start
Stop
Result list (right-click for options)
22:43:24 - trees.J48
Classifier output
outlook = sunny
| humidity <= 75: yes (2.0)
| humidity > 75: no (3.0)
outlook = overcast: yes (4.0)
outlook = rainy
| windy = TRUE: no (2.0)
| windy = FALSE: yes (3.0)
Number of Leaves : 5
Size of the tree : 8
Time taken to build model: 0.06 seconds
=== Stratified cross-validation ===
=== Summary ===
Correctly Classified Instances 9 64.2857 %
Incorrectly Classified Instances 5 35.7143 %
Kappa statistic 0.186
Mean absolute error 0.2857
Root mean squared error 0.4818
Relative absolute error 60 %
Root relative squared error 97.6586 %
Coverage of cases (0.95 level) 92.8571 %
Mean rel. region size (0.95 level) 64.2857 %
Total Number of Instances 14
=== Detailed Accuracy By Class ===
TP Rate FP Rate Precision Recall F-Measure MCC ROC Area PRC Area Class
0.778 0.600 0.700 0.778 0.737 0.189 0.789 0.847 yes
0.400 0.222 0.500 0.400 0.444 0.189 0.789 0.738 no
Weighted Avg. 0.643 0.465 0.629 0.643 0.632 0.189 0.789 0.808
=== Confusion Matrix ===
a b <-- classified as
7 2 | a = yes
3 2 | b = no
Status
OK
Log
x 0
11:06
09-02-2014


Weka Explorer
Preprocess
Classify
Cluster
Associate
Select attributes
Visualize
Clusterer
Choose
EM -I 100 -N -1 -X 10 -max -1 -ll-cv 1.0E-6 -ll-iter 1.0E-6 -M 1.0E-6 -num-slots 1 -S 100
Cluster mode
Use training set
Supplied test set
Set...
Percentage split
%
66
Classes to clusters evaluation
(Num) temperature
Store clusters for visualization
Ignore attributes
Start
Stop
Result list (right-click for options)
22:59:19 - EM
Clusterer output
==
Number of clusters selected by cross validation: 1
Number of iterations performed: 2
Cluster
Attribute 0
(1)
outlook
sunny 3
overcast 4
rainy 5
[total] 12
temperature
mean 73.7778
std. dev. 6.4425
humidity
mean 80.3333
std. dev. 10.3387
windy
TRUE 6
FALSE 5
[total] 11
play
yes 7
no 4
[total] 11
Time taken to build model (percentage split) : 0 seconds
Clustered Instances
0 5 (100%)
Log likelihood: -9.55423
Status
OK
Log
x 0
11:06
09-02-2014


Weka Explorer
Preprocess
Classify
Cluster
Associate
Select attributes
Visualize
Attribute Evaluator
Choose
CfsSubsetEval -P 1 -E 1
Search Method
Choose
BestFirst -D 1 -N 5
Left-click to edit properties for this object, right-click/Alt+Shift+left-click for menu
Attribute Selection Mode
Use full training set
Cross-validation
Folds
10
Seed
1
(Nom) play
Start
Stop
Result list (right-click for options)
22:58:14 - BestFirst + CfsSubsetEval
22:58:18 - BestFirst + CfsSubsetEval
23:03:10 - BestFirst + CfsSubsetEval
Attribute selection output
=== Run information ===
Evaluator: weka.attributeSelection.CfsSubsetEval -P 1 -E 1
Search: weka.attributeSelection.BestFirst -D 1 -N 5
Relation: weather
Instances: 14
Attributes: 5
outlook
temperature
humidity
windy
play
Evaluation mode: 10-fold cross-validation
=== Attribute selection 10 fold cross-validation (stratified), seed: 1 ===
number of folds (%) attribute
10(100 %) 1 outlook
0(0 %) 2 temperature
0(0 %) 3 humidity
7(70 %) 4 windy
Status
OK
Log
x 0
11:06
09-02-2014

Experiment No. 12

Study of Seiko DTRANS RT 3200 Robot

Aim: Study of Seiko DTRANS RT 3200 robot.

Theory: Robots have the potential to change our economy, our health, our standard of living, our knowledge and the world in which we live. As the technology progresses, we are finding new ways to use robots.

The RT 3200 robot delivers an eleven-pound payload across a thirty-six inch pick and a place cycle in less than 1.3 seconds, maintaining a repeatability of 0.001. In addition to Speed and Precision, factor in a New powerful control system, and you have an exceptional automation system.

The net benefits are as follows:

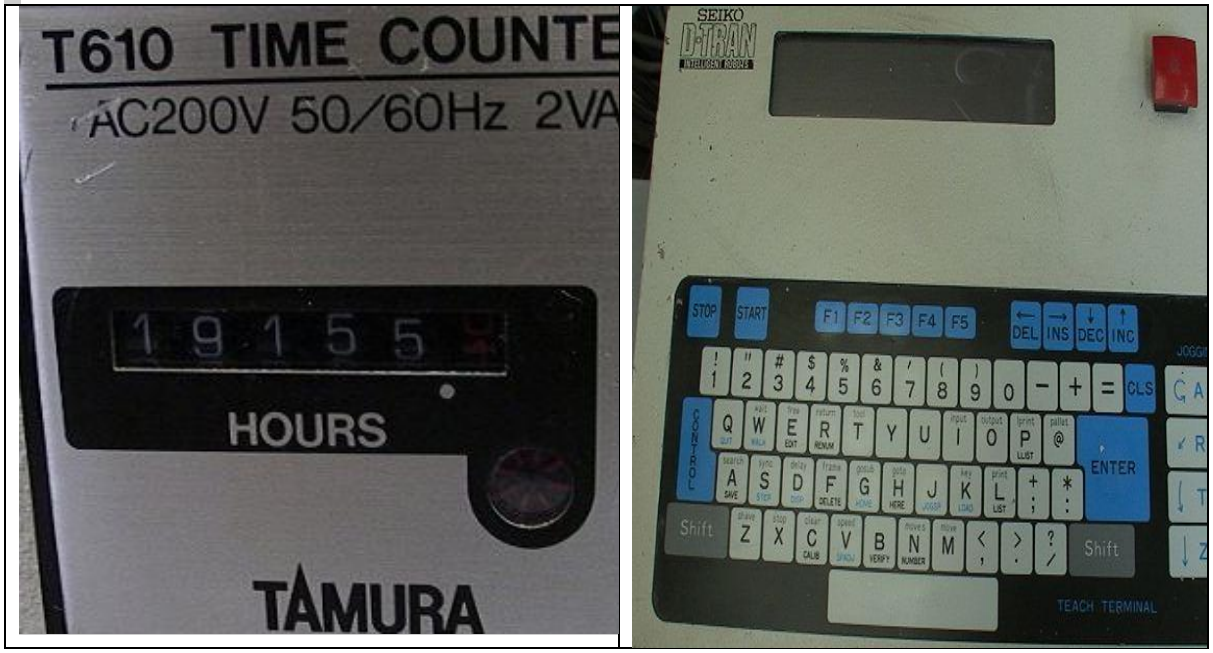
1. Fastest cycle time of any assembly robot assures you higher throughput and increased productivity.
2. Most powerful control system insures quick system implementation and fast return on your investment.
3. Modular design and proven reliability make this robot very flexible and easy to service.

We include the robot arm, controller, cables, teach pendant and documentation. (The teach pendant is just sitting on the top of the arm, it is not mounted there.)

This robot arm moves up and down at the top of the base, rotates at the top of the base, goes in and out of the arm and rotates at the end of the arm.

(a) Controller

(b) Hour Meter: Shows the hour meter at 19155 hours



(c) Teach Terminal: Enable one to move the robot around as desired and program the points and motions you want.

Successful applications:

1. Mechanical assembly
2. Machine tool load/unload
3. Waterjet cutting
4. Assembly adjustments
5. Wire harness and connector assembly

Experiment No. 13

Mini Expert System Using PROLOG

Aim: Implement Mini Expert system.

Theory: Robots have the potential to change our economy, our health, our standard of living, our knowledge and the world in which we live. As the technology progresses, we are finding new ways to use robots.

The RT 3200 robot delivers an eleven-pound payload across a 36 inch pick and a place cycle in less than 1.3 seconds, maintaining a repeatability of 0.001. In addition to speed and precision, factor in a new powerful control system, and you have an exceptional automation system.

The net benefits are:

1. Fastest cycle time of any assembly robot assures you higher throughput and increased productivity.

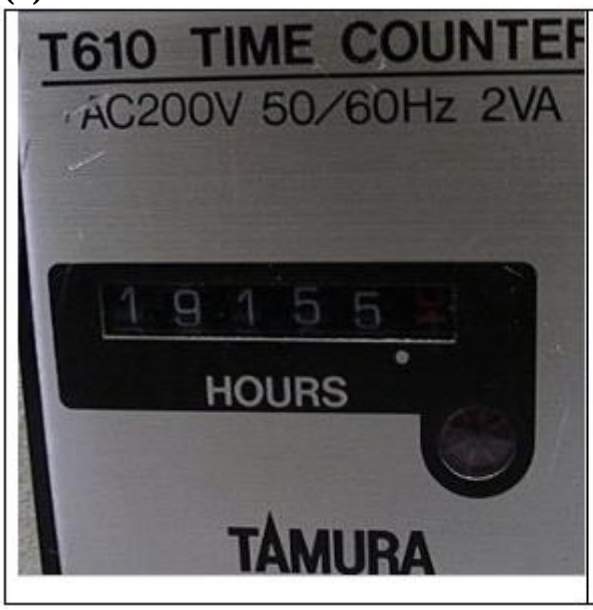
2. Most powerful control system insures quick system implementation and fast return on your investment.
3. Modular design and proven reliability make this robot very flexible and easy to service.

Included is the robot arm, controller, cables, teach pendant and documentation. (The teach pendant is just sitting on the top of the arm, it is not mounted there.)

This robot arm moves up and down at the top of the base, rotates at the top of the base, goes in and out of the arm and rotates at the end of the arm.

(a) Controller

(b) Hour Meter: Shows the hour meter at 19155 hours



(c) Teach Terminal: Enable one to move the robot around as desired and program the points and motions you want.

Successful applications:

1. Mechanical assembly

1. Machine tool load/unload
2. Water jet cutting
3. Assembly adjustments
4. Wire harness and connector assembly

Conclusion: RT 3200 robot is fastest assembly robot in the world. It offers a choice of control schemes, flexible work envelope, modular design and a history of proven reliability.

Program:

```
state(ac,bd,a):-
        write('move to b'),nl,
        state(ac,bd,b).
state(ac,bd,b):-
        write('clean b'),nl,
```

```
        state(ac,bc,b).
state(ac,bc,b):-
        write('both a & b are clean').
state(ad,bc,a):-
write('clean a'),nl,
        state(ac,bc,a).
state(ac,bc,a):-

        write('both a & b are clean').

state(ad,bc,b):-
        write('move to a'),nl,
        state(ad,bc,a).
state(ad,bd,a):-
        write('clean a'),nl,
        state(ac,bd,a).
state(ad,bd,b):-
        write('clean b'),nl,
        state(ad,bc,b).
```

Output:

```
1 ?-
% c:/Users/sakec/Desktop/vaccum.pl compiled 0.00 sec, 580
bytes 1 ?- state(ad,bd,b).
clean
b
move
to     a
clean
a

both  a  &  b  are
clean true .
```

2 ?- state(ad,bc,b).

move to a

clean a

both a & b are clean

true

Experiment No. 14

Programming Using Python

14.1 Water Jug Problem Using Python

14.1.1 Two Jug Problem

Given 2 jugs of capacities: 5 and 7 Litres. Using these 2 jugs to obtain exactly 4 Litres water .

Code :

```
def pour(jug1, jug2):
    max1, max2, fill = 5, 7, 4  #Change maximum capacity and final capacity
    print("%d\t%d" % (jug1, jug2))
    if jug2 is fill:
        return
    elif jug2 is max2:
        pour(0, jug1)
    elif jug1 != 0 and jug2 is 0:
        pour(0, jug1)
    elif jug1 is fill:
        pour(jug1, 0)
    elif jug1 < max1:
        pour(max1, jug2)
    elif jug1 < (max2-jug2):
        pour(0, (jug1+jug2))
    else:
        pour(jug1-(max2-jug2), (max2-jug2)+jug2)

print("JUG1\tJUG2")
pour(0, 0)
```

Output :

```
JUG1  JUG2
0     0
5     0
0     5
5     5
3     7
0     3
5     3
1     7
0     1
5     1
0     6
5     6
```

4 7
0 4

14.1.2 Three Jug Problem

Problem: Given 3 jugs of capacities: 12, 8 and 5 litres. Our 12 L jug is completely filled. Using these 3 jugs split the water to obtain exactly 6 Litres.

So I thought of writing a code in python to obtain the solution to the problem, instead of doing hit and trial.

I used DFS to search through all the states of the jugs. At each state, we'll have certain choices of emptying water from one jug into another. We'll try each choice, calling our function for each state, and if we reach the goal state, we stop.

[Note that the given program could be made smaller/modular, but it is more understandable given this way. Also, DFS might not give an optimal (best path) solution. For that use BFS]

Code :

```
# 3 water jugs capacity -> (x,y,z) where x>y>z
# initial state (12,0,0)
# final state (6,6,0)


capacity = (12,8,5)
# Maximum capacities of 3 jugs ->x,y,z
x = capacity[0]
y = capacity[1]
z = capacity[2]

# to mark visited states
memory = {}

# store solution path
ans = []

def get_all_states(state):
  # Let the 3 jugs be called a,b,c
  a = state[0]
  b = state[1]
  c = state[2]

  if(a==6 and b==6):
ans.append(state)
      return True

  # if current state is already visited earlier
  if((a,b,c) in memory):
      return False
```

```
  memory[(a,b,c)] = 1

  #empty jug a
  if(a>0):
     #empty a into b
     if(a+b<=y):
        if( get_all_states((0,a+b,c)) ):
ans.append(state)
           return True
     else:
        if( get_all_states((a-(y-b), y, c)) ):
ans.append(state)
           return True
     #empty a into c
     if(a+c<=z):
        if( get_all_states((0,b,a+c)) ):
ans.append(state)
           return True
     else:
        if( get_all_states((a-(z-c), b, z)) ):
ans.append(state)
           return True

  #empty jug b
  if(b>0):
     #empty b into a
     if(a+b<=x):
        if( get_all_states((a+b, 0, c)) ):
ans.append(state)
           return True
     else:
        if( get_all_states((x, b-(x-a), c)) ):
ans.append(state)
           return True
     #empty b into c
     if(b+c<=z):
        if( get_all_states((a, 0, b+c)) ):
ans.append(state)
           return True
     else:
        if( get_all_states((a, b-(z-c), z)) ):
ans.append(state)
           return True

  #empty jug c
  if(c>0):
     #empty c into a
     if(a+c<=x):
        if( get_all_states((a+c, b, 0)) ):
ans.append(state)
```

```
            return True
        else:
            if( get_all_states((x, b, c-(x-a))) ):
ans.append(state)
            return True
        #empty c into b
        if(b+c<=y):
            if( get_all_states((a, b+c, 0)) ):
ans.append(state)
            return True
        else:
            if( get_all_states((a, y, c-(y-b))) ):
ans.append(state)
            return True

    return False

initial_state = (12,0,0)
print("Starting work...\n")
get_all_states(initial_state)
ans.reverse()
for i in ans:
    print(i)
```

```
Starting work...


(12, 0, 0)
(4, 8, 0)
(0, 8, 4)
(8, 0, 4)
(8, 4, 0)
(3, 4, 5)
(3, 8, 1)
(11, 0, 1)
(11, 1, 0)
(6, 1, 5)
(6, 6, 0)
```

14.2 Wumpus World Problem Using Python

Code :

```
dirs= ["right","left","up","down"]
gm = 1
gn = 1
pm = 3
pn = 0
```

```
h = 0
visited = []
w = [[2,0,0,0],[1,2,0,2],[2,0,2,1],[0,0,0,2]]
print w
p = [[0,0,4,3],[0,4,3,4],[0,0,4,0],[0,4,3,4]]
print p
k = [["","","",""],["","","",""],["","","",""],["","","",""]]
k[pm][pn] = 'S'
print k
tl = []
tl2 = []
present = ""
prev = ""
tmp = ""
tv = 1
mypath = []
def getMatrix(msg,m):
        print msg
        matrix = []
        for i in range(m):
                rl = []
                tmp = raw_input();
                tmp = tmp.replace(' ', '')
                for i in range(len(tmp)):
                        fl =  int(tmp[i])
                        rl.append(fl)
                matrix.append(rl)
                rl = []
        print matrix
def setdirections_for(i,j):
        si = str(i)
        sj = str(j)
        r = str(j+1)
        l = str(j-1)
        u = str(i-1)
        d = str(i+1)
        if i is 0 and j is 0:
                dirs[0] = si + r
                dirs[1] = ""
                dirs[2] = ""
                dirs[3] = d  + sj
                #print i," ",j,"down right"
        elif i is 0 and j is 3:
                dirs[0] = ""
                dirs[1] = si + l
                dirs[2] = ""
                dirs[3] = d  + sj
                #print i," ",j,"left down"
        elif i is 3 and j is 0:
                dirs[0] = si + r
```

```
                dirs[1] = ""
                dirs[2] = u  + sj
                dirs[3] = ""
                #print i," ",j,"up right"
        elif i is 3 and j is 3:
                dirs[0] = ""
                dirs[1] = si + l
                dirs[2] = u  + sj
                dirs[3] = ""
                #print i," ",j,"left up"
        elif i is 0:
                dirs[0] = si + r
                dirs[1] = si + l
                dirs[2] = ""
                dirs[3] = d  + sj
                #print i," ",j,"leftdown right"
        elif i is 3:
                dirs[0] = si + r
                dirs[1] = si + l
                dirs[2] = u  + sj
                dirs[3] = ""
                #print i," ",j,"leftup right"
        elif j is 0:
                dirs[0] = si + r
                dirs[1] = ""
                dirs[2] = u  + sj
                dirs[3] = d  + sj
                #print i," ",j,"up right down"
        elif j is 3:
                dirs[0] = ""
                dirs[1] = si + l
                dirs[2] = u  + sj
                dirs[3] = d  + sj
                #print i," ",j,"left up down"
        else:
                dirs[0] = si + r
                dirs[1] = si + l
                dirs[2] = u  + sj
                dirs[3] = d  + sj
                #print i," ",j,"left up right down"
def getdiagonals_for(i,j):
        iup     =       str(i-1)
        idown   =       str(i+1)
        jright  =       str(j+1)
        jleft   =       str(j-1)
        if i is 0 and j is 0:
                dirs[0] = ""
                dirs[1] = ""
                dirs[2] = ""
                dirs[3] = idown + jright
```

```
            #print i," ",j,"down right"
elif i is 0 and j is 3:
            dirs[0] = ""
            dirs[1] = ""
            dirs[2] = idown + jleft
            dirs[3] = ""
            #print i," ",j,"left down"
elif i is 3 and j is 0:
            dirs[0] = ""
            dirs[1] = iup + jright
            dirs[2] = ""
            dirs[3] = ""
            #print i," ",j,"up right"
elif i is 3 and j is 3:
            dirs[0] = iup + jleft
            dirs[1] = ""
            dirs[2] = ""
            dirs[3] = ""
            #print i," ",j,"left up"
elif i is 0:
            dirs[0] = ""
            dirs[1] = ""
            dirs[2] = idown + jleft
            dirs[3] = idown + jright
            #print i," ",j,"leftdown right"
elif i is 3:
            dirs[0] = iup + jleft
            dirs[1] = iup + jright
            dirs[2] = ""
            dirs[3] = ""
            #print i," ",j,"leftup right"
elif j is 0:
            dirs[0] = ""
            dirs[1] = iup + jright
            dirs[2] = ""
            dirs[3] = idown + jright
            #print i," ",j,"up right down"
elif j is 3:
            dirs[0] = iup + jleft
            dirs[1] = ""
            dirs[2] = idown + jleft
            dirs[3] = ""
            #print i," ",j,"left up down"
else:
            dirs[0] = iup + jleft
            dirs[1] = iup + jright
            dirs[2] = idown + jleft
            dirs[3] = idown + jright
            #print i," ",j,"left up right down"
```

```
def allowedsteps(ls):
	for objs in dirs:
		if objs is not "":
			ls.append(objs)
	return ls
def checkforwumpus(steps):
	if w[int(steps[0])][int(steps[1])] is 1:
		print "Player got killed"
		print visited
		print k
		exit(0)
	elif w[int(steps[0])][int(steps[1])] is 2:
		return '2'
	else:
		return '0'
def checkforpit(steps):
	if p[int(steps[0])][int(steps[1])] is 3:
		print "Player got killed"
		print k
		exit(0)
	elif p[int(steps[0])][int(steps[1])] is 4:
		return '4'
	else:
		return '0'
def getdiagonals(step):
	ls = []
	getdiagonals_for(int(step[0]),int(step[1]))
	ls = allowedsteps(ls)
	return ls
def stepintocell(step):
	ws = checkforwumpus(step)
	pb = checkforpit(step)
	return ws+pb
def applylogic_from_knowledge():
	global tl2,present,tv
	print "\n\nApplying logic"
	for step in tl:
		print "_____thinking for %s_____"%(step)
		if step in visited:
			print step,"is visited so avoid thinking about that\n"
			if int(step[0]) is pm and int(step[0]) is pn:
				print "1"
			elif '2' in k[int(step[0])][int(step[1])]:
				print "2"
		else:
			setdirections_for(int(step[0]),int(step[1]))
			tl2 = tl2[0:0]
			tl2 = allowedsteps(tl2)
			print "Connections of",step,"checking for :",tl2
```

```
                        #print present
                        #print prev
                        for st in tl2:
                                if st  == present:
                                        print st,"is it is present"
                                else:
                                        print "**********************checking :",st
                                        #print "Visited :",visited
                                        #print present,step,start
                                        dg = getdiagonals(st)
                                        print "Diagonals",dg,
                                        if st in start:
                                                print st,"is Start state"
                                        elifst == present:
                                                print st,"is Present state"
                                        elifst in visited:
                print st,"yes it is in visited"
                                                if (k[int(st[0])][int(st[1])] in ['04','40','02','20']) and
(k[int(present[0])][int(present[1])] in ['02','20','04','40']):
print "Applying first condition and is true"
                                                        k[int(step[0])][int(step[1])] = 'S'
                                                        print "Putting Safe State at
",int(step[0]),int(step[1])
                                                        print "Visited :",visited
                                                        return step
                                                        print "\n"
                                        else:
                                                for objs in dg:
                                                        if objs in start:
                                                                print "Diagonal",objs,"is Start
state"
                                                        elifobjs == present:
                                                                print "Diagonal",objs,"is Present
state"
                                                        elifobjs in visited:
                                                                tv = 1
                                                                print "%s is diagonal of %s which is
visited"%(objs,st)

                                                        else:
                                                                print objs,"not visited"
                                                else:
                                                        if tv is not 1:
                                                                print "here Nothing could be done
for",st
        print "_____________\n"
        return 0
def whats_nextstep(ws,wp):
        global tl,present,prev,tmp,h
        print "\nThinkingwhats_nextstep from",ws,wp,
```

```
print "\nNow present is :",present,
print "Now previous is :",prev
setdirections_for(ws,wp)
tl = tl[0:0]
tl = allowedsteps(tl)
if ws == gm and wp == gn:
        h = 1
        if h is 1:
                print "\nHurray!!Got the Gold",
        print "\nPresently in %s returning back to %s"%(present,start)
        #print mypath
        for steps in range(len(mypath),0,-1):
                #print "Stepping into ",mypath[steps-1]
                prev = present
                present = mypath[steps-1]
                #print "Present : ",present,"Previous :",prev
        print "Now reached to :",present
        exit(0)
        #return_to_initial_postition()
elif k[ws][wp] is "S":
        print "\nSafe Cell"
        print "Allowed Steps",tl,
        for step in tl:
                tmp = step
                if step not in visited:
                        print step,"Not visited"
                        prev = present
                        print "previous cell :",prev
                        present = step[0]+step[1]
                        print "present cell",present
                        #print "Visited :",visited
                        #Theres something wrong here
                        k[int(step[0])][int(step[1])] = stepintocell(step)
                        print "\nStepping into ",step[0],step[1]
                        mypath.append(step)
                        visited.append(step)
                        print visited
                        print k
                        whats_nextstep(int(step[0]),int(step[1]))
                        Print
                else:
                        print step,"already visited",
else:
        print "\nNot a Safe Cell",
        print "\nConnections of ",present,":",tl,
        if ws is 2 and wp is 2:
                print tl
                #print "exiting"
                #exit(0)
        l = applylogic_from_knowledge()
```

```
                print "i got here",l
                if l is 0:
                        print "Stepping Back to",prev,"\n"
                        mypath.pop()
                        print prev
                        present = prev
                        print present,prev
                        whats_nextstep(int(prev[0]),int(prev[1]))
                else:
                        print "here I reached********************"
                        visited.append(l)
                        print visited
                        prev = tmp
                        present = l
                        print present,prev
                        print "\nfrom here Stepping into ",l,
                        mypath.append(l)
                        whats_nextstep(int(l[0]),int(l[1]))

if __name__ == "__main__":
        #getMatrix("Enter Matrix",4);
        '''
        for i in range(1,5):
                for j in range(1,5):
                        setdirections_for(i,j)
                        print "from",i,j,"it can move to",
                        allowedcells()
        '''
        prm = pm
        prn = pn
        start = str(pm)+str(pn)
        visited.append(start)
        print visited
        print "Starting from ",pm,pn
        prev = str(pm)+str(pn)
        present = str(pm)+str(pn)
        mypath.append(start)
        whats_nextstep(pm,pn)
        print present
        print k
```

14.3 Eight Puzzle Problem Using Python

```
# The state of the board is stored in a list. The list stores values for the
# board in the following positions:
#
# -------------
# | 0 | 3 | 6 |
# -------------
# | 1 | 4 | 7 |
# -------------
# | 2 | 5 | 8 |
# -------------
#
# The goal is defined as:
#
# -------------
# | 1 | 2 | 3 |
# -------------
# | 8 | 0 | 4 |
# -------------
# | 7 | 6 | 5 |
# -------------
#
# Where 0 denotes the blank tile or space.
goal_state= [1, 8, 7, 2, 0, 6, 3, 4, 5]
#
# The code will read state from a file called "state.txt" where the format is
# as above but space seperated. i.e. the content for the goal state would be
# 1 8 7 2 0 6 3 4 5
### Code begins.
import sys
defdisplay_board( state ):
        print"-------------"
        print"| %i | %i | %i |"% (state[0], state[3], state[6])
        print"-------------"
        print"| %i | %i | %i |"% (state[1], state[4], state[7])
        print"-------------"
        print"| %i | %i | %i |"% (state[2], state[5], state[8])
        print"-------------"

defmove_up( state ):
        """Moves the blank tile up on the board. Returns a new state as a list."""
        # Perform an object copy
        new_state= state[:]
        index =new_state.index( 0 )
        # Sanity check
        if index notin [0, 3, 6]:
```

```
                # Swap the values.
                temp =new_state[index -1]
                new_state[index -1] =new_state[index]
                new_state[index] = temp
                returnnew_state
        else:
                # Can't move, return None (Pythons NULL)
                returnNone
defmove_down( state ):
        """Moves the blank tile down on the board. Returns a new state as a list."""
        # Perform object copy
        new_state= state[:]
        index =new_state.index( 0 )
        # Sanity check
        if index notin [2, 5, 8]:
                # Swap the values.
                temp =new_state[index +1]
                new_state[index +1] =new_state[index]
                new_state[index] = temp
                returnnew_state
        else:
                # Can't move, return None.
                returnNone
defmove_left( state ):
        """Moves the blank tile left on the board. Returns a new state as a list."""
        new_state= state[:]
        index =new_state.index( 0 )
        # Sanity check
        if index notin [0, 1, 2]:
                # Swap the values.
                temp =new_state[index -3]
                new_state[index -3] =new_state[index]
                new_state[index] = temp
                returnnew_state
        else:
                # Can't move it, return None
                returnNone
defmove_right( state ):
        """Moves the blank tile right on the board. Returns a new state as a list."""
        # Performs an object copy. Python passes by reference.
        new_state= state[:]
        index =new_state.index( 0 )
        # Sanity check
        if index notin [6, 7, 8]:
                # Swap the values.
                temp =new_state[index +3]
                new_state[index +3] =new_state[index]
                new_state[index] = temp
                returnnew_state
        else:
```

```
                # Can't move, return None
                returnNone
defcreate_node( state, parent, operator, depth, cost ):
        return Node( state, parent, operator, depth, cost )
defexpand_node( node, nodes ):
        """Returns a list of expanded nodes"""
        expanded_nodes= []
        expanded_nodes.append( create_node( move_up( node.state ), node, "u",
node.depth+1, 0 ) )
        expanded_nodes.append( create_node( move_down( node.state ), node, "d",
node.depth+1, 0 ) )
        expanded_nodes.append( create_node( move_left( node.state ), node, "l",
node.depth+1, 0 ) )
        expanded_nodes.append( create_node( move_right( node.state), node, "r",
node.depth+1, 0 ) )
        # Filter the list and remove the nodes that are impossible (move function
returned None)
        expanded_nodes= [node for node inexpanded_nodesifnode.state!=None]
#list comprehension!
        returnexpanded_nodes
defbfs( start, goal ):
        """Performs a breadth first search from the start state to the goal"""
        # A list (can act as a queue) for the nodes.
        nodes = []
        # Create the queue with the root node in it.
        nodes.append( create_node( start, None, None, 0, 0 ) )
        whileTrue:
                # We've run out of states, no solution.
                iflen( nodes ) ==0: returnNone
                # take the node from the front of the queue
                node =nodes.pop(0)
                # Append the move we made to moves
                # if this node is the goal, return the moves it took to get here.
                ifnode.state== goal:
                        moves = []
                        temp = node
                        whileTrue:
                                moves.insert(0, temp.operator)
                                iftemp.depth==1: break
                                temp =temp.parent
                        return moves
                # Expand the node and add all the expansions to the front of the
stack
                nodes.extend( expand_node( node, nodes ) )
defdfs( start, goal, depth=10 ):
        """Performs a depth first search from the start state to the goal. Depth
param is optional."""
        # NOTE: This is a limited search or else it keeps repeating moves. This is an
infinite search space.
```

```
        # I'm not sure if I implemented this right, but I implemented an iterative depth search below
        # too that uses this function and it works fine. Using this function itself will repeat moves until
        # the depth_limit is reached. Iterative depth search solves this problem, though.
        #
        # An attempt of cutting down on repeat moves was made in the expand_node() function.
        depth_limit= depth
        # A list (can act as a stack too) for the nodes.
        nodes = []
        # Create the queue with the root node in it.
        nodes.append( create_node( start, None, None, 0, 0 ) )
        whileTrue:
                # We've run out of states, no solution.
                iflen( nodes ) ==0: returnNone
                # take the node from the front of the queue
                node =nodes.pop(0)
                # if this node is the goal, return the moves it took to get here.
                ifnode.state== goal:
                        moves = []
                        temp = node
                        whileTrue:
                                moves.insert(0, temp.operator)
                                iftemp.depth<=1: break
                                temp =temp.parent
                        return moves
                # Add all the expansions to the beginning of the stack if we are under the depth limit
                ifnode.depth<depth_limit:
                        expanded_nodes=expand_node( node, nodes )
                        expanded_nodes.extend( nodes )
                        nodes =expanded_nodes
defids( start, goal, depth=50 ):
        """Perfoms an iterative depth first search from the start state to the goal. Depth is optional."""
        foriinrange( depth ):
                result =dfs( start, goal, i )
                if result !=None:
                        return result
defa_star( start, goal ):
        """Perfoms an A* heuristic search"""
        # ATTEMPTED: does not work :(
        nodes = []
        nodes.append( create_node( start, None, None, 0, 0 ) )
        whileTrue:
                # We've run out of states - no solution.
                iflen( nodes ) ==0: returnNone
                # Sort the nodes with custom compare function.
```

```
                nodes.sort( cmp )
                # take the node from the front of the queue
                node =nodes.pop(0)
                # if this node is the goal, return the moves it took to get here.
                print"Trying state", node.state, " and move: ", node.operator
                ifnode.state== goal:
                        moves = []
                        temp = node
                        whileTrue:
                                moves.insert( 0, temp.operator )
                                iftemp.depth<=1: break
                                temp =temp.parent
                        return moves
                #Expand the node and add all expansions to the end of the queue
                nodes.extend( expand_node( node, nodes ) )

defcmp( x, y ):
        # Compare function for A*. f(n) = g(n) + h(n). I use depth (number of moves) for g().
        return (x.depth+ h( x.state, goal_state )) - (y.depth+ h( x.state, goal_state ))
defh( state, goal ):
        """Heuristic for the A* search. Returns an integer based on out of place tiles"""
        score =0
        foriinrange( len( state ) ):
                if state[i] != goal[i]:
                        score = score +1
        return score
# Node data structure
classNode:
        def__init__( self, state, parent, operator, depth, cost ):
                # Contains the state of the node
                self.state= state
                # Contains the node that generated this node
                self.parent= parent
                # Contains the operation that generated this node from the parent
                self.operator= operator
                # Contains the depth of this node (parent.depth +1)
                self.depth= depth
                # Contains the path cost of this node from depth 0. Not used for depth/breadth first.
                self.cost= cost
defreadfile( filename ):
        f =open( filename )
        data =f.read()
        # Get rid of the newlines
        data =data.strip( "\n" )
        #Break the string into a list using a space as a seperator.
        data =data.split( " " )
        state = []
```

```
        for element in data:
                state.append( int( element ) )
        return state
# Main method
defmain():
        starting_state=readfile( "state.txt" )
        ### CHANGE THIS FUNCTION TO USE bfs, dfs, ids or a_star
        result = ids( starting_state, goal_state )
        if result ==None:
                print"No solution found"
        elif result == [None]:
                print"Start node was the goal!"
        else:
                print result
                printlen(result), " moves"
# A python-isim. Basically if the file is being run execute the main() function.
if__name__=="__main__":
        main()
```

14.4 Tic-Tak-Toe Game Using Python

```
defboard():
        print("Welcome to the tic tak toe game. Player 1 will use 'O' as mark. Player 2 will use 'X' as mark.")
           d={}
        for n inrange(1,10):
              d[n]="-"
        return d
        defprintBoard(d):
        fork,vind.items():
        ifint(k)%3==0:
        print(v, end=" ")
        print("\n")
        else:
        print(v, end=" ")
        defcheckInput(d): # take input and check if the input is valid
           x=""
        # try: # check if the input is a number
        #     x=int(x)
        # except ValueError:
        #     print("Enter number please")
        #     continue
        while x notin'1 2 3 4 5 6 7 8 9'.split() ornotcheckSingle(d,int(x)):  # check if the number is in the range of 1 to 9
                  x=input("Please enter correct number(only number from 1-9): ")
        returnint(x)
        defcheckSingle(d,x): # check if the space has been filled
        return d[x] =="-"
        defcheckWon(d,n):
        contorl=0# 0 for ongoing, 1 for tie, 2 for player1 won, 3 for player3 won
        sumnum=0# tie counter
           sv0=0# player1 vertical counter
           sh0=0# player1 horizonal counter
           sd0=0# player1 dignose counter
           sv1=0
           sh1=0
           sd1=0
        if n >3and n<7:
        for e in [n-3,n+3]:
        if d[e] =="O":
                        sv0 +=1
        elif d[e] =="X":
                        sv1 +=1
        else:
        Pass
        for e in [4,5,6]:
        if d[e]=="O":
                        sh0 +=1
        elif d[e] =="X":
                        sh1 +=1
        else:
```

```
Pass
if sh0==3or sv0==3:
        control =2
return control
if sh1==3or sv1==3:
        control =3
return control
elif n<4:
for e in [n+3,n+6]:
if d[e] =="O":
            sv0 +=1
elif d[e] =="X":
            sv1 +=1
else:
Pass
for e in [1,2,3]:
if d[e]=="O":
            sh0 +=1
elif d[e] =="X":
            sh1 +=1
else:
Pass
if n==1:
for e in [1,5,9]:
if d[e]=="O":
                sd0+=1
elif d[e] =="X":
                sd1+=1
else:
Pass
if n==3:
for e in [3,5,7]:
if d[e]=="O":
                sd0+=1
elif d[e] =="X":
                sd1+=1
else:
Pass
if sh0==3or sv0==3or sd0==3:
        control =2
return control
if sh1==3or sv1==3or sd1==3:
        control =3
return control
elif n>6:
for e in [n-3,n-6]:
if d[e] =="O":
            sv0 +=1
elif d[e] =="X":
            sv1 +=1
```

```
else:
Pass
for e in [7,8,9]:
if d[e]=="O":
            sh0 +=1
elif d[e] =="X":
            sh1 +=1
else:
Pass
if n==7:
for e in [3,5,7]:
if d[e]=="O":
                sd0+=1
elif d[e] =="X":
                sd1+=1
else:
Pass
if n==9:
for e in [1,5,9]:
if d[e]=="O":
                sd0+=1
elif d[e] =="X":
                sd1+=1
else:
Pass
if sh0==3or sv0==3or sd0==3:
          control =2
return control
if sh1==3or sv1==3or sd1==3:
          control =3
return control
else:
Pass
# for e in d.values():
#     if e !="-":
#         sumnum+=1
if"-"notind.values():
       control =1
# if sumnum ==9:
#     control = 1
return control
defprocess(d,player):
   won =0
   x=checkInput(d)
print("Player",(player+1),"selected position",x)
if player ==0:
       d[x]="O"
       player =1
       won =checkWon(d,x)
if won ==1:
```

```
print("tie")
elif won ==2:
print("Player1 won")
elif won ==3:
print("Player2 won")
else:
printBoard(d)
        process(d,player)
else:
    d[x]="X"
    player =0
    won =checkWon(d,x)
if won ==1:
print("tie")
elif won ==2:
print("Player1 won")
elif won ==3:
print("Player2 won")
else:
printBoard(d)
        process(d,player)
defreplay():
   x =input("Do you want to play again(y/n): ").lower().startswith("y")
if  x ==True:
    player =0
    main()
else:
print("Bye!")
Pass
defmain():
   player =0# set the player switch
   d = board() # init the board
printBoard(d) # print the blank board
   process(d,player) # game on
printBoard(d) # print the final board
   replay() # ask if the players wnat to play next round
main()
```

14.5 Eight / N- Queen Problem Using Python

14.5.1 Eight Queen Problem

```
fromitertoolsimportpermutations

n=8
cols=range(n)
forvecinpermutations(cols):
if(n==len(set(vec[i]+iforiincols))
==len(set(vec[i]-iforiincols))):
printvec
```

Computes all 92 solutions for eight queens. By setting n to different values, other sized puzzles can be solved.

The output is presented in vector form (each number represents the column position of a queen on consecutive rows). The vector can be pretty printed with this function:

```
def board(vec):
    '''Translate column positions to an equivalent chess board.

    >>> board([0, 4, 7, 5, 2, 6, 1, 3])
    Q-------
    ----Q---
    -------Q
    -----Q--
    --Q-----
    ------Q-
    -Q------
    ---Q----

    '''

    for col invec:
        s =['-']*len(vec)
        s[col]='Q'
        print''.join(s)
    print
```

14.5.2 N-Queen Problem

```
N = 8                    # Default; command line overrides
```

```
class Queens:

    def __init__(self, n=N):
self.n = n
self.reset()

    def reset(self):
        n = self.n
self.y = [None] * n             # Where is the queen in column x
self.row = [0] * n              # Is row[y] safe?
self.up = [0] * (2*n-1)         # Is upward diagonal[x-y] safe?
self.down = [0] * (2*n-1)       # Is downward diagonal[x+y] safe?
self.nfound = 0                 # Instrumentation

    def solve(self, x=0):               # Recursive solver
        for y in range(self.n):
            if self.safe(x, y):
self.place(x, y)
                if x+1 == self.n:
self.display()
                else:
self.solve(x+1)
self.remove(x, y)

    def safe(self, x, y):
        return not self.row[y] and not self.up[x-y] and not self.down[x+y]

    def place(self, x, y):
self.y[x] = y
self.row[y] = 1
self.up[x-y] = 1
self.down[x+y] = 1

    def remove(self, x, y):
self.y[x] = None
self.row[y] = 0
self.up[x-y] = 0
self.down[x+y] = 0

    silent = 0                      # If true, count solutions only

    def display(self):
self.nfound = self.nfound + 1
        if self.silent:
            return
        print '+-' + '--'*self.n + '+'
        for y in range(self.n-1, -1, -1):
            print '|',
            for x in range(self.n):
```

```
            if self.y[x] == y:
                print "Q",
            else:
                print ".",
        print '|'
    print '+-' + '--'*self.n + '+'

def main():
    import sys
    silent = 0
    n = N
    if sys.argv[1:2] == ['-n']:
        silent = 1
        del sys.argv[1]
    if sys.argv[1:]:
        n = int(sys.argv[1])
    q = Queens(n)
q.silent = silent
q.solve()
    print "Found", q.nfound, "solutions."

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

14.6 Minimax & Alpha Beta Pruning AI Algorithm in Tic-Tac-Toe Using Python

```
# Minimax Search
```

```
def minimax_decision(state, game):
"""Given a state in a game, calculate the best move by searching
   forward all the way to the terminal states. [Fig. 6.4]"""

   player = game.to_move(state)

def max_value(state):
       if game.terminal_test(state):
          return game.utility(state, player)
       v = -infinity
       for (a, s) in game.successors(state):
          v = max(v, min_value(s))
       return v

def min_value(state):
       if game.terminal_test(state):
          return game.utility(state, player)
       v = infinity
       for (a, s) in game.successors(state):
          v = min(v, max_value(s))
       return v

# Body of minimax_decision starts here:
   action, state = argmax(game.successors(state),
                          lambda ((a, s)): min_value(s))
   return action
```

```
def alphabeta_full_search(state, game):
"""Search game to determine best action; use alpha-beta pruning.
   As in [Fig. 6.7], this version searches all the way to the leaves."""

   player = game.to_move(state)

def max_value(state, alpha, beta):
       if game.terminal_test(state):
          return game.utility(state, player)
       v = -infinity
       for (a, s) in game.successors(state):
          v = max(v, min_value(s, alpha, beta))
          if v >= beta:
             return v
          alpha = max(alpha, v)
       return v

def min_value(state, alpha, beta):
       if game.terminal_test(state):
          return game.utility(state, player)
```

```
        v = infinity
        for (a, s) in game.successors(state):
            v = min(v, max_value(s, alpha, beta))
            if v <= alpha:
                return v
            beta = min(beta, v)
        return v

# Body of alphabeta_search starts here:
    action, state = argmax(game.successors(state),
                           lambda ((a, s)): min_value(s, -infinity, infinity))
    return action

def alphabeta_search(state, game, d=4, cutoff_test=None, eval_fn=None):
"""Search game to determine best action; use alpha-beta pruning.
    This version cuts off search and uses an evaluation function."""

    player = game.to_move(state)

def max_value(state, alpha, beta, depth):
        if cutoff_test(state, depth):
            return eval_fn(state)
        v = -infinity
        for (a, s) in game.successors(state):
            v = max(v, min_value(s, alpha, beta, depth+1))
            if v >= beta:
                return v
            alpha = max(alpha, v)
        return v

def min_value(state, alpha, beta, depth):
        if cutoff_test(state, depth):
            return eval_fn(state)
        v = infinity
        for (a, s) in game.successors(state):
            v = min(v, max_value(s, alpha, beta, depth+1))
            if v <= alpha:
                return v
            beta = min(beta, v)
        return v

# Body of alphabeta_search starts here:
# The default test cuts off at depth d or at a terminal state
cutoff_test = (cutoff_test or
                (lambda state,depth: depth>d or game.terminal_test(state)))
eval_fn = eval_fn or (lambda state: game.utility(state, player))
    action, state = argmax(game.successors(state),
                           lambda ((a, s)): min_value(s, -infinity, infinity, 0))
    return action
```

```
# Players for Games

def query_player(game, state):
"Make a move by querying standard input."
game.display(state)
    return num_or_str(raw_input('Your move? '))

def random_player(game, state):
"A player that chooses a legal move at random."
    return random.choice(game.legal_moves())

def alphabeta_player(game, state):
    return alphabeta_search(state, game)

def play_game(game, *players):
"Play an n-person, move-alternating game."
    state = game.initial
    while True:
        for player in players:
            move = player(game, state)
            state = game.make_move(move, state)
            if game.terminal_test(state):
                return game.utility(state, players[0])
```

```
# Some Sample Games

class Game:
"""A game is similar to a problem, but it has a utility for each
    state and a terminal test instead of a path cost and a goal
    test. To create a game, subclass this class and implement
legal_moves, make_move, utility, and terminal_test. You may
    override display and successors or you can inherit their default
    methods. You will also need to set the .initial attribute to the
    initial state; this can be done in the constructor."""

def legal_moves(self, state):
"Return a list of the allowable moves at this point."
        abstract

def make_move(self, move, state):
"Return the state that results from making a move from a state."
        abstract

def utility(self, state, player):
"Return the value of this final state to player."
        abstract

def terminal_test(self, state):
```

```
"Return True if this is a final state for the game."
        return not self.legal_moves(state)

def to_move(self, state):
"Return the player whose move it is in this state."
        return state.to_move

def display(self, state):
"Print or otherwise display the state."
        print state

def successors(self, state):
"Return a list of legal (move, state) pairs."
        return [(move, self.make_move(move, state))
                for move in self.legal_moves(state)]

def __repr__(self):
        return '<%s>' % self.__class__.__name__

class Fig62Game(Game):
"""The game represented in [Fig. 6.2]. Serves as a simple test case.
>>> g = Fig62Game()
>>>minimax_decision('A', g)
    'a1'
>>>alphabeta_full_search('A', g)
    'a1'
>>>alphabeta_search('A', g)
    'a1'
    """
succs = {'A': [('a1', 'B'), ('a2', 'C'), ('a3', 'D')],
'B': [('b1', 'B1'), ('b2', 'B2'), ('b3', 'B3')],
'C': [('c1', 'C1'), ('c2', 'C2'), ('c3', 'C3')],
'D': [('d1', 'D1'), ('d2', 'D2'), ('d3', 'D3')]}
utils = Dict(B1=3, B2=12, B3=8, C1=2, C2=4, C3=6, D1=14, D2=5, D3=2)
    initial = 'A'

def successors(self, state):
        return self.succs.get(state, [])

def utility(self, state, player):
        if player == 'MAX':
            return self.utils[state]
        else:
            return -self.utils[state]

def terminal_test(self, state):
        return state not in ('A', 'B', 'C', 'D')

def to_move(self, state):
        return if_(state in 'BCD', 'MIN', 'MAX')
```

```
class TicTacToe(Game):
"""Play TicTacToe on an h x v board, with Max (first player) playing 'X'.
   A state has the player to move, a cached utility, a list of moves in
   the form of a list of (x, y) positions, and a board, in the form of
   a dict of {(x, y): Player} entries, where Player is 'X' or 'O'."""
def __init__(self, h=3, v=3, k=3):
        update(self, h=h, v=v, k=k)
        moves = [(x, y) for x in range(1, h+1)
                 for y in range(1, v+1)]
self.initial = Struct(to_move='X', utility=0, board={}, moves=moves)

def legal_moves(self, state):
"Legal moves are any square not yet taken."
        return state.moves

def make_move(self, move, state):
        if move not in state.moves:
            return state # Illegal move has no effect
        board = state.board.copy(); board[move] = state.to_move
        moves = list(state.moves); moves.remove(move)
        return Struct(to_move=if_(state.to_move == 'X', 'O', 'X'),
                      utility=self.compute_utility(board, move, state.to_move),
                      board=board, moves=moves)

def utility(self, state):
"Return the value to X; 1 for win, -1 for loss, 0 otherwise."
        return state.utility

def terminal_test(self, state):
"A state is terminal if it is won or there are no empty squares."
        return state.utility != 0 or len(state.moves) == 0

def display(self, state):
        board = state.board
        for x in range(1, self.h+1):
            for y in range(1, self.v+1):
                print board.get((x, y), '.'),
            print

def compute_utility(self, board, move, player):
"If X wins with this move, return 1; if O return -1; else return 0."
        if (self.k_in_row(board, move, player, (0, 1)) or
self.k_in_row(board, move, player, (1, 0)) or
self.k_in_row(board, move, player, (1, -1)) or
self.k_in_row(board, move, player, (1, 1))):
            return if_(player == 'X', +1, -1)
        else:
            return 0
```

```
def k_in_row(self, board, move, player, (delta_x, delta_y)):
"Return true if there is a line through move on board for player."
        x, y = move
        n = 0 # n is number of moves in row
        while board.get((x, y)) == player:
            n += 1
            x, y = x + delta_x, y + delta_y
        x, y = move
        while board.get((x, y)) == player:
            n += 1
            x, y = x - delta_x, y - delta_y
        n -= 1 # Because we counted move itself twice
        return n>= self.k

class ConnectFour(TicTacToe):
"""A TicTacToe-like game in which you can only make a move on the bottom
    row, or in a square directly above an occupied square.  Traditionally
    played on a 7x6 board and requiring 4 in a row."""

def __init__(self, h=7, v=6, k=4):
TicTacToe.__init__(self, h, v, k)

def legal_moves(self, state):
"Legal moves are any square not yet taken."
        return [(x, y) for (x, y) in state.moves
                if y == 0 or (x, y-1) in state.board]
```

14.7 Constraint Satisfaction Problem Using Python

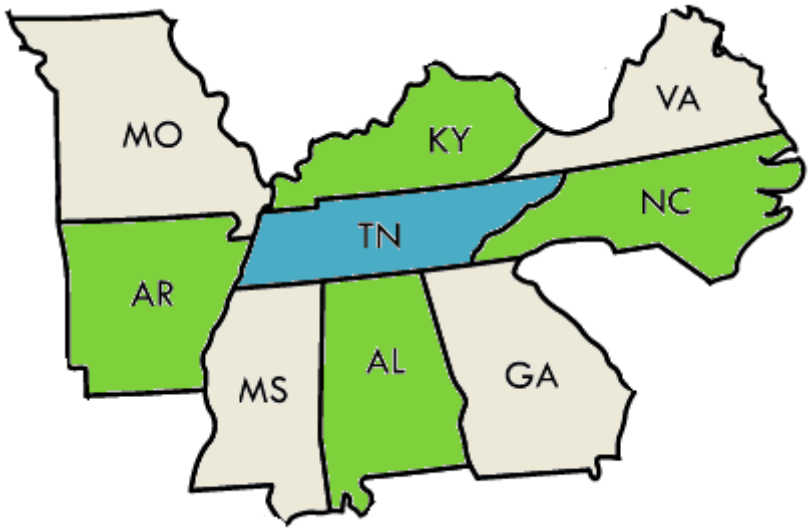
14.7.1 Crypt Arithmetic Problem Using Python

```
        from re import sub
        defsolve(q):
```

```
try:
    n = (iforiin q ifi.isalpha()).next()
exceptStopIteration:
return q ifeval(sub(r'(^|[^0-9])0+([1-9]+)', r'\1\2', q)) elseFalse
else:
foriin (str(i) foriinrange(10) ifstr(i) notin q):
      r = solve(q.replace(n,str(i)))
if r:
return r
returnFalse
if__name__=="__main__":
  query ="ABCDE * A == EEEEEE"
  r = solve(query)
print r if r else"No solution found."
# Other puzzles to try:
# query = "REASON == IT * IS + THERE"
# query = "MAD * MAN == ASYLUM"
# query = "THREE + THREE + ONE == SEVEN"
# query = "SEND + MORE == MONEY"
# query = "I + BB == ILL"
# query = "WHOSE + TEETH + ARE + AS == SWORDS"
# query = "BILL + WILLIAM + MONICA == CLINTON"
# query = "GREEN + ORANGE == COLORS"
# query = "PACIFIC + PACIFIC + PACIFIC == ATLANTIC"
# query = "CASSATT + RENOIR == PICASSO"
# query = "MANET + MATISSE + MIRO + MONET + RENOIR == ARTISTS"
# query = "COMPLEX + LAPLACE == CALCULUS"
```

14.7.2 Map Coloring Problem Using Python

Map to be filled :



We start off with a file containing a list of states and the set of states that are adjacent to each. The row for Tennessee might be as follows:

```
1    TN,AL;AR;GA;KY;MO;MS;NC;VA
```

Next we need to read that file.

```
1    def loadData(localFileName):
2        # expects: AA,BB;CC;DD where BB, CC and DD are the initial column values in other rows
3        with open(localFileName, mode='r') as infile:
4            reader = csv.reader(infile)
5            mydict = {row[0]: row[1].split(';') for row in reader if row}
6            return mydict
```

Then we need to build the rules. A Rule connects two states indicating that they are adjacent. We want to be able to put rules in a dictionary and find them in a list so we need to define __hash__ and __eq__. We might also want to be able to display a rule so we'll add a __str__ implementation.

```
1      class Rule:
2          Item = None
3          Other = None
4          Stringified = None
5
6          def __init__(self, item, other, stringified):
7              self.Item = item
8              self.Other = other
9              self.Stringified = stringified
10
11         def __eq__(self, another):
12             return hasattr(another, 'Item') and \
13                     hasattr(another, 'Other') and \
14                     self.Item == another.Item and \
```

```
15                 self.Other == another.Other
16
17         def __hash__(self):
18             return hash(self.Item) * 397 ^ hash(self.Other)
19
20         def __str__(self):
21             return self.Stringified
```

Next we're going to build the set of rules. While we're doing so we're going to perform a sanity check on the data. Whenever a state says it is adjacent to another state, the adjacent state should also say it is adjacent to the first state.

```
1       def buildLookup(items):
2           itemToIndex = {}
3           index = 0
4           for key in sorted(items):
5               itemToIndex[key] = index
6               index += 1
7           return itemToIndex
8
9       def buildRules(items):
10          itemToIndex = buildLookup(items.keys())
11          rulesAdded = {}
12          rules = []
13          keys = sorted(list(items.keys()))
14
15          for key in sorted(items.keys()):
16              keyIndex = itemToIndex[key]
17              adjacentKeys = items[key]
18              for adjacentKey in adjacentKeys:
19                  if adjacentKey == '':
20                      continue
21                  adjacentIndex = itemToIndex[adjacentKey]
22                  temp = keyIndex
23                  if adjacentIndex< temp:                 temp, adjacentIndex = adjacentIndex, temp           ruleKey =
24                  rule = Rule(temp, adjacentIndex, ruleKey)
25                  if rule in rulesAdded:
26                      rulesAdded[rule] += 1
27                  else:
28                      rulesAdded[rule] = 1
29                      rules.append(rule)
30
31          for k, v in rulesAdded.items():
32              if v == 1:
33                  print("rule %s is not bidirectional" % k)
34
35          return rules
```


Now we have the ability to convert a file of node relationships to a set of adjacency rules. Next we need to build the code used by the genetic solver. We'll start by determining what our genes

will be. In this case since we want to four-color the 50 states our gene set will be four color codes.

```
colors = ["Orange", "Yellow", "Green", "Blue"]
colorLookup = {}
for color in colors:
    colorLookup[color[0]] = color
geneset = list(colorLookup.keys())
```

Our Individuals will have 50 genes, one for each state, in alphabetical order. This lets us use the index into the genes as an index into the set of sorted state codes.

Since the expected optimal situation will be that all adjacent states have different colors we can set the optimal value to the number of rules.

At the end we'll write out the color of each state.

```
class GraphColoringTests(unittest.TestCase):
    def test(self):
        states = loadData("adjacent_states.csv")
        rules = buildRules(states)
        colors = ["Orange", "Yellow", "Green", "Blue"]
        colorLookup = {}
        for color in colors:
            colorLookup[color[0]] = color
        geneset = list(colorLookup.keys())
        optimalValue = len(rules)
        startTime = datetime.datetime.now()
        fnDisplay = lambda candidate: display(candidate, startTime)
        fnGetFitness = lambda candidate: getFitness(candidate, rules)
        best = genetic.getBest(fnGetFitness, fnDisplay, len(states), optimalValue, geneset)
        self.assertEqual(best.Fitness, optimalValue)

        keys = sorted(states.keys())

        for index in range(len(states)):
            print(keys[index] + " is " + colorLookup[best.Genes[index]])
```

As for display, it should be sufficient to output the color codes.

```
def display(candidate, startTime):
    timeDiff = datetime.datetime.now() - startTime
    print("%s\t%i\t%s" % (''.join(map(str, candidate.Genes)), candidate.Fitness, str(timeDiff)))
```

This gets output like the following. The number to the right of the gene sequence will indicate how many rules this gene sequence satisfies.

```
YGGBOOGOOBBYGGYYYYGBGYOOGBOYGGOOOYBOYBBGGOBYOGOGOGG	74	0:00:00.001000
```

Finally we need a fitness function that checks all the rules assuming the states are colored according to the gene sequence.

```
def getFitness(candidate, rules):
    rulesThatPass = 0
```

```
3       for rule in rules:
4           if candidate[rule.Item] != candidate[rule.Other]:
5               rulesThatPass += 1
6
7       return rulesThatPass
```

That's it. Now when we run our main test function we get the following output:

```
1       OOYYOBBGYOOYGBBYOOOBOGYGYGGBBYGOGGOYOYGYBBOBOBGOBBG 82  0:00:0
2       YYBOYGGGGOBYOYBGBOOBOOBBGGBGGYGBBGOBOBYYOGYYBBYOYGO
3       0:00:00.016001
4       BOOGOGGOGBGYGGBGOOYBYOBYGBBOGBGBBBOYYYGYYOYOOGYBOBY
5       0:00:00.316018
6       GOBOGGOYGBGOBGOGYBYBOOYYGGBBGOYBYYYOOBGYYOYGBGGOGYY
7       0:00:01.602092
8       BBBBGGBYGOGYBGOGBBYGOGYYYGYBBOOBYYYOOOGOYGOGOGBBGYB
9       0:00:04.933282
10      AK is Blue
11      AL is Blue
12      AR is Blue
13      AZ is Blue
14      CA is Green
15      CO is Green
16      CT is Blue
17      DC is Yellow
18      DE is Green
19      FL is Orange
20      GA is Green
21      HI is Yellow
22      IA is Blue
23      ID is Green
24      IL is Orange
25      IN is Green
26      KS is Blue
27      KY is Blue
28      LA is Yellow
29      MA is Green
30      MD is Orange
31      ME is Green
32      MI is Yellow
33      MN is Yellow
34      MO is Yellow
35      MS is Green
36      MT is Yellow
37      NC is Blue
38      ND is Blue
39      NE is Orange
40      NH is Orange
41      NJ is Blue
42      NM is Yellow
43      NV is Yellow
44      NY is Yellow
```

45 OH is Orange
46 OK is Orange
47 OR is Orange
48 PA is Green
49 RI is Orange
50 SC is Yellow
51 SD is Green
52 TN is Orange
53 TX is Green
54 UT is Orange
55 VA is Green
56 VT is Blue
WA is Blue
WI is Green
WV is Yellow
WY is Blue

Experiment No. 15

Construction of a Domain-Specific Chatbot Using Natural Language Processing Techniques

Aim: Construction of a domain-specific Chatbot using natural language processing techniques. (Applications can include medical diagnosis, personal shopping assistant, travel agent, troubleshooting, etc.).

Theory: Chatbots, or conversational interfaces as they are also known, present a new way for individuals to interact with computer systems. Traditionally, to get a question answered by a software program involved using a search engine or filling out a form. A chatbot allows a user to simply ask questions in the same manner that they would address a human. The most well-known chatbots currently are voice chatbots: Alexa and Siri. However, chatbots are currently being adopted at a high rate on computer chat platforms.

The technology at the core of the rise of the chatbot is natural language processing ("NLP"). Recent advances in machine learning have greatly improved the accuracy and effectiveness of natural language processing, making chatbots a viable option for many organizations. This improvement in NLP is firing a great deal of additional research which should lead to continued improvement in the effectiveness of chatbots in the years to come.

A simple chatbot can be created by loading an FAQ (frequently asked questions) into chatbot software. The functionality of the chatbot can be improved by integrating it into the organization's enterprise software, allowing more personal questions to be answered, like "What is my balance?", or "What is the status of my order?".

Most commercial chatbots are dependent on platforms created by the technology giants for their natural language processing. These include Amazon Lex, Microsoft Cognitive Services, Google Cloud Natural Language API, Facebook DeepText, and IBM Watson. Platforms where chatbots are deployed include Facebook Messenger, Skype, and Slack, among many others.

Applications: A chatbot can be used anywhere a human is interacting with a computer system. These are the areas where the fastest adoption is occurring:

Customer Service: A chatbot can be used as an "assistant" to a live agent, increasing the agent's efficiency. When trained, they can also provide service when the call centre is closed, or eventually even act as an independent agent, if desired.

Sales/Marketing/Branding: Chatbots can be used for sales qualification, ecommerce, promotional campaigns, or as a branding vehicle.

Human Resources: An HR chatbot can help with frequently asked questions ("how many vacation days do I have left?") and can act as an onboarding assistant.

Benefits

1. Economically offer 24/7 Service
2. Improve Customer Satisfaction
3. Reach a Younger Demographic
4. Reduce Costs
5. Increase Revenue

Installing Chatterbot

You can install Chatterbot on your system using Python's pip command. Pip install chatterbot
Creating your first chat bot
Create a new file named *chatbot.py*. Then open *chatbot.py* in your editor of choice.
Before we do anything else, Chatterbot needs to be imported. The import for Chatterbot should look like the following line.
From chatterbot import Chatbot
Create a new instance of the Chatbot class.
Bot = Chatbot('Norman')
This line of code has created a new chat bot named *Norman*. There is a few more parameters that we will want to specify before we run our program for the first time.

Setting the storage adapter

Chatterbot comes with built in adapter classes that allow it to connect to different types of databases. In this tutorial, we will be using the SQLStorageAdapter which allows the chat bot to connect to SQL databases. By default, this adapter will create a SQLite database.

The database parameter is used to specify the path to the database that the chat bot will use. For this example, we will call the database *sqlite:///database.sqlite3*. this file will be created automatically if it does not already exist.

```
Bot = Chatbot(
'Norman',
storage_adapter='chatterbot.storage.SQLStorageAdapter',
database_uri='sqlite:///database.sqlite3'
)
```

Note
The SQLStorageAdapter is ChatterBot's default adapter. If you do not specify an adapter in your constructor, the SQLStorageAdapter adapter will be used automatically.
Specifying logic adapters
The *logic_adapters* parameter is a list of logic adapters. In Chatterbot, a logic adapter is a class that takes an input statement and returns a response to that statement.
You can choose to use as many logic adapters as you would like. In this example we will use two logic adapters. The TimeLogicAdapter returns the current time when the input statement asks for it. The Mathematical Evaluation adapter solves math problems that use basic operations.

```
Bot = Chatbot(
'Norman',
storage_adapter='chatterbot.storage.SQLStorageAdapter',
logic_adapters=[
'chatterbot.logic.MathematicalEvaluation',
'chatterbot.logic.TimeLogicAdapter'
   ],
database_uri='sqlite:///database.sqlite3'
)
```

Getting a Response from Your Chat Bot: Next, you will want to create a while loop for your chat bot to run in. By breaking out of the loop when specific exceptions are triggered, we can exit the loop and stop the program when a user enters *ctrl+c*.

```
while True:
   try:
bot_input = bot.get_response(input())
      print(bot_input)
```

```
except(KeyboardInterrupt, EOFError, SystemExit):
    break
```

Training Your Chatbot: At this point your chat bot, Norman will learn to communicate as you talk to him. You can speed up this process by training him with examples of existing conversations.

```
From chatterbot.trainers import ListTrainer

trainer = ListTrainer(bot)

trainer.train([
'How are you?',
'I am good.',
'That is good to hear.',
'Thank you',
'You are welcome.',
])
```

You can run the training process multiple times to reinforce preferred responses to particular input statements. You can also run the train command on a number of different example dialogs to increase the breadth of inputs that your chat bot can respond to.

```
From chatterbot import Chatbot
from chatterbot.trainers import ListTrainer

# Create a new chat bot named Charlie
chatbot = ChatBot('Charlie')

trainer = ListTrainer(chatbot)

trainer.train([
"Hi, can I help you?",
"Sure, I'd like to book a flight to Iceland.",
"Your flight has been booked."
])

# Get a response to the input text 'I would like to book a flight.'
Response = chatbot.get_response('I would like to book a flight.')

print(response)
```

Terminal Example: This example program shows how to create a simple terminal client that allows you to communicate with your chat bot by typing into your terminal.

```
From chatterbot import Chatbot

# Uncomment the following lines to enable verbose logging
# import logging
# logging.basicConfig(level=logging.INFO)
```

```
# Create a new instance of a ChatBot
bot = ChatBot(
‘Terminal’,
storage_adapter=’chatterbot.storage.SQLStorageAdapter’,
logic_adapters=[
‘chatterbot.logic.MathematicalEvaluation’,
‘chatterbot.logic.TimeLogicAdapter’,
‘chatterbot.logic.BestMatch’
    ],
database_uri=’sqlite:///database.db’
)

print(‘Type something to begin’…')

# The following loop will execute each time the user enters input
while True:
    try:
user_input = input()

bot_response = bot.get_response(user_input)

        print(bot_response)

    # Press ctrl-c or ctrl-d on the keyboard to exit
    except (KeyboardInterrupt, EOFError, SystemExit):
        break
```

Using MongoDB: Before you can use ChatterBot’s built in adapter for MongoDB, you will need to install MongoDB. Make sure MongoDB is running in your environment before you execute your program. To tell Chatterbot to use this adapter, you will need to set the *storage_adapter* parameter.

```
storage_adapter="chatterbot.storage.MongoDatabaseAdapter"
from chatterbot import ChatBot

# Uncomment the following lines to enable verbose logging
# import logging
# logging.basicConfig(level=logging.INFO)

# Create a new ChatBot instance
bot = ChatBot(
  'Terminal',
storage_adapter='chatterbot.storage.MongoDatabaseAdapter',
logic_adapters=[
        'chatterbot.logic.BestMatch'
    ],
database_uri='mongodb://localhost:27017/chatterbot-database'
)

print('Type something to begin...')
```

```
while True:
    try:
user_input = input()

bot_response = bot.get_response(user_input)

        print(bot_response)

    # Press ctrl-c or ctrl-d on the keyboard to exit
    except (KeyboardInterrupt, EOFError, SystemExit):
        break
```

Time and Mathematics Example: Chatterbot has natural language evaluation capabilities that allow it to process and evaluate mathematical and time-based inputs.

```
from chatterbot import ChatBot

bot = ChatBot(
  'Math & Time Bot',
logic_adapters=[
        'chatterbot.logic.MathematicalEvaluation',
        'chatterbot.logic.TimeLogicAdapter'
    ]
)

# Print an example of getting one math based response
response = bot.get_response('What is 4 +’9?')
print(response)

# Print an example of getting one time based response
response = bot.get_response('What time is it?')
print(response)
```

Using SQL Adapter: Chatterbot data can be saved and retrieved from SQL databases. From chatterbot, import ChatBot

```
# Uncomment the following lines to enable verbose logging
# import logging
# logging.basicConfig(level=logging.INFO)

# Create a new instance of a ChatBot
bot = ChatBot(
   'SQLMemoryTerminal',
storage_adapter='chatterbot.storage.SQLStorageAdapter',
database_uri=None,
logic_adapters=[
        'chatterbot.logic.MathematicalEvaluation',
        'chatterbot.logic.TimeLogicAdapter',
        'chatterbot.logic.BestMatch'
    ]
```

)

Get a few responses from the bot

bot.get_response('What time is it?')

bot.get_response('What is 7 plus 7?')

Conclusion: Your chat bot will learn based on each new input statement it receives. If you want to disable this learning feature after your bot has been trained, you can set read-only=True as a parameter when initializing the bot.
chatbot = ChatBot("Johnny Five", read_only=True)