INDEX[1]		
S.No	Name of Experiment	Page No.
1	Write a program to implement Uninformed search techniques: a. BFS b. DFS	
2	Write a program to implement Informed search techniques a. Greedy Best first search b. A* algorithm	
3	a.Write a program to implement Factorial, Fibonacci of a given number. b.Write a program for Family-tree.	
4	a. Write a program to train and validate the Decision Tree classifiers given data using python. b.Write a pogram for data preprocessing using python	
5	a.Write a program to solve the Monkey Banana problem using prolog predicates. b.Write a program to solve water jug problem using prolog predicates.	
6	Write a program to Solve 7-puzzle problem using Best First Search.	
7	Text processing using NLTK a. Remove stop words b. Implement stemming c. POS (Parts of Speech) tagging	
8	Write a program to train and validate, Multi-layer Feed Forward neural network	

1

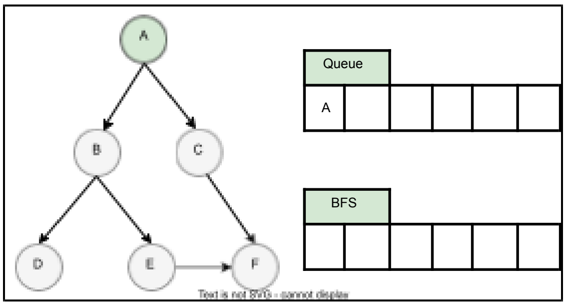
week-1
Write a program to implement Uninformed search techniques:
a. BFS   b. DFS

1.a.Aim: Write a program to impliment Breadth First Search technique in Python.

Procedure:

```
BFS (graph, startNode):
   create a queue Q
   mark startNode as visited and enqueue startNode into Q
   while Q is not empty:
      node = Q.dequeue()
      print(node)
      for each neighbor of node:
         if neighbor is not visited:
            mark neighbor as visited
            enqueue neighbor into Q
      for each neighbor of node:
         if neighbor is not visited:
            mark neighbor as visited
            enqueue neighbor into Q
```

Input graph:



Program:

```
graph = {
  'A': ['B','C'],
  'B': ['D', 'E'],
  'C': ['F'],
  'D': [],
  'E': ['F'],
  'F': []
}
visited = [] # List to keep track of visited nodes.
queue = []        #Initialize a queue
def bfs(visited, graph, node):
  visited.append(node)
  queue.append(node)
  while queue:
    s = queue.pop(0)
    print (s, end = " ")
    for neighbour in graph[s]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

# Driver Code
 bfs(visited, graph, 'A')
```

Output:

A B C D E F

1.b.Aim: Write a program to impliment Depth First Search technique in Python.

Procedure:

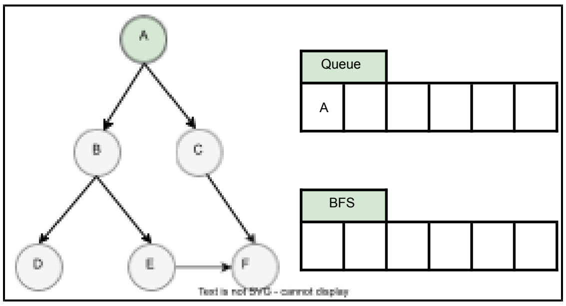
A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.
The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

Input graph:



Output:
A
B
D
E
F
C

Program:

```
# Using a Python dictionary to act as an adjacency list
graph = {
  'A' : ['B','C'],
  'B' : ['D', 'E'],
  'C' : ['F'],
  'D' : [],
  'E' : ['F'],
  'F' : []
}

visited = set() # Set to keep track of visited nodes.
def dfs(visited, graph, node):
  if node not in visited:
    print (node)
    visited.add(node)
    for neighbour in graph[node]:
      dfs(visited, graph, neighbour)

dfs(visited, graph, 'A')
```

week-2

Write a program to implement Informed search techniques

- Greedy Best first search
- A* algorithm

2.a.Aim:Write a program to implement Greedy Best first search in python.

Procedure:

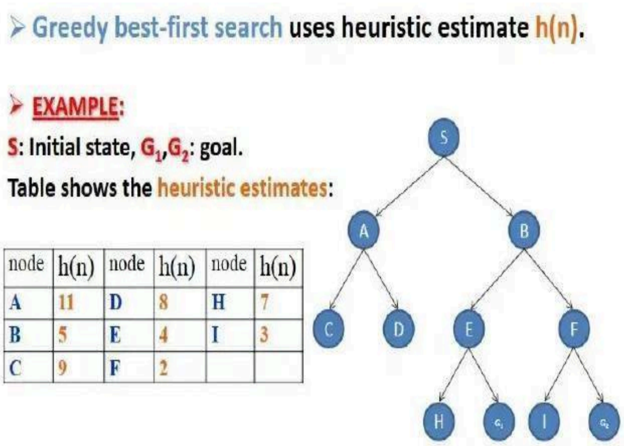
Greedy Best-First Search is an AI search algorithm that attempts to find the most promising path from a given starting point to a goal. It prioritizes paths that appear to be the most promising, regardless of whether or not they are actually the shortest path. The algorithm works by evaluating the cost of each possible path and then expanding the path with the lowest cost. This process is repeated until the goal is reached.

- Greedy Best-First Search works by evaluating the cost of each possible path and then expanding the path with the lowest cost. This process is repeated until the goal is reached.
- The algorithm uses a heuristic function to determine which path is the most promising.
- The heuristic function takes into account the cost of the current path and the estimated cost of the remaining paths.
- If the cost of the current path is lower than the estimated cost of the remaining paths, then the current path is chosen. This process is repeated until the goal is reached.

Algorithm:

1. Initialize a tree with the root node being the start node in the open list.
2. If the open list is empty, return a failure, otherwise, add the current node to the closed list.
3. Remove the node with the lowest h(x) value from the open list for exploration.
4. If a child node is the target, return a success. Otherwise, if the node has not been in either the open or closed list, add it to the open list for exploration

Input Graph:



Output:

S
B
F
G
2

Program:

```
graph = { 'S':[('A',11), ('B',5)],
  'A':[('C',9),
  ('D',8)], 'C':[],
  'D':[],
  'B':[('E',4),('F',2)],
  'E':[('H',7),('G1',0)],
  'F':[('I',3),('G2',0)]
  }
  def bfs(start, target, graph, queue=[],
     visited=[]): if start not in visited:
        print(start)
        visited.append(star
        t)
     queue=queue+[x for x in graph[start] if x[0][0] not in
     visited] queue.sort(key=lambda x:x[1])
     if
        queue[0][0]==targ
        et:
        print(queue[0][0])
     else:
        processing=queue[0]
        queue.remove(processing)
        bfs(processing[0], target, graph, queue, visited)

  bfs('S', 'G2', graph)
```

2.b.Aim: Write a program to implement A* search algorithm in python.

Procedure:

Total estimated cost f(n)

The total estimated cost f(n) is the cornerstone of A* algorithm's decision-making process, combining both the actual path cost and the heuristic estimate to evaluate each node's potential. For any node n, this cost is calculated as:

$$f(n) = g(n) + h(n)$$

Where:

g(n) represents the actual cost from the start to the current node,
h(n) represents the estimated cost from the current node to the goal.

The A* algorithm maintains two essential lists

Open list:

- Contains nodes that need to be evaluated
- Sorted by f(n) value (lowest first)
- New nodes are added as they're discovered

Closed list:

- Contains already evaluated nodes
- Helps avoid re-evaluating nodes
- Used to reconstruct the final path

The algorithm continually selects the node with the lowest f(n) value from the open list, evaluates it, and moves it to the closed list until it reaches the goal node or determines no path exists.

1. **Initialization:** The algorithm initializes the starting node and creates an open set and a closed set. The open set contains nodes that have been visited but their neighbors have not been explored yet, while the closed set contains nodes that have been visited along with their explored neighbors.

2. **Calculating the Cost Function:** The algorithm calculates the cost function for the starting node based on the path value (the weight of traversing from one node to another) and the heuristic value of the node. The cost function is the sum of these two values.

3. **Exploring Neighboring Nodes:** The algorithm explores the neighboring nodes of the current node and calculates the cost function for each neighbor. It adds the neighbors to the open set if they are not already in it and updates their parent nodes if their cost function is lower than the current cost function.

4. **Goal Check:** The algorithm checks if the current node is the target node or the final destination. If it is, the algorithm terminates and uses pointers to Trace back the path from the target node to the starting node.

5. **Finding the Node with the Lowest Cost Function:** The algorithm selects the node with the lowest cost function from the open set and sets it as the current node. It removes this node from the open set and adds it to the closed set.

6. **Repeat the Process:** Steps 3-5 are repeated until either the target node is reached or there are no more nodes in the open .

Program:

```
from collections import deque

class Graph:
    # example of adjacency list (or rather map)
    # adjacency_list = {
    # 'A': [('B', 1), ('C', 3), ('D', 7)],
    # 'B': [('D', 5)],
    # 'C': [('D', 12)]
    # }

    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        return self.adjacency_list[v]

    # heuristic function with equal values for all nodes
    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }

        return H[n]

    def a_star_algorithm(self, start_node, stop_node):
        # open_list is a list of nodes which have been visited, but who's neighbors
        # haven't all been inspected, starts off with the start node
        # closed_list is a list of nodes which have been visited
        # and who's neighbors have been inspected

        open_list = set([start_node])
        closed_list = set([])

        # g contains current distances from start_node to all other nodes
        # the default value (if it's not found in the map) is +infinity
        g = {}

        g[start_node] = 0

        # parents contains an adjacency map of all nodes
        parents = {}
        parents[start_node] = start_node

        while len(open_list) > 0:
            n = None

            # find a node with the lowest value of f() - evaluation function
            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n] + self.h(n):
                    n = v;
```

```
            if n == None:
                print('Path does not exist!')
                return None

            # if the current node is the stop_node
            # then we begin reconstructin the path from it to the start_node
            if n == stop_node:
                reconst_path = []
                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]
                reconst_path.append(start_node)
                reconst_path.reverse()
                print('Path found: {}'.format(reconst_path))
                return reconst_path

            # for all neighbors of the current node do
            for (m, weight) in self.get_neighbors(n):
                # if the current node isn't in both open_list and closed_list
                # add it to open_list and note n as it's parent
                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                # otherwise, check if it's quicker to first visit n, then m
                # and if it is, update parent data and g data
                # and if the node was in the closed_list, move it to open_list
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n

                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)

            # remove n from the open_list, and add it to closed_list
            # because all of his neighbors were inspected
            open_list.remove(n)
            closed_list.add(n)

        print('Path does not exist!')
        return None
adjacency_list = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')
```

Output:

```
Path found: ['A', 'B', 'D']
['A', 'B', 'D']
```

3.A.<u>Aim</u>-Write a program to impliment Factorial,Fibonacci,Towers of Hanoi Predicates using Prolog.

<u>Procedure</u><u>:</u>Prolog language basically has three different elements:

Facts: The fact is predicate that is true, for example, if we say, "Tom is the son of Jack",then this is a fact.

Rules: Rules are extinctions of facts that contain conditional clauses. To satisfy a rule these conditions should be met. For example, if we define a rule as:

This implies that for *X* to be the grandfather of *Y*, Z should be a parent of Y and X shouldbe father of Z.

Queries: And to run a prolog program, we need some questions, and those questions can be answered by the given facts and rules.

Program:

%Find Fibonacci Number Of N^th term

```
fib(1,0). % FACT-1 term of fibonacci is 0
fib(2,1). %FACT-2 term of fibonacci is 1
%RULE-Nth term fibonacci series
fib(N,X):- N1 is N-1,N2 is N-2,fib(N1,X1),fib(N2,X2),X is X1+X2.
```

%Factorial of number

```
fact(0,1). % FACT-factorial of 0 is 1
            %factorial of number N is
fact(N,X):-N1 is N-1,fact(N1,Y),X is Y*N,!.
```

%Towers of Hanoi

```
move(1,X,Y,_):-write("move disk from"),write(X),write('to'),write(Y),nl.
move(N,X,Y,Z):-N>1,M is N-1,move(M,X,Z,Y),move(1,X,Y,_),move(M,Z,Y,X).
Toh(N):-move(N,left,right,center).
```

Queries:

```
? - fact(5).
? -fib(8,X).
?- Toh(3).
```

3.B. <u>Aim</u>-Write a program for Family tree-predicates and queries using prolog.

<u>Procedure:</u>Prolog language basically has three different elements:

Facts: The fact is predicate that is true, for example, if we say, "Tom is the son of Jack",then this is a fact.

Rules: Rules are extinctions of facts that contain conditional clauses. To satisfy a rule these conditions should be met. For example, if we define a rule as:

This implies that for *X* to be the grandfather of *Y*, Z should be a parent of Y and X shouldbe father of Z.

Queries: And to run a prolog program, we need some questions, and those questions can be answered by the given facts and rules.

<u>Program:</u>

%FACTS:pam is parent of bob

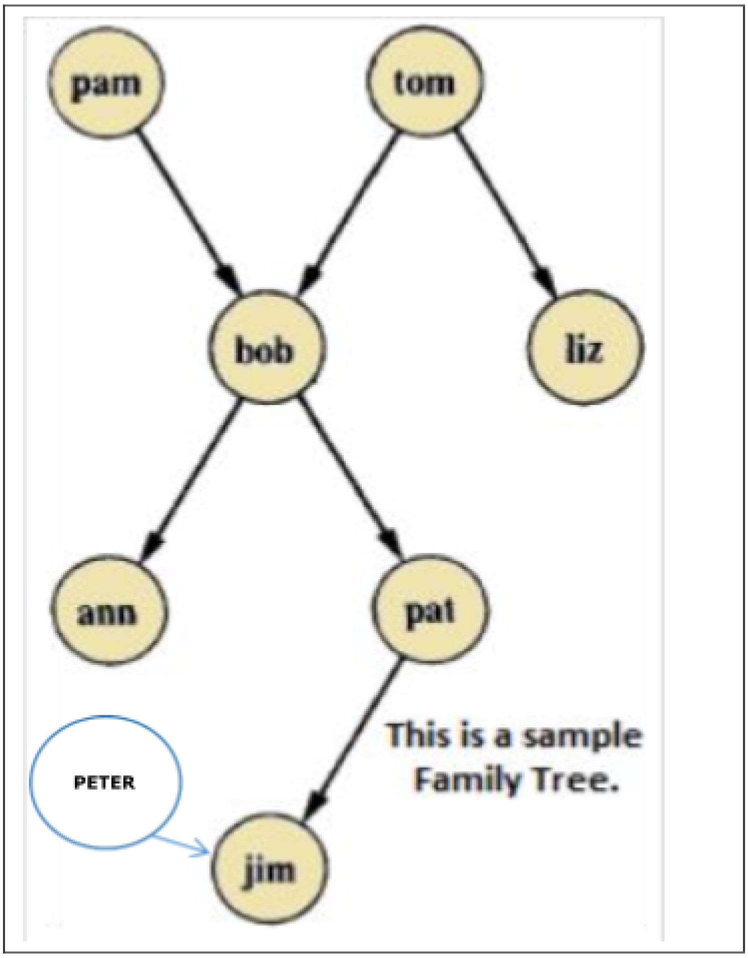
```
parent(pam, bob).
parent(tom, bob).
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).
parent(bob, peter).
parent(peter, jim).
```

%FACTS:tom is male,bob is male.

```
male(tom).
male(bob).
male(jim).
male(peter).
```

%FACTS:pam is female,

```
female(pam).
female(liz).
female(pat).
female(ann).
```



%RULES ON FAMILY PREDICATES

```
mother(X,Y):- parent(X,Y),female(X).
father(X,Y):-parent(X,Y),male(X).
sister(X,Y):-parent(Z,X),parent(Z,Y),female(X),X\==Y.
brother(X,Y):-parent(Z,X),parent(Z,Y),male(X),X\==Y.
grandparent(X,Y):-parent(X,Z),parent(Z,Y).
grandmother(X,Z):-mother(X,Y),parent(Y,Z).
grandfather(X,Z):-father(X,Y),parent(Y,Z).
wife(X,Y):parent(X,Z),parent(Y,Z),female(X),male(Y)
uncle(X,Z):-brother(X,Y),parent(Y,Z).
predecessor(X, Z) :- parent(X, Z).
predecessor(X, Z) :- parent(X, Y),predecessor(Y, Z).
```

Queries:

1. **?- predecessor(peter,X).**

```
X = jim ? ;
```

2. **| ?- parent(X,jim).**

```
X = pat ? ;
X = peter

yes
```

3. **| ?- mother(X,Y).**

```
X = pam
Y = bob ? ;

X = pat
Y = jim ? ;

no
```

4. **| ?- haschild(X).**

```
X = pam ? ;

X = tom ? ;
```

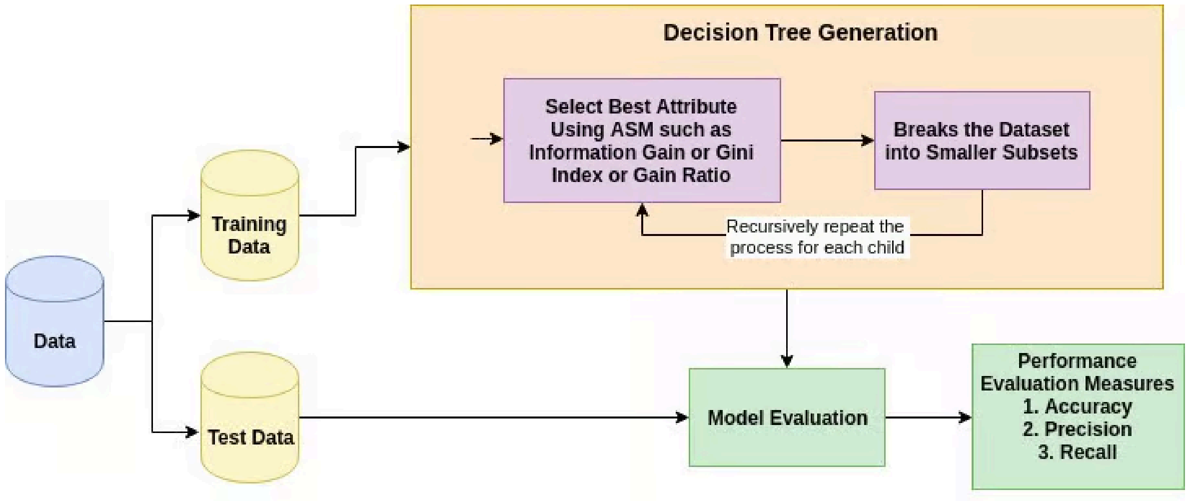
4.A.Write a program to train and validate the Decision Tree classifiers given data using python(scikit-learn).

Aim: Write a program to train and validate the Decision Tree classifiers given data using python(scikit-learn).

Procedure:

1. Select the best attribute using Attribute Selection Measures (ASM) to split the records.
2. Make that attribute a decision node and breaks the dataset into smaller subsets.
3. Start tree building by repeating this process recursively for each child until one of the conditions will match:

- All the tuples belong to the same attribute value.
- There are no more remaining attributes.
- There are no more instances.



<u>Calculate Information Gain:</u>

Information Gain = entropy(parent) – [weighted average] * entropy(children)

$$Entropy = -\sum_{i=1}^{n} p_i \log_2 p_i$$

Steps to split a decision tree using Information Gain:

1. For each split, individually calculate the entropy of each child node
2. Calculate the entropy of each split as the weighted average entropy of child nodes
3. Select the split with the lowest entropy or highest information gain
4. Until you achieve homogeneous nodes, repeat steps 1-3

Program:

```
# Load libraries
import pandas as pd
from sklearn.tree import DecisionTreeClassifier # Import Decision Tree Classifier
from sklearn.model_selection import train_test_split # Import train_test_split function
from sklearn import metrics #Import scikit-learn metrics module for accuracy calculation


col_names = ['pregnant', 'glucose', 'bp', 'skin', 'insulin', 'bmi', 'pedigree', 'age', 'label']

# load dataset
pima = pd.read_csv("diabetes.csv", header=None, names=col_names)

pima.head()

#split dataset in features and target variable
feature_cols = ['pregnant', 'insulin', 'bmi', 'age','glucose','bp','pedigree']
X = pima[feature_cols] # Features
y = pima.label # Target variable


# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1) # 70% training and 30% test


# Create Decision Tree classifer object
clf = DecisionTreeClassifier()

# Train Decision Tree Classifer
clf = clf.fit(X_train,y_train)

#Predict the response for test dataset
y_pred = clf.predict(X_test)


# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))


from sklearn.tree import export_graphviz
from sklearn.externals.six import StringIO
from IPython.display import Image
import pydotplus

dot_data = StringIO()
export_graphviz(clf, out_file=dot_data,
                filled=True, rounded=True,
                special_characters=True,feature_names = feature_cols,class_names=['0','1'])
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
graph.write_png('diabetes.png')
Image(graph.create_png())
```

4.B.Write a program to Pre Process data using Python.

Aim:. Write a program to Pre Process data using Python.

Procedure:

1. Load data in Pandas.
2. Drop columns that aren't useful.
3. Drop rows with missing values.
4. Create dummy variables.
5. Take care of missing data.
6. Convert the data frame to NumPy.
7. Divide the data set into training data and test data.

Program:

```
#Load data in Pandas.
df = pd.read_csv('train.csv')


#Drop columns that aren't useful.
cols = ['Name', 'Ticket', 'Cabin']
df = df.drop(cols, axis=1)

#Drop rows with missing values.
df = df.dropna()

#Create dummy variables.
dummies = []
cols = ['Pclass', 'Sex', 'Embarked']
for col in cols:
   dummies.append(pd.get_dummies(df[col]))

#Take care of missing data.
df['Age'] = df['Age'].interpolate()


#Convert the data frame to NumPy.
X = df.values
y = df['Survived'].values


#Divide the data set into training data and test data.
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
```

5.A.Write a program to solve Monkey Banana Problem using prolog.

Aim:program to solve Monkey banana problem using state space search.

Procedure to solve AI problem:

Environment:

1.Monkey at a door into the Room
2.In the middle of the room,there is a monkey hanging from ceiling.
3.The monkey is hungry and wants to get banana but he can't stretch enough from the floor.
4.At the window of the room,there a box the monkey may use.

Monkey's possible actions:

1. walk on the floor
2. climb the box
3. push the box around if its already at the box
4. Grasp the banana if the standing on the box directly under banana.

Problem states:

state(A,B,C,D):

A:Horizontal position of Monkey(middle,atdoor)
B:Vertical Position of Monkey(onfloor,onbox)
C:Position of Box.(middle,atwindow)
D:Monkey has or has not banana.(has,hasnot)

Initial state:-state(atdoor,onfloor,atwindow,hasnot).
goal state:-state(_,_,_,has).

Program:

```
move(state(middle,onbox,middle,hasnot),
        grasp,
    state(middle,onbox,middle,has)).

move(state(P,onfloor,P,H),
        climb,
     state(P,onbox,P,H)).

move(state(P1,onfloor,P1,H),
        drag(P1,P2),
    state(P2,onfloor,P2,H)).

move(state(P1,onfloor,B,H),
        walk(P1,P2),
    state(P2,onfloor,B,H)).
canget(state(_,_,_,has)).

canget(State1) :-move(State1,_,State2),canget(State2).
```

Query:?- canget(state(atdoor,onfloor,atwindow,hasnot)).

Output: True.

5.A.Write a program to solve Water jug Problem using prolog.

Aim: Write a program to solve Water jug Problem using prolog.

Procedure:

State(X,Y):
X:water jug that can hold 4 litres of water.X=0,1,2,3,4
Y:water jug that hold 3 litres of water.Y=0,1,2,3.

Initial State:(0,0).
Goal State:(2,n) for any n.

Algorithm:

Water Jug Problem in Artificial Intelligence

1	(x, y) is X<4 ->(4, Y)	Fill the 4-liter jug
2	(x, y) if Y<3 -> (x, 3)	Fill the 3-liter jug
3	(x, y) if x>0 -> (x-d, y)	Pour some water out of the 4-liter jug.
4	(x, y) if Y>0 -> (x, y-d)	Pour some water out of the 3-liter jug.
5	(x, y) if x>0 -> (0, y)	Empty the 4-liter jug on the ground
6	(x, y) if y>0 -> (x,0)	Empty the 3-liter jug on the ground
7	(x, y) if X+Y >= 4 and y>0 -> (4, y-(4-x))	Pour water from the 3-liter jug into the 4-liter jug until the 4-liter jug is full
8	(x, y) if X+Y>=3 and x>0 -> (x-(3-y), 3))	Pour water from the 4-liter jug into the 3-liter jug until the 3-liter jug is full.
9	(x, y) if X+Y <=4 and y>0 -> (x+y, 0)	Pour all the water from the 3-liter jug into the 4-liter jug.
10	(x, y) if X+Y<=3 and x>0 -> (0, x+y)	Pour all the water from the 4-liter jug into the 3-liter jug.
11	(0, 2) -> (2, 0)	Pour the 2-liter water from the 3-liter jug into the 4-liter jug.
12	(2, Y) -> (0, y)	Empty the 2-liter in the 4-liter jug on the ground.

1.START STATE (0,0)
2.LOOP UNTIL GOAL STATE REACHED (2,n) :

- APPLY A RULE WHOSE LEFT SIDE MATCHES CURRENT STATE.
- SET THE NEW CURRENT STATE TO BE THE RESULTING STATE.

Program:

```
%database
  dynamic visited_state(integer,integer).

%predicates
   state(integer,integer).

%clauses
   state(2,0).

state(X,Y):- X < 4,
   not(visited_state(4,Y)),
   assert(visited_state(X,Y)),
   write("Fill the 4-Gallon Jug: (",X,",",Y,") --> (", 4,",",Y,")\n"),
   state(4,Y).

   state(X,Y):- Y < 3,
         not(visited_state(X,3)),
         assert(visited_state(X,Y)),
         write("Fill the 3-Gallon Jug: (", X,",",Y,") --> (", X,",",3,")\n"),
         state(X,3).

   state(X,Y):- X > 0,
         not(visited_state(0,Y)),
         assert(visited_state(X,Y)),
         write("Empty the 4-Gallon jug on ground: (", X,",",Y,") --> (", 0,",",Y,")\n"),
         state(0,Y).

   state(X,Y):- Y > 0,
         not(visited_state(X,0)),
         assert(visited_state(X,0)),
         write("Empty the 3-Gallon jug on ground: (", X,",",Y,") --> (", X,",",0,")\n"),
         state(X,0).

   state(X,Y):- X + Y >= 4,
         Y > 0,
         NEW_Y = Y - (4 - X),
         not(visited_state(4,NEW_Y)),
         assert(visited_state(X,Y)),
         write("Pour water from 3-Gallon jug to 4-gallon until it is full: (", X,",",Y,") --> (", 4,",",NEW_Y,")\n"),
         state(4,NEW_Y).


    state(X,Y):- X + Y >=3,
         X > 0,
         NEW_X = X - (3 - Y),
         not(visited_state(X,3)),
         assert(visited_state(X,Y)),
         write("Pour water from 4-Gallon jug to 3-gallon until it is full: (", X,",",Y,") --> (", NEW_X,",",3,")\n"),
         state(NEW_X,3).
```

```
 state(X,Y):- X + Y>=4,
          Y > 0,
          NEW_X = X + Y,
          not(visited_state(NEW_X,0)),
          assert(visited_state(X,Y)),
          write("Pour all the water from 3-Gallon jug to 4-gallon: (", X,",",Y,") --> (", NEW_X,",",0,")\n"),
          state(NEW_X,0).

   state(X,Y):- X+Y >=3,
          X > 0,
          NEW_Y = X + Y,
          not(visited_state(0,NEW_Y)),
          assert(visited_state(X,Y)),
          write("Pour all the water from 4-Gallon jug to 3-gallon: (", X,",",Y,") --> (", 0,",",NEW_Y,")\n"),
          state(0,NEW_Y).

   state(0,2):- not(visited_state(2,0)),
          assert(visited_state(0,2)),
          write("Pour 2 gallons from 3-Gallon jug to 4-gallon: (", 0,",",2,") --> (", 2,",",0,")\n"),
          state(2,0).

   state(2,Y):- not(visited_state(0,Y)),
          assert(visited_state(2,Y)),
          write("Empty 2 gallons from 4-Gallon jug on the ground: (", 2,",",Y,") --> (", 0,",",Y,")\n"),
          state(0,Y).

goal:-
           makewindow(1,2,3,"4-3 Water Jug Problem",0,0,25,80),
           state(0,0).
```