INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
Dundigal - 500 043, Hyderabad, Telangana

COURSE CONTENT

PROGRAMMING IN LOGIC LABORATORY								
V Semester: CSE (AI & ML)								
Course Code	**Category**	**Hours / Week**			**Credits**	**Maximum Marks**		
		L	**T**	**P**	**C**	**CIA**	**SEE**	**Total**
ACAC12	**Core**	0	0	3	1.5	30	70	100
Contact Classes: Nil	**Tutorial Classes: Nil**	**Practical Classes: 36**				**Total Classes:36**		
Prerequisite: Artificial Intelligence and Expert Systems								

I.COURSE OVERVIEW:
This course covers fundamental concepts and underlying assumptions about intelligence. The goal is to produce programs to do intelligent things as people do. It also explains different kinds of techniques useful for solving AI problems. It gives an insight to model human intelligence. Different Heuristic approaches are explored to measure how far a node in a search tree seems to be from a goal. Intelligence requires knowledge. Defining the problem accurately and segregating the background knowledge needed in the solution of the problem are clearly stated and implemented.

II. COURSE OBJECTIVES:
The students will try to learn:

I. The basic concepts of Artificial Intelligence and Expert systems.
II. Identify and apply different search techniques and algorithms to solve real world problems.
III. Design Python programs for various Learning algorithms.

III.COURSE OUTCOMES:
At the end of the course students will be able to:

CO1	Summarize knowledge representation and issues in AI and Related fields.	Understand
CO2	Choose knowledge reasoning with predicate logic and inference rules to solve problems to new situations.	Apply
CO3	Make use of Heuristic, Adversarial search and game playing algorithms for addressing a particular AI problem and implement the selected strategy.	Apply
CO4	Experiment with uncertainty issues by using statistical and symbolic reasoning approaches.	Apply
CO5	Select the various algorithms used in the prediction and perception of things in an intelligent environment	Apply
CO6	Utilize knowledge representation with the help of Expert systems to solve complex problems and to provide decision-making ability	Apply

IV. COURSE CONTENT:

EXERCISES FOR PROGRAMMING IN LOGIC LABORATORY

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

1. Getting Started Exercises

1.1 Knowledge Base and Querying

1. Install SWI Prolog on your machine.
2. Write a prolog program to create a knowledge base and querying including:
    - Facts and Rules representing Alex and John
3. Upload the code into prolog so that it can learn from the facts and rules mentioned in the knowledge base.
4. Find the answers for the queries:
5.
    - Who is the progenitor of John.
    - Who is the successor of Alex.

Try: Create a knowledge base including the likes and dislikes of George and Kate. Find the answers to the queries like What George and Kate like.

1.2 Facts and Rules

Prepare the knowledge base including the facts and rules to perform the conversion of Centigrade temperature to Fahrenheit temperature and knowing the freezing temperature.

Input: Two rules, one for conversion and the other to know the freezing temperature value.

Output: X = 212.0, 50.90, and freezing output as false for 45 degrees and true for 15 degrees.

Hint:

```
c_to_f(C, F) :- F is ((/*write the formula here */)).
freezing(F) :- F = < 32.
```

Try:

1. Create a knowledge base to perform the factorial of a given number.
2. Define the predicate palindrome(List). A list is a palindrome if it reads the same in the forward and in the backward direction. For example, [m, a, d, a, m]

1.3 Population Density

Prepare a knowledge base including facts and rules to determine the population density in various countries like USA, India, China, Brazil using '**is**' operator. Start asking the queries to know the population density of USA and China.

Input: Facts including the population and areas, rules representing the calculation of population density.

Output: Population density of USA and China.

Hints:

```
/* Prepare 4 facts representing the population density in millions */

/* Prepare 4 facts representing the country names */

density(X,Y) :- The population density of country X is Y, if:
        pop(X,P),
        The population of X is P, and
        area(X,A), The area of X is A, and
        Y is P/A. Y is calculated by dividing P by A.
```

Try: Create the Knowledge base equivalent to the statements mentioned below in Prolog.

1. John likes all kinds of food.
2. Apple and vegetables are food.
3. Anything anyone eats and not killed is food.
4. Anil eats peanuts and is still alive.
5. Harry eats everything that Anil eats.

1.4 Assertions and Queries

Create a knowledge base including facts and rules (like axioms) and generate queries presenting that it is in effect a theorem can be proved. Talk about two modes like entering assertions and making queries.

Hints

```
likes(fred,beer).
likes(fred,cheap_cigars).
likes(fred,monday_night_football).
likes(sue,jogging).
likes(sue,yogurt).
likes(sue,bicycling).
likes(sue,amy_goodman).

likes(mary,jogging).
likes(mary,yogurt).
likes(mary,bicycling).
likes(mary,rush_limbaugh).

/* Represent the likes of yogurt as jogging using a variable X*/

health_freak(X) :-
   likes(X,yogurt),
   likes(X,jogging).

left_wing(X) :-
   likes(X,amy_goodman).

right_wing(X) :-
   likes(X,rush_limbaugh).

low_life(X) :-
  likes(X,cheap_cigars).
```

Try:

Consider the following program:

```
f( 1, one).
f( s(1), two).
f( s((1)), three).
f( s(s(s(X))), N) :- f( x, N).
```

How will Prolog answer the following questions? Whenever several answers are possible, give at least two.

(a) ?- f(s(1), A).
(b) ?- f(s(s(l), two).
(c) ?- f(s(s(s(s(s(1))))))), C).
(d) ?- (D, three)

1.5 Unification

Create a knowledge base and include facts and rules that are always true. Start asking queries to the knowledge base to prove that prolog will attempt to return every solution the order them appear in the program. Also prove that prolog uses Unification to match queries with rule heads and facts.

Input: Facts representing the likes of Max, Julia, and Amabel.

Output: Return every solution that matches the queries.

Hints

```
jealous(Jealous, Victim) :- likes(Person, Jealous),
        likes(Person, Victim).
likes(max, julia).
likes(max, amabel).

/* Step 1 - add to stack */
jealous(julia, Victim) :- likes(Person, julia),
        likes(Person, Victim).
/* Step 2 - match first subgoal */
jealous(julia, Victim) :- likes(max, julia),
        likes(max, Victim).
/* Step 3 - match second subgoal */
jealous(julia, julia) :- likes(max, julia),
        likes(max, julia).
        jealous(julia, amabel) :- likes(max, julia),
                likes(max, amabel).
/* Step 4 - report success to the user */
| ?- jealous(julia, Who).
Who = julia ;
jealous(julia, amabel) :- likes(max, julia),
        likes(max, amabel).

/*Step 5 - report success to the user */
| ?- jealous(julia, Who).
Who = julia ;
Who = amabel ;

/*Step 6 - report no more solutions */
| ?- jealous(julia, Who).
Who = julia ;
Who = amabel ;
no
```

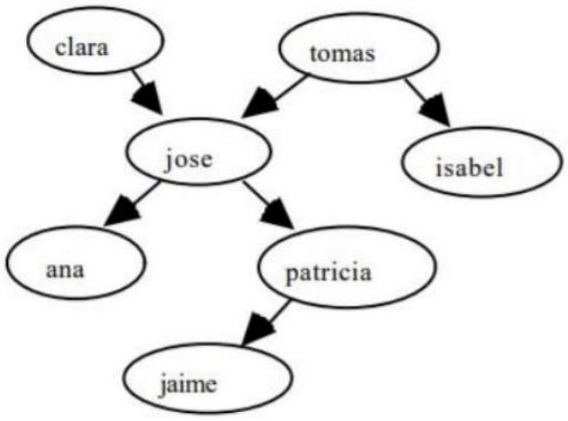
Try: Create a knowledge base including facts and rules defining our friend as being either an immediate friend of ours or a friend of a friend.

1.6 Conjunction and Disjunction of Objectives

Conjunction: Create a knowledge base including the following facts, start asking questions and apply unification.
Disjunction: Suppose we have a predicate, father(jhon, bob). This tells that "Jhon is father of Bob", and another predicate, mother(lili,bob), this tells that "lili is mother of bob". If we create another predicate as child(), this will be true when father(jhon, bob) is true OR mother(lili,bob) is true.

Input:

progenitor (clara, jose).
progenitor (thomas, jose).
progenitor (thomas, isabel).
progenitor (jose, ana).
progenitor (jose, patricia).
progenitor (patricia, jaime).



Output: The progenitor of Thomas, progenitor of Clara and Isabel, and know the progenitor of Thomas using a variable.

Hints

```
/*
 Trying if-else statement and modulus (%) operator.
 */
/* Let us ask some questions: Hint: Do unification. */

?- progenitor (thomas, jose), progenitor (thomas, Isabel).  /* Predict the answer */
?- progenitor (clara, jose), progenitor (isabel, patricia). /* Predict the answer */
?- progenitor (thomas, X), progenitor (X, Y). /* Predict the answer and check the
possibilities of other answers */
```

```
parent(jhon,bob).
parent(lili,bob).

male(jhon).
female(lili).
```

```
% Conjunction Logic
father(X,Y) :- parent(X,Y),male(X).
mother(X,Y) :- parent(X,Y),female(X).

% Disjunction Logic
child_of(X,Y) :- father(X,Y);mother(X,Y).
```

Try:

a. ?- progenitor(jaime, X).
b. ?- progenitor(X, jaime).
c. ?- progenitor(clara, X), progenitor(X, patricia).
d. ?- progenitor(thomas, X), progenitor(X,Y), progenitor(Y,Z).

Observation: Here ',' represents '^' which is logical connective AND.

1.7 Lists

Create a knowledge base and include facts and rules that are always true. Start asking queries to the knowledge base to prove that prolog will attempt to return every solution the order them appear in the program. Also prove that prolog uses Unification to match queries with rule heads and facts.

Input: Facts representing the likes of Max, Julia, and Amabel.

Output: Return every solution that matches the queries.

Hints

```
jealous(Jealous, Victim) :- likes(Person, Jealous),
        likes(Person, Victim).
likes(max, julia).
likes(max, amabel).

/* Step 1 - add to stack */
jealous(julia, Victim) :- likes(Person, julia),
        likes(Person, Victim).

/* Step 2 - match first subgoal */
jealous(julia, Victim) :- likes(max, julia),
        likes(max, Victim).

/* Step 3 - match second subgoal */
jealous(julia, julia) :- likes(max, julia),
        likes(max, julia).
        jealous(julia, amabel) :- likes(max, julia),
                likes(max, amabel).

/* Step 4 - report success to the user */
| ?- jealous(julia, Who).
Who = julia ;
jealous(julia, amabel) :- likes(max, julia),
        likes(max, amabel).

/*Step 5 - report success to the user */
| ?- jealous(julia, Who).
```

```
Who = julia ;
Who = amabel ;

/*Step 6 - report no more solutions */
| ?- jealous(julia, Who).
Who = julia ;
Who = amabel ;
no
```

Try: Create a knowledge base including facts and rules defining our friend as being either an immediate friend of ours or a friend of a friend.

1.8 Permutations

Sometimes it is useful to generate permutations of a given list. Define the permutation relation with two arguments. The arguments are two lists such that one is a permutation of the other. The intention is to generate permutations of a list through backtracking using the permutation procedure.

Input: Two Lists

Output: Instantiate a List L successfully to all possible permutations.

Hints

```
/* The program for permutation can be, again, based on the consideration of two
cases, depending on the first list:
(1) If the first list is empty then the second list must also be empty.
(2) If the first list is not empty then it has the form [X I L], and a permutation of
such a list can be constructed as shown in Figure: first permute L obtaining L1 and
then insert X at any position into L1. */

Two Prolog clauses that correspond to these two cases are:

permutation( [ ], [ ] ).
permutation( [X I L], P) :- permutation( L, L1), insert( X, L1, P).
One alternative to this program would be to delete an element, X, from the
first list, permute the rest of it obtaining a list P, and then add X in
front of P. The corresponding program is:

permutation2( [ ], [ ]).
permutation2(L, [X|P]) :-
        del(X, L, L1),
        permutation2(L1, P).
It is instructive to do some experiments with our permutation programs. Its
normal use would be something like this:
?- permutation( [red, blue, green], P).

This would result in all six permutations, as intended:

P = [red, blue, green];
P = [red, green, blue];
P = [blue, red, green];
P = [blue, green, red];
P = [green, red, blue];
P = [green, blue, red];
```

```
no

Another attempt to use permutation is:
?- permutation(L, [a, b, c]).
```

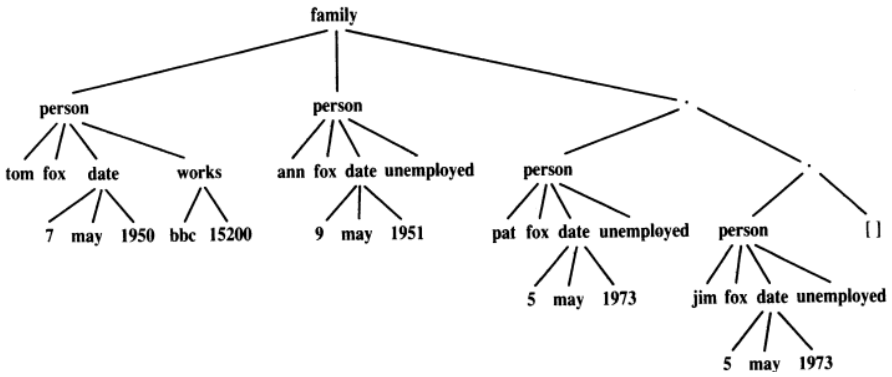
Try:

1. Define two predicates evenlength(List) and oddlength(List) so that they are true if their argument is a list of even or odd length respectively. For example, the list [a,b,c,d] is 'evenlength' and [a,b,c] is 'oddlength'.

2. Define the relation
reverse(List, Reversedlist)
that reverses lists. For example, reverse([a,b,c,d], [d,c,b,a]).

1.9 Structures

Create a knowledge base to develop the skills of representing and manipulating structured data objects. Prepare the knowledge about the families representing each family by one clause. Make sure that each family has three components like husband, wife, and children. Each person is, in turn, represented by a structure of four components like name, surname, date of birth, and job. Here the job information is 'unemployed', or it specifies the working organization and salary.



Input: Sequence of facts representing the three components of a family and four components of each person.

Output:

a. Find the names of all the people in the database?
b. Find all children born in 1981?
c. Find all employed wives?
d. Find the names of unemployed people who were born before 1963?
e. Find people born before 1950 whose salary is less than 8000?
f. Find the names of families with at least three children?

Hints

```
/* Create the knowledge base representing the family information mentioned in the figure */
```

```
family(
        person( tom, fox, date(7,may,1950), works(bbc,15200) ),
        person( ann, fox, dat{9,may, 1951), unemployed),
        [ person( pat, fox, date(5,may,1973), unemployed),
                person( jim, fox, date(S,may,1973), unemployed) ] ).

/* Provide a set of procedures that can serve as a utility to make the interaction
with the database more comfortable and make them as a part of the user interface */

husband( X) :-                          % X is a husband.
        family(X, _, _).
wife(X) :-                              % X is a wife.
        family(_, X, _).
child(X) :-                             % X is a child.
        family(_, _, Children),
        member(X, Children).
member(X, [X | L]).
member(X, X, [Y | L) :-
        member(X, L).

exists(Person) :-                       % Any person in the knowledge base.
        husband(Person);
        wife(Person);
        child(Person).

dataofbirth(person(_, _, Date, _), Date).
salary(person(_, _, _, works(_, S) ), S).     % Salary of working person
salary(person(_, _, _, unemployed), 0).       % Salary of unemployed
```

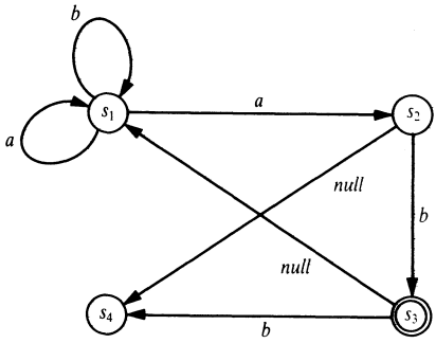
Try:

1. Calculate the total income of a family it is useful to define the sum of salaries of a list of people as a two-argument relation.
2. Find all married women that have at least three children.

1.10 Simulating a non-deterministic automaton.

Create a knowledge base that shows how an abstract mathematical construct can be translated into Prolog. Also create a non-deterministic finite automaton as an abstract machine that reads as input a string of symbols and decides whether to accept or to reject the input string.

Input: Transition graph with s_1 (initial state), s_2, s_3 and s_4 states.



Output:

1. About the acceptance of the string *aaab*.
2. Which state our automation can be in initially so that it will accept the string *ab*.
3. All the strings of length 3 that are accepted from state s_1.

Hints

```
/* Program the simulator as a binary relation, accepts, which defines the acceptance
of a string from a given state. */

accept{ State, String)
/* is true if the automaton, starting from the state State as initial state, accepts
the string String. The accepts relation can be defined by three clauses. They
correspond to the following three cases:
(1) The empty string, [], is accepted from a state S if S is a final state.
(2) A non-empty string is accepted from a state S if reading the first symbol in the
string can bring the automaton into some state S1, and the rest of the string is
accepted from S1.
(3) A string is accepted from a state s if the automaton can make a silent move from
S to 51 and then accept the (whole) input string from s1.*/

These rules can be translated into Prolog as:

accept(S, [ ]) :-                              % Accept empty string.
        final(S).
accepts(S, [X | Rest]) :-                      % Accept by reading first symbol.
        trans(S, X, S1),
        accepts(S1, Rest).
accepts(S, String) :-                          % Accept by making silent move.
        silent(S, S1),
        accepts(S1, String).
```

Try: From what states will the automaton accept input strings of length 7?

2. Exercises on First Order Predicate Logic using Prolog

2.1 Translate

The goal of this exercise is to translate each of the following sentences into First Order Logic (FOL). Later convert the FOL into a prolog program and asking questions.

Input: Domain Knowledge like:

(a) Not all cars have carburetors.
(b) Some people are either religious or pious.
(c) No dogs are intelligent.
(d) All babies are illogical.
(e) Every number is either negative or has a square root.
(f) Some numbers are not real.
(g) Every connected and circuit-free graph is a tree.

Output: Equivalent FOL statement.

Hints

$\neg\forall x\ [car(x) \rightarrow carburetors(x)]$ or
$\exists x\ [car(x) \wedge \neg carburetors(x)]$

```
# Translate the other statements referring the above
```

Try: Translate each of the following sentences into First Order Logic (FOL):

(a) Not every graph is connected.
(b) All that glitters is not gold.
(c) Not all that glitters is gold.
(d) There is a barber who shaves all men in the town who do not shave themselves.
(e) There is no business-like show business.

2.2 Translate

The goal of this exercise is to rewrite each proposition symbolically, given that the universe of discourse is a set of real numbers.

Input: Domain Knowledge like:

(a) For each integer x, there exist an integer y such that x + y = 0.
(b) There exist an integer x such that x + y = y for every integer y.
(c) For all integers x and y, x.y = y.x
(d) There are integers x and y such that x+y=5.

Output: Equivalent FOL statement

Hints

$(\forall x \in \mathbb{Z})(\exists y \in \mathbb{Z})(x + y = 0)$.
We could read this as, "For every integer x, there exists an integer y such that $x + y = 0$." This is a true statement.

```
# Translate the other statements referring the above
```

Try: Using FOL, express the following:

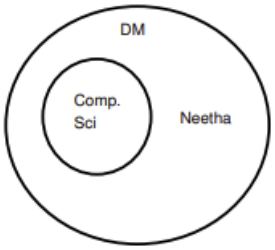
(a) Every student in this class has taken exactly two mathematics courses at this school.
(b) Someone has visited every country in the world except Libya.
(c) No one has climbed every mountain in the Himalayas.

2.3 Check the Availability

Every computer science student takes discrete mathematics. Neetha is taking discrete mathematics. Therefore, Neetha is a computer science student. The given conclusion is false. The following Venn diagram is a counter example for the given conclusion.

If it does not rain or it is not foggy then the sailing race will be held, and lifesaving demonstrations will go on. If the sailing race is held, then the trophy will be awarded. The trophy was not awarded. Therefore, it rained. The goal of this exercise is to translate each of the following sentences into First Order Logic (FOL)

Input: Venn Diagram as a counter example for the given conclusion.



Output: Prove that the above statements are TRUE.

Hints

```
# Consider the below statements and infer the statement by the following arguments
```

$premise$	$\neg R \vee \neg F \rightarrow S \wedge D$	$\ldots(1)$
$premise$	$S \rightarrow T$	$\ldots(2)$
$premise$	$\neg T$	$\ldots(3)$
1	$\neg R \vee \neg F \rightarrow S$	$\ldots(4)$
$4, 2$	$\neg R \vee \neg F \rightarrow T$	$\ldots(5)$
5	$\neg T \rightarrow \neg(\neg R \vee \neg F)$	$\ldots(6)$
$6, 3$	$\neg(\neg R \wedge \neg F)$	$\ldots(7)$
7	$R \wedge F$	$\ldots(8)$
8	R	

Try: Prove or Disprove: All doctors are college graduates. Some doctors are not golfers. Hence, some golfers are not college graduates.

2.4 Rewrite the sentences.

The goal of this exercise is to translate each of the following sentences into First Order Logic (FOL). Later convert the FOL into a prolog program and asking questions.

Input:

(a) Some boys are sharp and intelligent.
UOD(x): all persons.
Sharp(x): x is sharp.
Boy(x): x is a boy.
Intelligent(x): x is intelligent.

(b) Not all boys are intelligent.
(c) Some students of DM course have cleared JEE main and the rest cleared SAT.
UOD(x): all persons.
ClearJEE(x): x clears JEE main.
ClearSAT(x): x clears SAT.
(d) Something that is white is not always milk, whereas the milk is always white.
UOD(x): things.
White(x): x is white.
M ilk(x): x is milk.
(e) Breakfast is served in mess on all days between 7am and 9am except Sunday. And, on Sundays it is served till 9.15 am. UOD(x): days. Day(x): x is a day of the week. Breakfast-time-non-sunday(x): Breakfast is served in mess on x between 7am and 9am. Breakfast-time-sunday(x): Breakfast is served in mess on x till 9.15am.

Output: Equivalent FOL statement.

Hints

$\exists x \text{boys}(x) \wedge \text{intelligent}(x)$

```
# Translate the other statements referring the above
```

Try: Translate each of the following sentences into First Order Logic (FOL):

(a) The speed of light is not same in all mediums. The speed of light in fiber is 2×108 m/s. Therefore, there exists at least two mediums having different speed of light. UOD(x): mediums. Medium(x): Light travels in medium x. Speed(x): Speed of light in medium x. P: Speed of light in fiber is 2 × 108 m/s.
(b) Some students have joined IIITDM. There exists a student who has not joined any IIITDM. Not all students have cleared JEE advanced. Therefore, some students have joined deemed universities. UOD(x) : people. UOD(y) : Educational institutes. Stud(x): x is a student. IIIT DM(y) : y is a IIITDM. JoinIIIT DM(x, y) : x joins IIITDM y. ClearJEE(x) : x cleared JEE advanced. JoinDeemed(x) : x joins a deemed university.

2.5 Propositions

Identify propositions from the following. If not a proposition, justify, why it is not.

(a) I shall sleep or study.
(b) x 2 + 5x + 6 = 0 such that x ∈ integers.

Input: Propositions

Output: Justify the statements either to be a proposition or not

Hints

```
# The rule of logic allows to distinguish between valid and invalid arguments.
Example:
If x+1=5, then x=4=4. Therefore, if x≠4, then x+1≠5.
```

```
If I watch Monday night football, then I will miss the following Tuesday
8 a.m. class. Therefore, if I do not miss my Tuesday 8 a.m. class, then I did not
watch football the previous Monday night.
# Use the same format:
If p then q. Therefore, if q is false then p is false.
If we can establish the validity of this type of argument, then we have proved at
once that both arguments are legitimate. In fact, we have also proved that any
argument using the same format is also credible.
# Use the above example and give the justifications
```

Try: Express the following in first order logic (identify the right universe of discourse, predicates before attempting each question. Think twice and do not oversimplify the problem)

(a) The fundamental law of nature is change.

(b) We cannot help everyone, but everyone can help someone.

(c) Power does not corrupt people, people corrupt power.

(d) It is nice of somebody to do something.

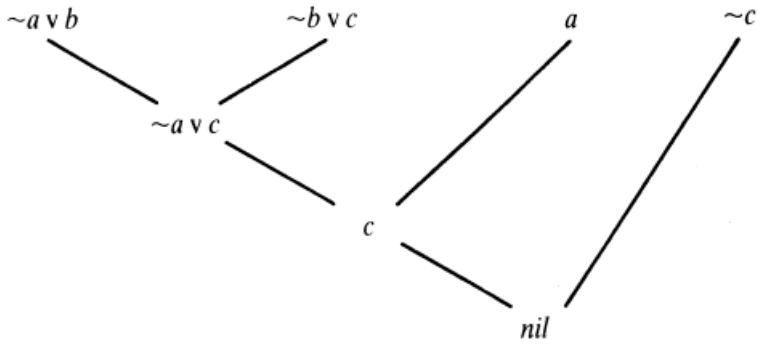
(e) No one who has no complete knowledge of himself will ever have a true understanding of another.

(f) Thought or thinking is what set human beings apart from other living things.

2.5 A Simple Theorem Prover – Resolution Principle

Implement a simple theorem prover as a pattern-directed system by limiting only proving theorem in the simple propositional logic just to illustrate the principle of resolution mechanism. Define the theorem proving as an extendable to handle the first-order predicate calculus.

Input: A formula as a theorem which is always true regardless of the interpretation of the symbols that occur in the formula.



Example:

p v ~p

read as 'p or not p', is always true regardless of the meaning of p. We will be using the following symbols as logic operations:

~ - negation, read as 'not'

& - conjunction, read as 'and'

v – disjunction, read as 'or'

= > - implication, read as 'implies'

Output: Prove that the following propositional formula is a theorem:

(a=>b)&(b=>c)=>(a=>c)

Hints

```
# Write the production rules for resolution theorem proving

# Contradicting the clauses
[ clause(X), clause(~X) ] --->
[ write('Contradiction found'), stop].

# Remove a true clause

# Simply the clause

# Resolution step, a special case

[clause(P), clause(C), delete(~P, C, C1), not done(P, C, P) ] --->
[assert(clause(C1)), assert(done(~P, C, P))].

# Repeat the above step for other special cases and write the last rule as a
resolution process stuck

# delete(P, E, E1) means: delete a disjunction subexpression P from E giving E1
```

```
# in(P,E) means: P is a disjunction subexpression in E

# Write the Translating a propositional formula into(asserted) clauses

# Write the Transformation rules for propositional formulas
```

Try: Implement an interpreter for pattern-directed programs that does not maintain its database as Prolog's own internal database (with assert and retract), but as a procedure argument according to the foregoing remark. Such a new interpreter would allow for automatic backtracking. Try to design a representation of the database that would facilitate efficient pattern matching.

3. Exercises on State Space Search using Prolog

3.1 Water Jug Problem

Given two jugs, a 4-gallon one and a 3-gallon one. Neither has any measuring markers on it. There is a pump that can be used to fill the jugs with water. How can we get exactly 2 gallons of water into a 4-gallon jug?

Input: 4 3 2

Output: {(0,0),(0,3),(3,0),(3,3),(4,2),(0,2)}

a. Describe the state space as a set of ordered pairs of integers.
b. Generate production rules and perform basic operations to achieve the goal.
c. Initialize the start state and apply the rules iteratively until the goal state is reached.
d. Generate a search tree (Depth-First Search / Breadth-First Search)

Hints

```
water(X,Y):- X>4, Y=<3, write('4G Jug is overflow'), nl.
water(X,Y):- X<4, Y>3, write('3G Jug is overflow'), nl.
water(X,Y):- X>=4, Y>=3, write('4G and 3G Jugs are full').

/*step taken: Fill 3G jug.*/
/*step taken: Fill 4G jug.*/

(X=:=2, Y=:=0,nl, write('4L=2 and 3L=0 (step taken: Goal Reached.)'));
(X=:=4, Y=:=0,nl, write('4L=1 and 3L=3 (step taken: Pour water from 4G jugto 3G
jug.)'), XX is X-3, YY is 3, water(XX,YY));
(X=:=0, Y=:=3,nl, write('4L=3 and 3L=0 (step taken: Pour water from 3G jugto 4G
jug.)'), XX is 3, YY is 0, water(XX,YY));

/*Step taken: Empty 3G jug.*/

(X=:=3, Y=:=3,nl, write('4L=3 and 3L=3 (step taken: Fill 3G jug.)'), YY is 3,
water(X,YY));
(X=:=3, Y=:=3,nl, write('4L=4 and 3L=2 (step taken: Pour water from 3G jug to 4G jug
until 4G jug is Full.)'), XX is X+1, YY is Y-1, water(XX,YY));
(X=:=1, Y=:=0,nl, write('4L=0 and 3L=1 (step taken: Pour water from 4G jug to 3G
jug.)'), XX is Y, YY is X, water(XX,YY));
(X=:=0, Y=:=1,nl, write('4L=4 and 3L=1 (step taken: Fill 4G jug.)'), XX is 4,
water(XX,Y));
```

```
/*step taken: Pour water from 4G jug to 3G jug until 3G jug is Full.*/

(X=:=2, Y=:=3,nl, write('4L=2 and 3L=0 (step taken: Empty 3G jug.)'), YY is 0,
water(X,YY));
(X=:=4, Y=:=2,nl, write('4L=0 and 3L=2 (step taken: Empty 4G jug.)'), XX is 0,
water(XX,Y));
(X=:=0, Y=:=2,nl, write('4L=2 and 3L=0 (step taken: Pour water from 3G jug to 4G
jug.)'), XX is Y, YY is X, water(XX,YY)).
```

Try: Given two jugs, a 7-gallon one and a 11-gallon one. Neither has any measuring markers on it. There is a pump that can be used to fill the jugs with water. How can we get exactly 7 gallons of water into a 7-gallon jug?

3.2 Monkey Banana Problem

Imagine a room containing a monkey, chair and some bananas that have been hung from the center of ceiling. If the monkey is clever enough, he can reach the bananas by placing the box directly below the bananas and climbing on the chair .The problem is to prove whether the monkey can reach the bananas. The monkey wants it but cannot jump high enough from the floor. At the window of the room there is a box that the monkey can use.

Input: The monkey can perform the following actions:-

1) Walk on the floor.
2) Climb the box.
3) Push the box around (if it is beside the box).
4) Grasp the banana if it is standing on the box directly under the banana.

Output:

a. Write down the initial state description and action schemes.
b. Prepare all the required predicates that will make the monkey to perform some action and move from one state to the other until the goal state is reached.
c. Set the initial position of the monkey (initial state) and raise questions to whether the knowledge represented can make the monkey get the banana.
d. Trace the flow of actions from initial state to goal state.

Hints

```
move (State1, M, State2)
Clauses:
        on(floor,monkey).
        on(floor,chair).
/* Write the clauses representing the position of a chair and banana */
        in(room, monkey).
        in(room,chair).
        in(room,banana).
        at(ceiling,banana).
        strong(monkey).
        grasp(monkey).
        climb(monkey,chair).
        push(monkey,chair):- strong(monkey).
        under(banana,chair):- push(monkey,chair).
        canreach(banana,monkey):-at(floor,banana);at(ceiling,banana),
under(banana,chair).
```

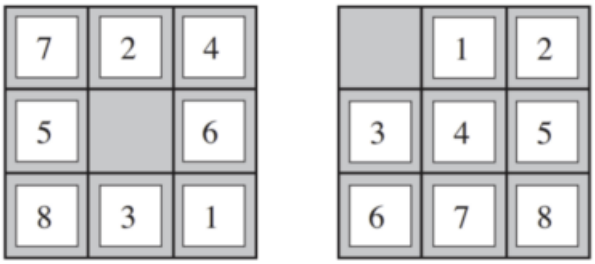
```
        climb(monkey,chair).
        canget(banana,monkey):- canreach(banana,monkey),grasp(money).
```

Try: Extensively make use of state space search to represent and solve Tic-Tac-Toe. Represent the problem as a state space and define the rules. In the state space, represent the starting state, set of legal moves, and the goal state.

3.3 Eight Puzzle Problem

The 8-puzzle consists of a 3×3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The task is to reach a specified goal state, such as the one shown on the right of the figure. The objective is to place the numbers on tiles to match the final configuration using the empty space. You can slide four adjacent (left, right, above, and below) tiles into the empty space.

Input:



Start State Goal State

Output:

a. Examine the problem, formulate all the states and actions to reach the goal.
b. Prepare the production rules that initialize the problem states.
c. Through iterative process determines whether the current and the destination tiles are a valid move.

Hints

```
% Make all arc costs are 1
% Swap Empty and T in L giving Ll
s( [Empty I L], [T I LU, 1) :-
swap( Empty, T, L, L1).
swap( E, T, [T I L], [E I Ll ) :- d( E, T, 1).
swap( E, T, [T1 | Ll, [T1 | LU ) :- swap( E, T, L, L1).

% D is Manh. dist. between two squares
d(X/Y, X1/Y1, D) :- dif( X, Xl, Dx), dif( Y, Y1, Dy), DisDx*Dy.
dif( A, B, D) .- D is A-B, D >:0, !; D is B-A.

/* Calculate the Heuristic estimate h as the sum of distances of each tile from its
home square plus 3 times sequence score. */

/* Set the starting positions for some puzzles */
/* Display a solution path as a list of board positions */
```

```
% Display the board position with X and Y coordinates and perform the backtrack to next square.
```

Try: Extensively make use of state space search to represent and solve the 15 Puzzle problem. Represent the problem as a state space and define the rules.

Start state:

3	10	13	7
9	14	6	1
4		15	2
11	8	5	12

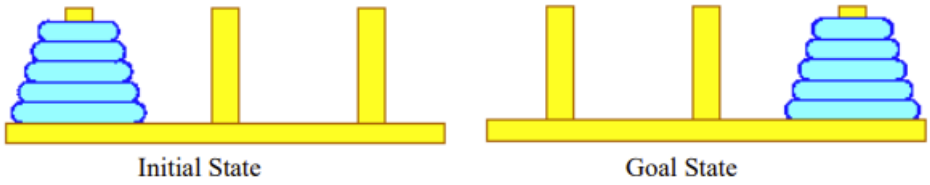
Goal state:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

3.4 The Tower of Hanai

There are three pegs, 1, 2, and 3, and three disks, a, b, and c (a being the smallest and c being the biggest). Initially, all the disks are stacked on peg 1. The problem is to transfer them all on to peg 3. Only one disk can be moved at a time, and no disk can ever be placed on top of a smaller disk.

Input:



Initial State Goal State

Output:

1. Discover the quite simple strategy which will correctly play the Towers of Hanoi game with three poles and N discs.
2. Define a predicate Hanoi having one argument, such that Hanoi(N) means to print the sequence of moves when N discs are on the source pole.
3. Define a predicate that can print the names of the poles that are involved in moving disc.

Hints

```
/* Mention the production Rules including the legal moves */
domains
        loc =right;middle;left
predicates
        hanoi(integer) move(integer,loc,loc,loc) inform(loc,loc)
clauses
hanoi(N):-
        move(N,left,middle,right).
move(1,A,_,C):- inform(A,C),!.
move(N,A,B,C):- N1=N-1,
move(N1,A,C,B), inform(A,C), move(N1,B,A,C). inform(Loc1, Loc2):-
        write("\nMove a disk from ", Loc1, " to ", Loc2).

/* Call the Hanoi function by passing number of pegs */
```

Try: Extensively make use of state space search to represent and solve Maze problem. Represent the problem as a state space and define the rules.

4. Exercises on State Space Search using Prolog

4.1 Blocks Rearrangement Problem

The problem is to find a plan for rearranging a stack of blocks as shown below. We are allowed to move one block at a time. A block can be grasped only when its top is clear. A block can be put on the table or on some other blocks. To find a required plan, we must find a sequence of moves that accomplish the given transformation. Think the problem as a problem of exploring among possible alternatives.

Input:

Output:

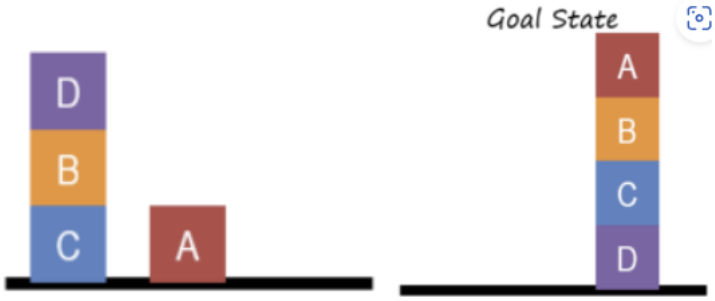
a. Generate the rules involving accomplishing various tasks involving the blocks world.
b. Formulate the more careful definitions for program and the actions.
c. Present the graphical representation of the problem (state space representations) including initial state and the goal state.

Hints

```
/* Mention the prolog actions and plans */
on(a,b).
on(b,c).
on(c,table).
put on(A,B) :- A = table, A ¯ = B, on(A,X), clear(A), clear(B), retract(on(A,X)),
assert(on(A,B)), ¯ assert(move(A,X,B)).
clear(table).
clear(B) :- not(on(X,B)).

/* Represent a state in the blocks world as a list of terms.*/
/* introduce a top-level term that allows us to come up with a plan to go from
an Initial state to a Final state. */

/* Mention all the action clauses that determines the legal actions in a state. */

/* Mention the perform clause for changing the Source state into the Target state by
performing an Action. */

/* Mention the substitute relation to substitute every term with another term to
produce the new term */
```

Try: The problem is to find a plan for rearranging a stack of blocks as shown below. We are allowed to move one block at a time. A block can be grasped only when its top is clear. A block can be put on the

table or on some other blocks. To find a required plan, we must find a sequence of moves that accomplish the given transformation. Think of the problem as a problem of exploring among possible alternatives.



The goal here is to move Block B from the middle of the pile on the left and onto the top of the pile on the right. Hence this sequence of moves would be an acceptable solution:

[("C", "Table"), ("B", "E"), ("C", "A")]

4.2 River Crossing Puzzle

A farmer wants to get a lion, a fox, a goose, and some corn across a river. There is a boat, but the farmer can only take one passenger in addition to himself on each trip, or else both the goose and the corn, or both the fox and the corn. The corn cannot be left with the goose because the goose will eat the corn; the fox cannot be left with the goose because the fox will eat the goose; and the lion cannot be left with the fox because the lion will eat the fox. How does everything get across the river? Assume animals do not wander off when left alone.

Input: Domain specific knowledge including objects and relationship among them.

Output:

a. Represent the search space by giving the starting, ending states and the operations.
b. Draw the first two levels of the search graph. That's two besides the starting state.
c. What is the average branching factor for these two levels? Disregard branches back to previous states.
d. Give an upper bound on the size of the search space.
e. Is this problem decomposable about an intermediate state?

Hints

```
writenlist([]):-
    nl.


writenlist([H|T]):-
    write(H),
    write(' '),
    writenlist(T).


reverse_writenllist([]).


reverse_writenllist([H|T]):-
```

```
    reverse_writenllist(T),
    write(H),
    nl.


member(X,[X|_]).


member(X,[_|T]):-
    member(X,T).


/* function for work with list - END*/


/* change value from e (East) to w (West) */
opposite(e,w).


/* change value from w (West) to e (East) */
opposite(w,e).


/* wolf eats goat */
unsafe(state(X, Y, Y, _)) :-
    opposite(X,Y).


/* goat eats cabbage */
unsafe(state(X, _, Y, Y)) :-
    opposite(X,Y).


/* farmer takes wolf to other side */
move(state(X, X, G, C), state(Y, Y, G, C)):-
    opposite(X,Y),
    not(unsafe(state(Y, Y, G, C))),
    writenlist(['Try farmer takes wolf ', Y, Y, G, C]).


/* farmer takes goat to other side */


/* farmer takes cabbage to other side */
move(state(X, W, G, X), state(Y, W, G, Y)):-
    opposite(X, Y),
    not(unsafe(state(Y, W, G, Y))),
    writenlist(['Try farmer takes cabbage ', Y, W, G, Y]).


/* farmer takes himself to other side */


/* gets here when none of the above fires, system predicate ‘fail’ causes a
backtrack*/
move(state(F, W, G, C), state(F, W, G, C)):-
    writenlist(['    Bactrack from: ', F, W, G, C]),
    fail.
```

```
/* write solution, if start state = goal  */
/* make move */
/* run program */
```

Try: Aside from the river, there is 1 policeman, 1 robber, 1 blond haired woman, and her 2 children and 1 red-haired woman and her 2 children. There is a boat having a carrying capacity of a maximum of 2 people. Only adults can sail but not kids. Please help all people to move across the river, knowing that if the policeman is absent, the robber will kill all people there. If the blond-haired woman is absent, the red-haired woman will beat the blond-haired woman's children (and vice versa).

5. Exercises on Heuristic Search Techniques using Prolog

5.1 Best First Search Algorithms

The Best First Search algorithm is a set of rules that work together to perform a search. It considers the various characteristics of a prioritized queue and heuristic search. The goal of this algorithm is to reach the state of final or goal in the shortest possible time.

Input: Best First Search Algorithm

1. Create 2 empty lists: OPEN and CLOSED
2. Start from the initial node (say N) and put it in the 'ordered' OPEN list.
3. Repeat the next steps until the GOAL node is reached.
   - If the OPEN list is empty, then EXIT the loop returning 'False'.
   - Select the first/top node (say N) in the OPEN list and move it to the CLOSED list. Also, capture the information of the parent node.
   - If N is a GOAL node, then move the node to the Closed list and exit the loop returning 'True'. The solution can be found by backtracking the path.
   - If N is not the GOAL node, expand node N to generate the 'immediate' next nodes linked to node N and add all those to the OPEN list.
   - Reorder the nodes in the OPEN list in ascending order according to an evaluation function f(n)

Output:

a. Perform the search process by using additional information to determine the next step towards finding the solution.

b. Perform the search process using an evaluation function to decide which among the various available nodes is the most promising before traversing to that node.

c. Apply priority queues and heuristic search functions to track the traversal.

Hints

```
best_first([[Goal|Path]|_],Goal,[Goal|Path],0).
best_first([Path|Queue],Goal,FinalPath,N) :-
    extend(Path,NewPaths),
    append(Queue,NewPaths,Queue1),
    sort_queue1(Queue1,NewQueue), wrq(NewQueue),
    best_first(NewQueue,Goal,FinalPath,M),
    N is M+1.
```

```
extend([Node|Path],NewPaths) :-
    findall([NewNode,Node|Path],
            (arc(Node,NewNode,_),
            \+ member(NewNode,Path)), % for avoiding loops
            NewPaths).

sort_queue1(L,L2) :-
    swap1(L,L1), !,
    sort_queue1(L1,L2).
sort_queue1(L,L).

swap1([[A1|B1],[A2|B2]|T],[[A2|B2],[A1|B1]|T]) :-
    hh(A1,W1),
    hh(A2,W2),
    W1>W2.
swap1([X|T],[X|V]) :-
    swap1(T,V).


% Obtain the heuristic value and check if the heuristic function is ok.

wrq(Q) :- length(Q,N), writeln(N).
```

Try: Make use of Heuristic Search principle and apply the best first search to solve the 8-puzzle problem.

5.2 A* Algorithm

A* Algorithm – A square grid is composed of many obstacles that are scattered randomly. The goal is to find the final cell of the grid in the shortest possible time. Implement A* algorithm to search for the shortest path among the given initial and the final state.

Input: n = 2, m = 3, grid[][] = {{0, 2, 1}, {0, 1, 0}}

Output:

a. Initially represent the problem statement as a graph traversal problem.

b. Perform the search process to obtain the shorter path first, thus making it optimal.

c. Find the least cost outcome for the problem by finding all the possible outcomes.

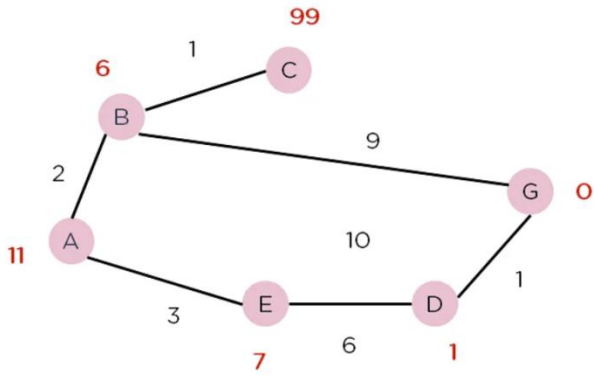
d. Make use of weighted graph by using numbers to represent the cost of taking each path and find the best route with the least cost in terms of distance and time.

Hints

```
# Identify the missing arguments and other logic statements and complete the code.
solve(State,Soln)   :-   f   function(State,0,F),   search([State#0#F#[]],S), reverse(S,Soln).
f_function(State,D,F) :- h function(State,H), F is D + H.
search([State# # #Soln– ], Soln) :- goal(State).
search([B–R],S)    :-    expand(B,Children),    insert    all(Children,R,Open), search(Open,S).
insert     all([F–R],Open1,Open3)     :-     insert(F,Open1,Open2),     insert all(R,Open2,Open3).
insert all([],Open,Open).
insert(B,Open,Open) :- repeat node(B,Open), ! .
```

```
insert(B,[C—R],[B,C—R]) :- cheaper(B,C), ! .
insert(B,[B1—R],[B1—S]) :- insert(B,R,S), !.
insert(B,[],[B]).
repeat node(P# # # , [P# # # — ]).
cheaper( # #F1# , # #F2# ) :- F1 ¡ F2.
expand(State#D# #S,All My Children) :- bagof(Child#D1#F#[Move—S], (D1 is D+1,
move(State,Child,Move), f function(Child,D1,F)), All My Children).
```

Try: Implement the A* algorithm and calculate the shortest distance between the initial and the goal states by considering the following weighted graph:



5.3 AO* Algorithm

Implement the algorithm to generate AND-OR graph or tree to represent the solution by dividing the problem into sub problems and solve them separately to obtain the result by combining all the sub solutions.

Input: The AO* algorithm works on the formula given below : f(n) = g(n) + h(n) where,

- g(n): The actual cost of traversal from initial state to the current state.
- h(n): The estimated cost of traversal from the current state to the goal state.
- f(n): The actual cost of traversal from the initial state to the goal state.

Output:

a. Follow problem decomposition approach and solve each sub problem separately and later combine all the solutions.

b. Traverse the graph starting at the initial node and following the current best path and accumulate the set of nodes that are on the path and have not yet been expanded.

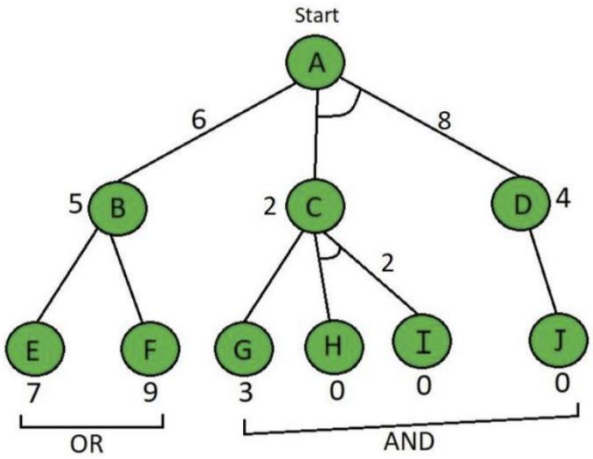
c. Pick one of these best unexpanded nodes and expand it. Add its successors to the graph and compute cost of the remaining distance for each of them.

d. Change the cost estimate of the newly expanded node to reflect the new information produced by its successors. Propagate this change backward through the graph. Decide which of the current best path

Hints

```
% Use to order search
heuristic_distance_to_goal(GoalSituation, Situation, Distance) :-
    ord_subtract(GoalSituation, Situation, Dif),
    length(Dif, Distance).
```

Try: Implement the algorithm to generate AND-OR graph or tree to represent the solution by dividing the problem into sub problems and solve them separately to obtain the result by combining all the sub solutions.

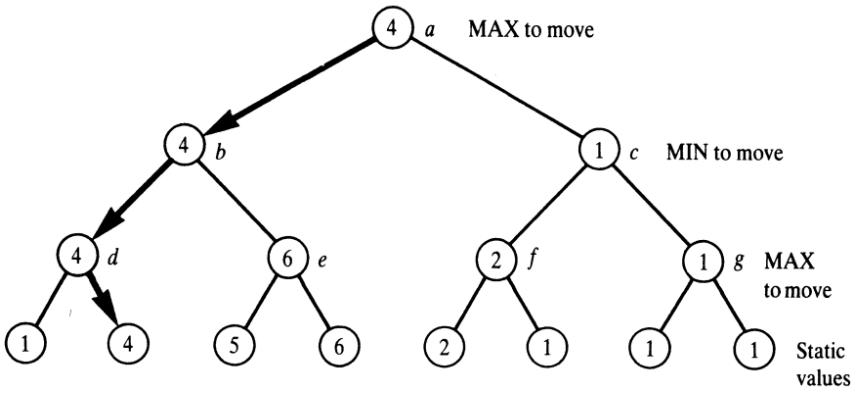


6. Exercises on Adversarial Search Techniques using Prolog

6.1. Minimax Algorithm

As searching game trees exhaustively is not feasible for interesting games, other methods that rely on searching only part of the game tree have been developed. Among these, a standard technique used in computer game playing (chess) is based on the minimax principle. The goal of this exercise is to implement the minimax principle and identify the changes that a player has to with a game.

Input: Game Tree and a Search Tree like:



Output: Make use of static and backup values and estimate the best position from a list of candidate positions.

Hints

```
/* Calculate the heuristic estimator using the estimation function and estimate the changes that a player must win. */
```

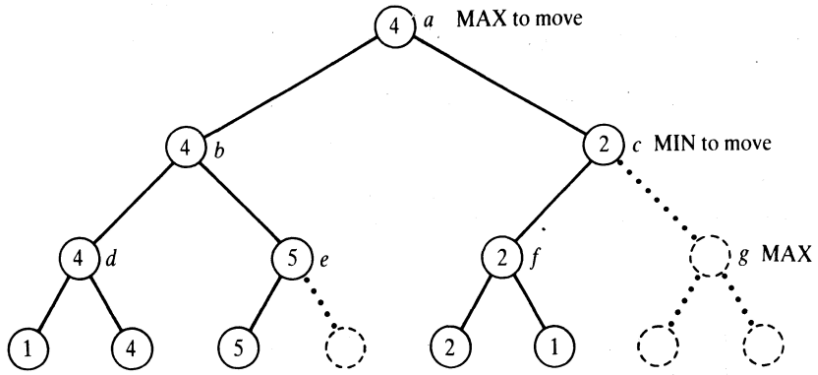
```
/* Implement the procedure as: Minimax procedure: minimax( Pos, BestSucc, Val)  Pos is a position, Val is its minimax value; best move Vo from Pos leads to position BestSuc */
```

Try: Create a knowledge base including the clauses that help in implementing the straightforward method of minimax principle.

6.2. Alpha Beta Pruning

Create a knowledge base representing the straightforward procedure to implement alpha-beta algorithm. Develop a prolog program that systematically visits all the positions in the search tree, up to its terminal positions in a depth-first fashion, and statically evaluates all the terminal positions of this tree.

Input: Search Tree, starting point, legal moves, maximum successors of each move, and apply backtracking wherever applicable.



Output: Compute the exact value of a root position P by setting the bounds as follows:

$$V(P, -infinity, +infinity) = V(P)$$

Hints

```
alphabeta(/* Pass the 4 arguments like position, alpha beta values, good position and the current value */) :-
        moves(Pos, PostList), !,
        boundedbest(Postlist, Alpha, Beta, GoodPos, Val);
        staticval(Pos, Val).

boundedbest( [Pos | Poslist], Alpha, Beta, GoodPos, GoodVal) :-
        alphabeta( Pos, Alpha, Beta, _, Val),
        goodenough( Poslist, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough( /* Mention an empty list */, -, -, Pos, Val, Pos, Val) '- !. % No other candidate

goodenough( -, Alpha, Beta, Pos, Val, Pos, Val) :-
        min-to-rnove( Pos), Val ) Beta, !; % Maximizer attained upper bound
        marto-mov{ Pos), Val ( Alpha, !. % Minimizer attained lower bound

goodenough( Poslist, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
        newbounds( /* Pass the 6 arguments like alpha beta values, good position, current value, new alpha and beta values */), % Refine bounds boundedbes( Poslist, NewAlpha, NewBeta, Posl, Vall),
        betterof( Pos, Val, Posl, Vall, GoodPos, GoodVal).
```

```
newbounds( Alpha, Beta, Pos, Val, Val, Beta) :-
        min-to-rnove( Pos), Vd > Alpha, !. % Maximizer increased lower bound

newbounds( Alpha, Beta, Pos, Val, Alpha, Vat) :-
        marto-rnove( Pos), Val < Beta, !. % Minimizer decreased upper bound

newbounds( Alpha, Beta, -, -, Alpha, Beta)

betterof( Pos, Val, Posl, Val1, Pos, Val) :-
        min-to-rnove( Pos), Vd > Vall, !;
        marto:nove( Pos), Val < Vall, !.

betterof( -, -, Pos1, Val1, Posl, Vall).
```

Try: Consider a two-person game (for example, some non-trivial version of tic-tac-toe). Write game-definition relations (legal moves and terminal game positions) and propose a static evaluation function to be used for playing the game with the alpha-beta procedure. Use the principle of alpha beta to reduce the search in the tree mentioned above.

6.3. Iterative Deepening Techniques

The goal is to prove that search is ubiquitous in artificial intelligence. The performance of most AI systems is dominated by the complexity of a search algorithm in their inner loops. Prove with an example that this algorithm gives optimal solution for exponential tree searches.

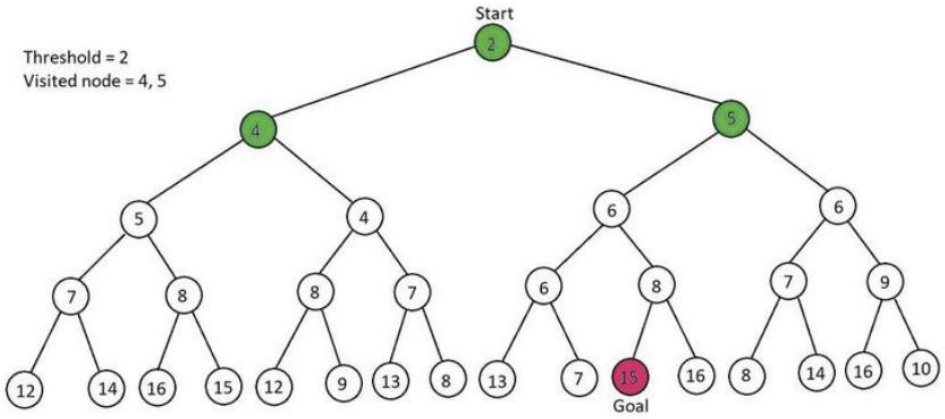
Input:

Output:

a. Complete the search process if the branching factor is finite and there is a solution at some finite depth and obtain optimal in finding the shortest solution first.
b. Avoid exploring each non-solution branch of the tree, omit cycle detection and retain completeness.
c. Use additional logical features of Prolog to terminate the search process whenever if there are no solutions identified even after backtracking.
d. Document the steps if the search process does not obtain optimal solution even after backtracking.

Hints

Try: The 15-puzzle problem is a classic example of a **sliding puzzle game**. It consists of a 4×4 grid of numbered tiles with one tile missing. The aim is to rearrange the tiles to form a specific goal configuration. The state space of the puzzle can be represented as a tree where each node represents a configuration of the puzzle, and each edge represents a legal move. IDA* can be used to find the **shortest sequence of moves** to reach the goal state from the initial state.



7. Exercises on Expert Systems using Prolog

7.1 Identify Animals

The goal is to create an expert system that can identify animals. We can use the rules of inference that we have learned about animals to perform this task. These rules serve as a starting point for developing an expert system, and they show the importance of having input from the users. The goal of an expert system is to provide useful information based on its users' inputs.

Input:

- If it has a tawny color and has dark spots, then the animal is a cheetah.
- If it has a tawny color and has black stripes, then the animal is a tiger.
- If it has a long neck and has long legs, then the animal is a giraffe.
- If it has black stripes, then the animal is a zebra.
- If it does not fly and has long neck, then the animal is an ostrich.
- If it does not fly, swims, black and white in color, then the animal is penguin.
- If it appears in story ancient mariner and flys well, then the animal is albatross.

Output:

a. Create an expert system that can identify the animal class using the inference rules.
b. Utilize the user inputs and predict the animal class based on the behaviors already learned by the expert system.

Try: Consider the if-then rules of figures and translate them into our rule notation. Propose extensions to the notation to handle certainty measures when needed.

if

1 the infection is primary bacteremia, and
2 the site of the culture is one of the sterilesites, and
3 the suspected portal of entry of the organism is the gastrointestinal tract

then

there is suggestive evidence (0.7) that the identity of the organism is bacteroides.

if

1 there is a hypothesis, H, that a plan P succeeds, and
2 there are two hypotheses,
H_1, that a plan R_1 refutes plan P, and
H_2, that a plan R_2 refutes plan P, and
3 there are facts: H_1 *is false*, and H_2 *is false*

then

1 generate the hypothesis, H_3, that the combined plan 'R_1 *or* R_2' refutes plan P, and
2 generate the fact: H_3 *implies not(H)*

7.2 Locating Failures in a Simple Electric Network

Create a knowledge base which can help locating failures in a simple electric network that consists of some electric devices and fuses. Such a network is shown in figure.

---

if

light1 is on *and*
light1 is not working *and*
fuse1 is proved intact

then

light1 is proved broken.

Another rule can be:

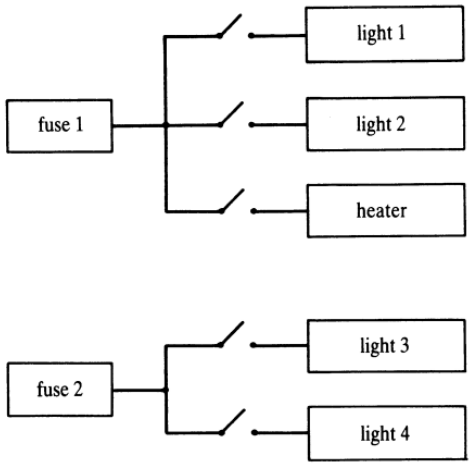
if

heater is working

then

fuse1 is proved intact.

Input:



Output:

```
% A small knowledge base for locating faults in an electric network
% If a device is on and not working and its fuse is intact then the device is broken

broken_rule:
        if
                on(Device) and
                device(Device) and
                not working(Device) and
                connected(Device, Fuse) and
                proved(intact(Fuse))
        then
                proved(broken(Device))
% If a unit is working then its fuse is OK
fuse_ok_rule:
        if
                connected(Device,Fuse) and
                working(Device)
        then
                proved(intact(Fuse)).

% If two different devices are connected to a fuse and are both on and not working
% then the fuse has failed.
% NOTE: This assumes that at most one device is broken!

fused_rule:
        if
                connected(Device1, Fuse) and
```

```
                on(Device1) and
                not working(Device1) and
                samefuse(Device2, Device1) and
                on(Device2) and
                not working(Device2)
        then
                proved(failed(Fuse)).
same_fuse_rule:
        if
                connected(Device1, Fuse) and
                connected(device2, Fuse) and
                different(Device1, Device2)
        then
                samefuse(Device1, Device2).
fact: different(X, Y) :- not(X=Y).

fact: device(heater).
fact(device(light1).
fact(device(light2).
fact(device(light3).
fact(device(light4).

fact: connected(light1, fuse1).
fact: connected(light2, fuse1).
fact: connected(heater, fuse1).
fact: connected(light3, fuse2).
fact: connected(light4, fuse2).

askable(on(D), on('Device')).
askable(working(D), working('Device')).
```

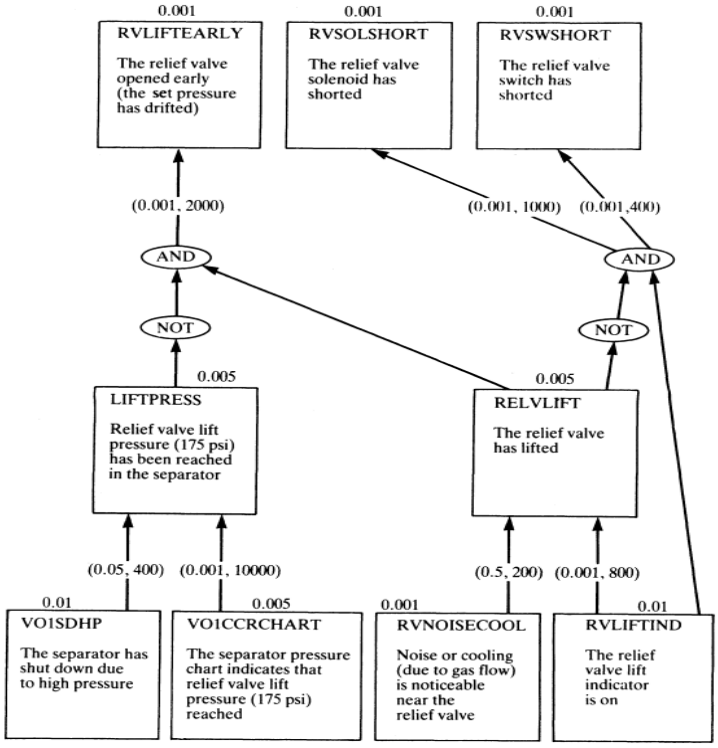
Try: Think of some decision problem and try to formulate the corresponding knowledge in the form of if-then rules. You may consider choice of holiday, weather prediction, simple medical diagnosis, and treatment, etc.

7.3 Dealing with Uncertainty

Our expert system shell of the previous section only deals with questions that are either true or false. Such domains in which all answers are reduced to true or false are called categorical. As data, rules were also categorical: 'categorical implications'. However, many expert domains are not categorical. Typical expert behaviour is full of guesses (highly articulated, though) that are usually true, but there can be exceptions. Both data about a particular problem and implications in general rules can be less than certain. We can model uncertainty by assigning to assertions some qualification other than just true and false. Such

qualification can be expressed by descriptors-for example, true highly likely, likely, unlikely, impossible. The goal of this exercise is to implement the Prospector Model to measure the likelihood of events by modelling them by real numbers between 0 and 1. Finding the likelihood of a hypothesis by certainty propagation in an inference network of the Prospector-Al/X type.

Input:



Output: Explore the Goal, Trace, and the Answer. Find the likelihood measure that Goal is true, Trace the chain of ancestors, goals, and rules that can be used for 'why' explanation.

Hints

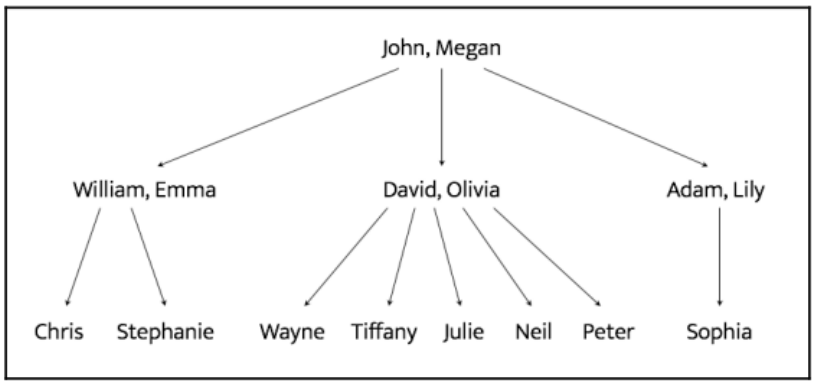
Try:

Complete our shell that deals with uncertainties (add a corresponding **useranswer** and other procedures). Consider critical comments and possible extensions to our expert system shell, as discussed, and design and implement corresponding improvements.

8. Exercises on Logic Programming using Python

8.1 Parsing a Family Tree

Use the most familiar Logic programming and solve an interesting problem of Parsing a Family Tree by considering the following diagram. John and Megan have three sons – William, David, and Adam. The wives of William, David, and Adam are Emma, Olivia, and Lily respectively. William and Emma have two children – Chris and Stephanie. David and Olivia have five children – Wayne, Tiffany, Julie, Neil, and Peter. Adam and Lily have one child – Sophia. Based on these facts, create a program that can tell us the name of Wayne's grandfather or Sophia's uncles are.



Input: A .json file specified with the relationship among all the people involved.

Output: Ask the following questions to understand if our solver can come up with the right answer.

1. Who John's children are?
2. Who is William's mother?
3. Who are Adam's parents?
4. Who are Wayne's grandparents?

Hints

```
/*Create a new Python file and import the following packages: */
import json
from logpy import Relation, facts, run, conde, var, eq

/* Define a function to check if x is the parent of y. We will use the logic that if
x is the parent of y, then x is either the father or the mother. We have already
defined “father” and “mother” in our fact base: */

/* Define a function to check if x is the grandparent of y. We will use the logic
that if x is the grandparent of y, then the offspring of x will be the parent of y:
*/

# Check for sibling relationship between 'a' and 'b'
def sibling(x, y):
        temp = var()
        return conde((parent(temp, x), parent(temp, y)))

/* Define the main function and initialize the relations father and mother */
/* Load the data from the relationships.json file */
/* Read the data and add them to our fact base */
Define the variable x:
x = var()

/* We are now ready to ask some questions and see if our solver can come up with the
right answers. Let's ask who John's children are:

# John's children
name = 'John'
output = run(0, x, father(name, x))
print("\nList of " + name + "'s children:")
for item in output:
        print(item)
```

```
/* Start asking the questions like: Who is William's mother?, Who are Adam's parents?, Who are Wayne's grandparents?, Who are Megan's grandchildren?, Who are David's siblings?, Who are Tiffany's uncles? */
/* List out all the spouses in the family */
```

Try:

1. Who are Megan's grandchildren?
2. Who are David's siblings?
3. Who are Tiffany's uncles?
4. List out all the spouses in the family.

8.2 Analyzing Geography

Use logic programming to build a solver to analyze geography. In this problem, specify information about the location of various states in the US and then query our program to answer various questions based on those facts and rules. The following is a map of the US:

Input:

1. Define the input files to load the data from.
2. Read the files containing the coastal states.
3. Add the adjacency information to the fact base.
4. Initialize the variables x and y.

Output:

1. Print out all the states that are adjacent to Oregon.
2. List all the coastal states that are adjacent to Mississippi.
3. List all the coastal states that are adjacent to Mississippi.

```
Is Nevada adjacent to Louisiana?:
No

List of states adjacent to Oregon:
Washington
California
Nevada
Idaho

List of coastal states adjacent to Mississippi:
Alabama
Louisiana

List of 7 states that border a coastal state:
Georgia
Pennsylvania
Massachusetts
Wisconsin
Maine
Oregon
Ohio
```

Hints

```
/*Create a new Python file and import the following: */

from logpy import run, fact, eq, Relation, var

/*Initialize the relations: */

adjacent = Relation() coastal = Relation()

/*Define the input files to load the data from: */

file_coastal = 'coastal_states.txt'
file_adjacent = 'adjacent_states.txt'

/*Load the data: # Read the file containing the coastal states */
with open(file_coastal, 'r') as f:
        line = f.read()
        coastal_states = line.split(',')

/*Add the information to the fact base:
# Add the info to the fact base
for state in coastal_states:
        fact(coastal, state)

/* Read the adjacency data */

/* Add the adjacency information to the fact base */

/* Initialize the variables x and y */

/* Check if Nevada is adjacent to Louisiana */

/* Print out all the states that are adjacent to Oregon, List all the coastal states
that are adjacent to Mississippi, List seven states that border a coastal state, List
states that are adjacent to both Arkansas and Kentucky */
```

Try: Add more questions like "list states that are adjacent to both Arkansas and Kentucky" to the program to see if it can answer them.

8.2 Building a Puzzle Solver

The interesting application of logic programming is in solving puzzles. The goal of this exercise is to specify the conditions of a puzzle and the program has to come up with a solution. Also specify various bits and pieces of information about four people and ask for the missing piece of information.

Input: In the logic program, we specify the puzzle as follows:

- Steve has a blue car.
- The person who owns the cat lives in Canada Matthew lives in USA.
- The person with the black car lives in Australia.
- Jack has a cat.
- Alfred lives in Australia.
- The person who has a dog lives in France.
- Who has a rabbit?

Output: The goal is the find the person who has a rabbit. Here are the full details about the four people:

Name	Pet	Car color	Country
Steve	dog	blue	France
Jack	cat	green	Canada
Matthew	rabbit	yellow	USA
Alfred	parrot	black	Australia

Hints

```
/*Create a new Python file and import the following packages: */
from logpy import *
from logpy.core
import lall

/* Declare the variable people: */
# Declare the variable
people = var()

/* Define all the rules using lall, The first rule is that there are four people, The
person named Steve has a blue car */

/* The person who has a cat lives in Canada, The person named Matthew lives in USA,
The person who has a black car lives in Australia, The person named Jack has a cat,
The person named Alfred lives in Australia, The person who has a dog lives in France,
The person who has a dog lives in France.*/

/* Run the solver with the preceding constraints */

/* Extract the output from the solution */

/* Print the full matrix obtained from the solver */
```

Try: Demonstrate how to solve a puzzle with incomplete information. You can play around with it and see how you can build puzzle solvers for various scenarios.

9. Exercises on Heuristic Search Techniques using Python

9.1 Constructing a String using Greedy Search

Recreate the input string based on the alphabets using a greedy search. Ask the algorithm to search the solution space and construct a path to the solution.

Input:

1. A function to parse the input arguments.
2. A class (SearchProblem) that contains the methods needed to solve the problem.
3. Check the current state and take the right action.
4. Concatenate state and action to get the result.
5. Check if the goal has been achieved.
6. The heuristic that will be used.
7. Initialize the CustomerProblem object.
8. Set the starting point and the goal we want to achieve.
9. Run the solver and
10. Print the path to the solution.

Output: Calculate how far we are from the goal and use that as the heuristic to guide it towards the goal.

1. Run the code with an empty initial state.
2. Run the code with a non-empty starting point.

```
Path to the solution:
(None, '')
('A', 'A')
('r', 'Ar')
('t', 'Art')
('i', 'Arti')
('f', 'Artif')
('i', 'Artifi')
('c', 'Artific')
('i', 'Artifici')
('a', 'Artificia')
('l', 'Artificial')
(' ', 'Artificial ')
('I', 'Artificial I')
('n', 'Artificial In')
('t', 'Artificial Int')
('e', 'Artificial Inte')
('l', 'Artificial Intel')
('l', 'Artificial Intell')
('i', 'Artificial Intelli')
('g', 'Artificial Intellig')
('e', 'Artificial Intellige')
('n', 'Artificial Intelligen')
('c', 'Artificial Intelligenc')
('e', 'Artificial Intelligence')
```

```
Path to the solution:
(None, 'Artificial Inte')
('l', 'Artificial Intel')
('l', 'Artificial Intell')
('i', 'Artificial Intelli')
('g', 'Artificial Intellig')
('e', 'Artificial Intellige')
('n', 'Artificial Intelligen')
('c', 'Artificial Intelligenc')
('e', 'Artificial Intelligence')
(' ', 'Artificial Intelligence ')
('w', 'Artificial Intelligence w')
('i', 'Artificial Intelligence wi')
('t', 'Artificial Intelligence wit')
('h', 'Artificial Intelligence with')
(' ', 'Artificial Intelligence with ')
('P', 'Artificial Intelligence with P')
('y', 'Artificial Intelligence with Py')
('t', 'Artificial Intelligence with Pyt')
('h', 'Artificial Intelligence with Pyth')
('o', 'Artificial Intelligence with Pytho')
('n', 'Artificial Intelligence with Python')
```

Hints

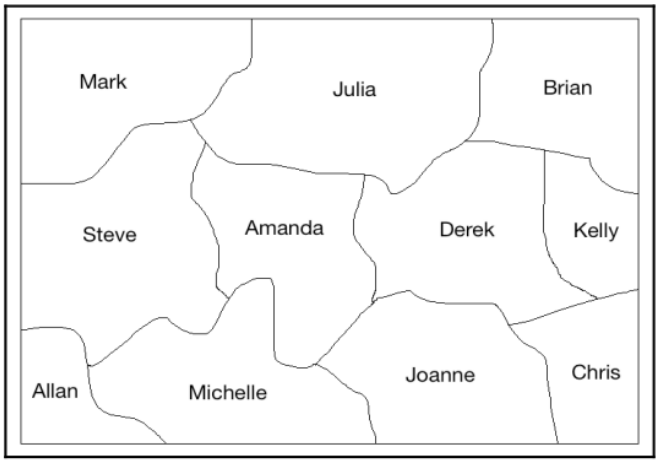
```
/*Create a new Python file and import the following packages: */
import argparse
import simpleai.search as ss

/* Define a function to parse the input arguments */
```

```
class CustomProblem(ss.SearchProblem):
        def set_target(self, target_string):
                self.target_string = target_string

# Check the current state and take the right action
def actions(self, cur_state):
        if len(cur_state) < len(self.target_string):
                alphabets = 'abcdefghijklmnopqrstuvwxyz'
                return list(alphabets + ' ' + alphabets.upper())
        else:
                return []

# Concatenate state and action to get the result
def result(self, cur_state, action):
        return cur_state + action

# Check if goal has been achieved
def is_goal(self, cur_state):
        return cur_state == self.target_string

/* Initialize the CustomProblem object */

/* Set the starting point as well as the goal we want to achieve */

Run the solver:
# Solve the problem
output = ss.greedy(problem)

/* Print the path to the solution */
```

Try: Solve the same problem with constraints. Specify three constraints as follows: John, Anna, and Tom should have different values. Tom's value should be bigger than Anna's value If John's value is odd, then Patricia's value should be even and vice versa.

9.2 Solving a Region Coloring Problem

Consider the following screenshot:



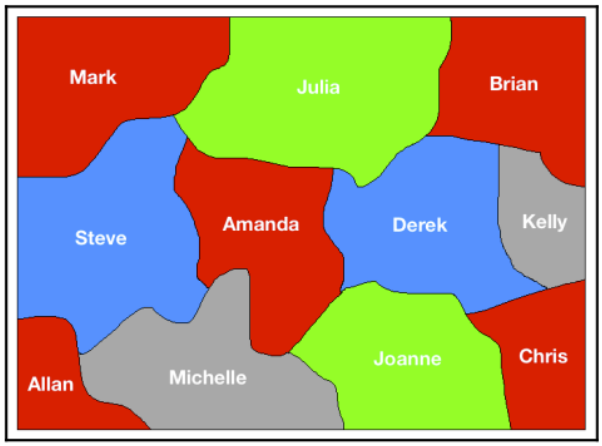
We have a few regions in the preceding figure that are labeled with names. Our goal is to color with four colors so that no adjacent regions have the same color. Make use of Constraint Satisfaction framework to solve the region-coloring problem.

Input:

1. Constraints that specify different values
2. Main function and a list of names.
3. List of possible colors

Output:

```
Color mapping:

Derek ==> blue
Michelle ==> gray
Allan ==> red
Steve ==> blue
Julia ==> green
Amanda ==> red
Joanne ==> green
Mark ==> red
Kelly ==> gray
Brian ==> red
```



Hints

```
/* Create a new Python file and import the following packages:*/
from simpleai.search import CspProblem, backtrack

/* Define the constraint that specifies that the values should be different: */
/* # Define the function that imposes the constraint */
/* # that neighbors should be different */
def constraint_func(names, values):
        return values[0] != values[1]

/* Define the main function and specify the list of names: */

if __name__=='__main__':
# Specify the variables names = ('Mark', 'Julia', 'Steve', 'Amanda', 'Brian',
'Joanne', 'Derek', 'Allan', 'Michelle', 'Kelly')

/*Define the list of possible colors */

/* convert the map information into something that the algorithm can understand */

/* Use the variables and constraints to initialize the object */

/* Solve the problem and print the solution */
```

Try:

9.3 Building an 8 Puzzle Solver

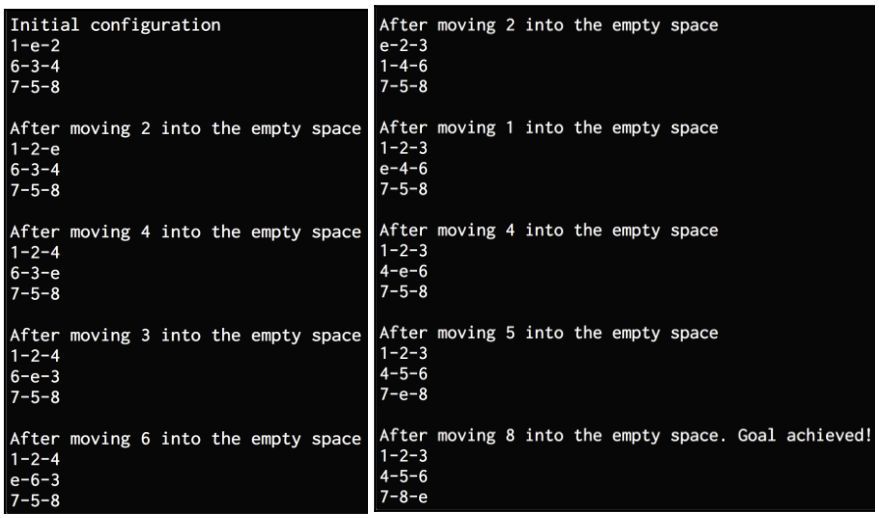
8-puzzle is a variant of the 15-puzzle. You can check it out at https://en.wikipedia.org/wiki/15_puzzle. You will be presented with a randomized grid and your goal is to get it back to the original ordered configuration. You can play the game to get familiar with it at http://mypuzzle.org/sliding. The goal is to use A* algorithm to solve this problem and find the paths to the solution in a graph.

Input:

1. A class that contains the methods.
2. Action method to get the list of the possible numbers.
3. Check the location of the empty space and create a new action.

Output:

1. Return the resulting state after moving a piece to the empty space.
2. Computes the distance between the current state and goal state using Manhattan distance.

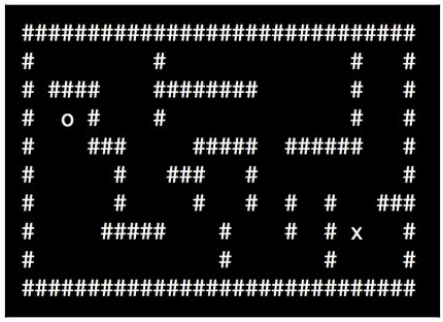


Hints

```
/* Create a new Python file and import the following packages */
from simpleai.search import astar, SearchProblem

/* Define a class that contains the methods to solve the 8-puzzle: */
# Class containing methods to solve the puzzle class PuzzleSolver(SearchProblem):

/* Override the actions method to align it with our problem: */
# Action method to get the list of the possible
# numbers that can be moved in to the empty space
def actions(self, cur_state):
        rows = string_to_list(cur_state)
        row_empty, col_empty = get_location(rows, 'e')

/* Check the location of the empty space and create the new action */

/* Override the result method. Convert the string to a list and extract the location
of the empty space. */

/* # Return the resulting state after moving a piece to the empty space */

/* Check if the goal has been reached */

/* Define the heuristic method. */
```

```
/* Compute the distance */

/* Define a function to convert a list to string */

/* Define a function to convert a string to a list */

/* Define a function to get the location of a given element in the grid */

/* Define the initial state and the final goal we want to achieve */

/* Track the goal positions for each piece by creating a variable */

/* Create the A* solver object using the initial state we defined earlier and extract
the result */

/* Print the solution */
```

Try: Use the A* Algorithm to solve a maze. Consider the below figure to build a maze solve.



10. Exercises on Genetic Algorithms using Python

10.1 Generating a Bit Pattern using Predefined Parameters

Generate a bit string that contains a predefined number of ones. Perform the selection process during each iteration by applying the genetic algorithm. Choose the strongest individuals and terminate the weakest one where the survival of the fittest concept comes into play. Carry out the selection process by using a fitness function and compute the strength of each individual.

Input:

1. Deap Python Library
2. Consider the problem as solving the variant of the One Max problem.

Output:

1. Generate a bit string that contains a predefined number of ones.
    a. Generate a bit pattern of length 75.
2. Evaluate all the individuals in the population using the fitness function.
3. Evaluate all the individuals with invalid fitness values.
4. Print the stats for the current generation to see how its progressing.

```
Starting the evolution process

Evaluated 500 individuals

===== Generation 0
Evaluated 297 individuals
Min = 58.0 , Max = 75.0
Average = 70.43 , Standard deviation = 2.91

===== Generation 1
Evaluated 303 individuals
Min = 63.0 , Max = 75.0
Average = 72.44 , Standard deviation = 2.16

===== Generation 2
Evaluated 310 individuals
Min = 65.0 , Max = 75.0
Average = 73.31 , Standard deviation = 1.6

===== Generation 3
Evaluated 273 individuals
Min = 67.0 , Max = 75.0
Average = 73.76 , Standard deviation = 1.41
```

```
===== Generation 57
Evaluated 306 individuals
Min = 68.0 , Max = 75.0
Average = 74.02 , Standard deviation = 1.27

===== Generation 58
Evaluated 276 individuals
Min = 69.0 , Max = 75.0
Average = 74.15 , Standard deviation = 1.18

===== Generation 59
Evaluated 288 individuals
Min = 69.0 , Max = 75.0
Average = 74.12 , Standard deviation = 1.24

==== End of evolution

Best individual:
 [1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1
, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0
, 1, 0, 0, 0, 1]

Number of ones: 45
```

Hints

```
/* Install the Python package like DEAP */
$ pip3 install deap

/* Create a new Python file and import the following: */
import random from deap
import base, creator, tools

# Evaluation function
def eval_func(individual):
        target_sum = 45
        return len(individual) - abs(sum(individual) - target_sum),

# Create the toolbox with the right parameters
# Initialize the toolbox
# Generate attributes
# Initialize structures
# Define the population to be a list of individuals
# Register the evaluation operator
# Register the crossover operator
# Register a mutation operator
# Operator for selecting individuals for breeding

/* Write the appropriate code here by printing the stats for the current generation
individuals and print the final output */
```

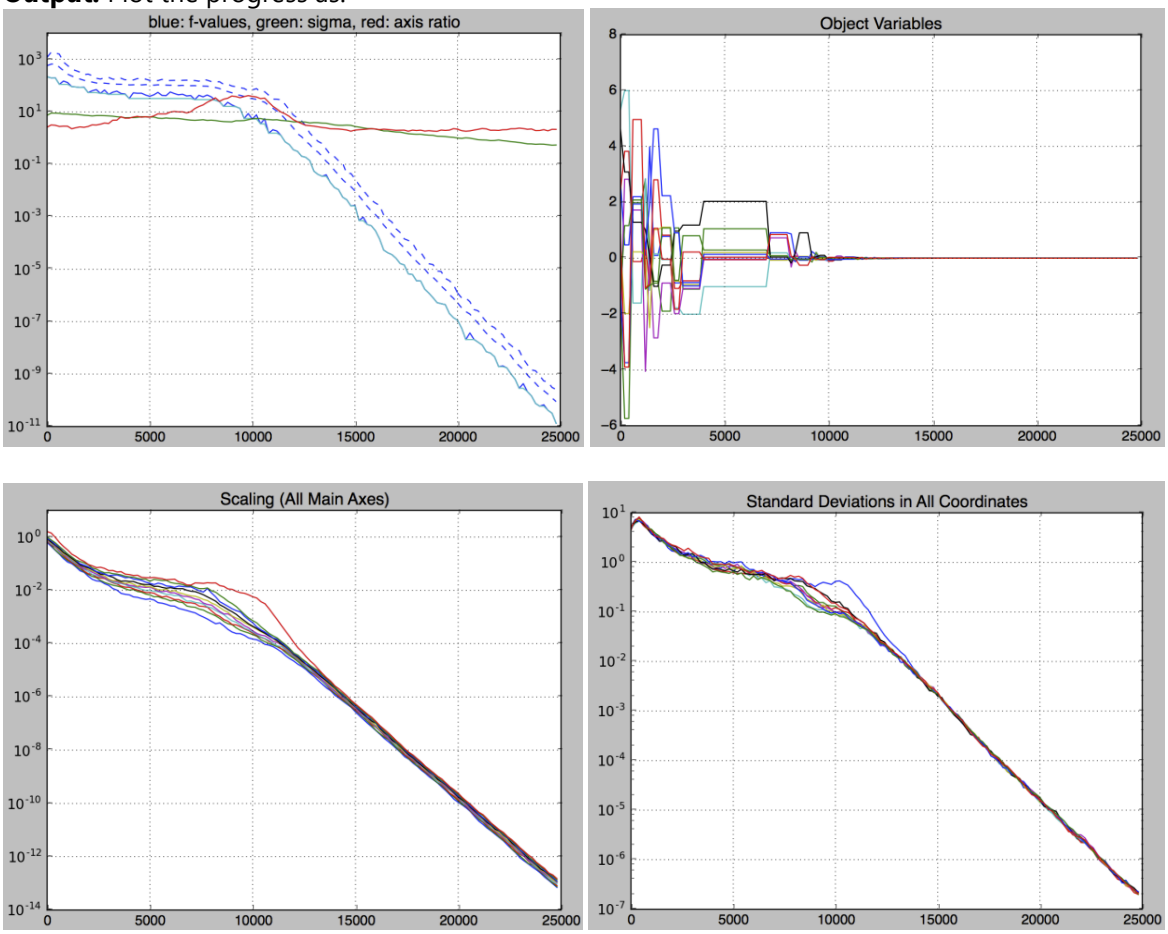
Try:

10.2 Visualizing the Evolution

Understand the process of visualizing the evolution process by using a CMA-ES to solve non-linear problems in the continuous domain. The goal of this exercise is to work by delving into the code provided in their source code by using DEAP library.

Input:

Output: Plot the progress as:



Hints

```
/*Create a new Python file and import the following: */
import numpy as np
import matplotlib.pyplot as plt from deap
import algorithms, base, benchmarks, \ cma, creator, tools

/* Define a function to create the toolbox. We will define a FitnessMin function
using negative weights: */

/* Create the toolbox and register the evaluation function */
```

```
/* Register the generate and update methods, Define the main function, define a strategy before we start the process */

/* Create the toolbox based on the strategy */

/* Register the stats using the Statistics method */

/* Define objects to compile all the data, define objects to compile all the data, and Evaluate individuals using the fitness function */

/* Update the strategy based on the population and save the data for plotting*/

/* Define the x axis and plot the stats and plot the progress */
```

Try: Extend the same program to see the progress printed on the Terminal. Observe the values keep decreasing as the process progress and indicate that it's converging.

10.3 Solving the Symbol Regression Problem

Use genetic programming to solve the symbol regression problem and understand that genetic programming is not the same as genetic algorithms. The goal of this exercise is to understand how the programs are modified, at each iteration.

Input: Create a division operator, define the evaluation function, and compute the mean squared error (MSE).

Output: Run the evolutionary algorithm using the above parameters:

```
population, log = algorithms.eaSimple(population, toolbox,
        probab_crossover, probab_mutate, num_generations,
        stats=mstats, halloffame=hall_of_fame, verbose=True)
```

Hints

```
/* Create a new Python file and import the following: */
import operator import math
import random
import numpy as np
from deap import algorithms, base, creator, tools, gp

/* Create a division operator that can handle divide-by-zero error gracefully: */

/* Create a division operator that can handle divide-by-zero error gracefully: */

/* Compute the mean squared error (MSE) between the function defined earlier and the original expression: */

/* Define a function to create the toolbox and register the stats using the objects defined previously. */

/* Define the crossover probability, mutation probability, and the number of generations */
```

```
/* Define the crossover probability, mutation probability, and the number of generations */
```

		fitness				size			
gen	nevals	avg	max	min	std	avg	max	min	std
0	450	18.6918	47.1923	7.39087	6.27543	3.73556	7	2	1.62449
1	251	15.4572	41.3823	4.46965	4.54993	3.80222	12	1	1.81316
2	236	13.2545	37.7223	4.46965	4.06145	3.96889	12	1	1.98861
3	251	12.2299	60.828	4.46965	4.70055	4.19556	12	1	1.9971
4	235	11.001	47.1923	4.46965	4.48841	4.84222	13	1	2.17245
5	229	9.44483	31.478	4.46965	3.8796	5.56	19	1	2.43168
6	225	8.35975	22.0546	3.02133	3.40547	6.38889	15	1	2.40875
7	237	7.99309	31.1356	1.81133	4.08463	7.14667	16	1	2.57782
8	224	7.42611	359.418	1.17558	17.0167	8.33333	19	1	3.11127
9	237	5.70308	24.1921	1.17558	3.71991	9.64444	23	1	3.31365
10	254	5.27991	30.4315	1.13301	4.13556	10.5089	25	1	3.51898

Try: Building two bots to play Hexapawn against each other.

11. Exercises on Search Algorithms in Games using Python

11.1 Building a bot to play Last Coin Standing

This is a game where we have a pile of coins, and each player takes turns to take a number of coins from the pile. There is a lower and an upper bound on the number of coins that can be taken from the pile. The goal of the game is to avoid taking the last coin in the pile.

Input: A class handling all the operations of the game, define who starts the game, overall number of coins in the pile, maximum number of coins per move, and possible moves.

Output:

```
d:2, a:0, m:1
d:3, a:0, m:1
d:4, a:0, m:1
d:5, a:0, m:1
d:6, a:0, m:1
d:7, a:0, m:1
d:8, a:0, m:1
d:9, a:0, m:1
d:10, a:100, m:4
1 10 4
25 coins left in the pile

Move #1: player 1 plays 4 :
21 coins left in the pile

Player 2 what do you play ? 1

Move #2: player 2 plays 1 :
20 coins left in the pile

Move #3: player 1 plays 4 :
16 coins left in the pile
```

```
Move #5: player 1 plays 2 :
11 coins left in the pile

Player 2 what do you play ? 4

Move #6: player 2 plays 4 :
7 coins left in the pile

Move #7: player 1 plays 1 :
6 coins left in the pile

Player 2 what do you play ? 2

Move #8: player 2 plays 2 :
4 coins left in the pile

Move #9: player 1 plays 3 :
1 coins left in the pile

Player 2 what do you play ? 1

Move #10: player 2 plays 1 :
0 coins left in the pile
```

Hints

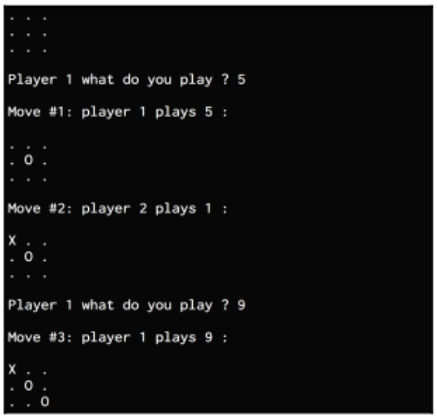
Try:

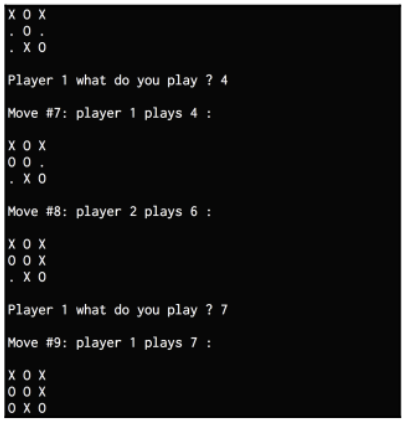
11.2 Building a bot to play Tic-Tac-Toe.

Tic-Tac-Toe (Noughts and Crosses) is probably one of the most famous games. Here the goal is to build a game where the computer can play against the user.

Input: A 3×3 board numbered from one to nine row-wise, all possible moves, and updates the board moves until some player has lost the game.

Output:





Hints

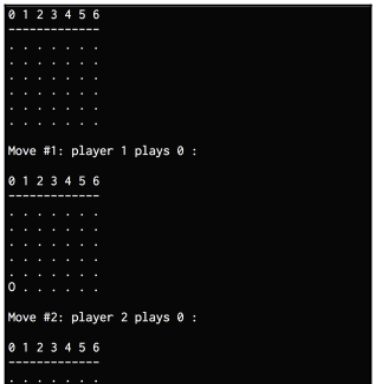
```
# Create a new Python file and import the following packages:
from easyAI import TwoPlayersGame, AI_Player, Negamax
from easyAI.Player import Human_Player

# Define a class that contains all the methods to play the game. Start by defining
the players and who starts the game

# Define a method to compute all the possible moves

# Define a method to update the board after making a move

# Define a method to see if somebody has lost the game

# Define a method to show the current progress and compute the score using the
loss_condition method
```

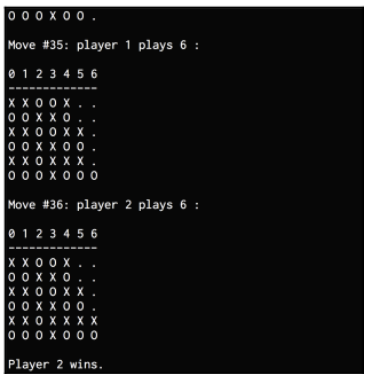
11.3 Building Two Bots to Play Connect Four against each other.

Connect Four™ is a popular two-player game sold under the Milton Bradley trademark. It is also known by other names such as Four in a Row or Four Up. In this game, the players take turns dropping discs into a vertical grid consisting of six rows and seven columns. The goal is to get four discs in a line. This is a variant of the Connect Four recipe given in the easyAI library. In this recipe, instead of playing against the computer, create two bots that will play against each other. We will use a different algorithm for each to see which one wins.

Input: A board with six rows and seven columns, define the who is going to start the games, define the positions, and define all the possible legal moves.

Output: Compute the score and print the results. Get the following output on your Terminal at the beginning and towards the end.





Hints

```
# Create a new Python file and import the following packages:
import numpy as np from easyAI
import TwoPlayersGame, Human_Player, AI_Player, \ Negamax, SSS

# Define a class that contains all the methods needed to play the game

# Define the board with six rows and seven columns

# Define who's going to start the game. In this case, let's have player one starts
the game and define the position as:

# Define the positions
self.pos_dir = np.array([[[i, 0], [0, 1]]
        for i in range(6)] + [[[0, i], [1, 0]]
        for i in range(7)] + [[[i, 0], [1, 1]]
        for i in range(1, 3)] + [[[0, i], [1, 1]]
        for i in range(4)] + [[[i, 6], [1, -1]]
        for i in range(1, 3)] + [[[0, i], [1, -1]] for i in range(3, 7)])

# Define a method to get all the possible moves, define a method to control how to
make a move, and Define a method to show the current status.
```

Try: Building two bots to play Hex pawn against each other

12. Exercises on Game Playing

12.1 - Two-Persons, Perfect Information Games

Consider games with just two outcomes: win and Loss. Games where a draw is a possible outcome can be reduced to two outcomes: win, not win. The two players will be called 'us' and 'them'. 'Us' can win in a non-terminal 'us-to-move' position if there is a legal move that leads to a won position. On the other hand, a non-terminal 'them-to-move' position is won for 'us' if all the legal moves from this position lead to won positions. These rules correspond to AND/OR tree representation of problems. The goal of this exercise is to find whether an us-to-move position is won.

Input: Rules corresponding to AND/OR trees that can be adopted for searching game trees.

Output: Find whether an us-to-move position

Hints

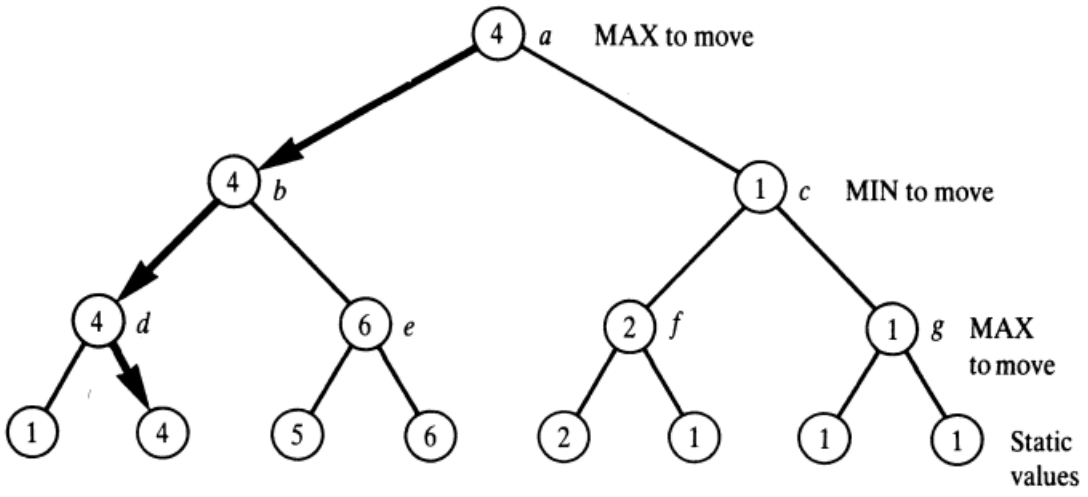
```
# Create a new Python file and import the following packages:
import numpy as np from easyAI
import TwoPlayersGame, Human_Player, AI_Player, \ Negamax, SSS

# Define a class that contains all the methods needed to play the game

# Define the board with six rows and seven columns

# Define who's going to start the game. In this case, let's have player one starts
the game and define the position as:

# Define the positions
self.pos_dir = np.array([[[i, 0], [0, 1]]
        for i in range(6)] + [[[0, i], [1, 0]]
        for i in range(7)] + [[[i, 0], [1, 1]]
        for i in range(1, 3)] + [[[0, i], [1, 1]]
        for i in range(4)] + [[[i, 6], [1, -1]]
        for i in range(1, 3)] + [[[0, i], [1, -1]] for i in range(3, 7)])

# Define a method to get all the possible moves, define a method to control how to
make a move, and Define a method to show the current status.
```

Try: Write a program to play some simple game (like nim) using the straightforward AND/OR search approach.

12.2 The minimax principle

The goal of this exercise is the straightforward implementation of the minimax principle.

Input:



Output: Compute the minimax backed-up value for a given position.

Hints

```
# minimax_simplenim.py

def minimax(state, max_turn):
    if state == 0:
        return 1 if max_turn else -1

    # ...
```

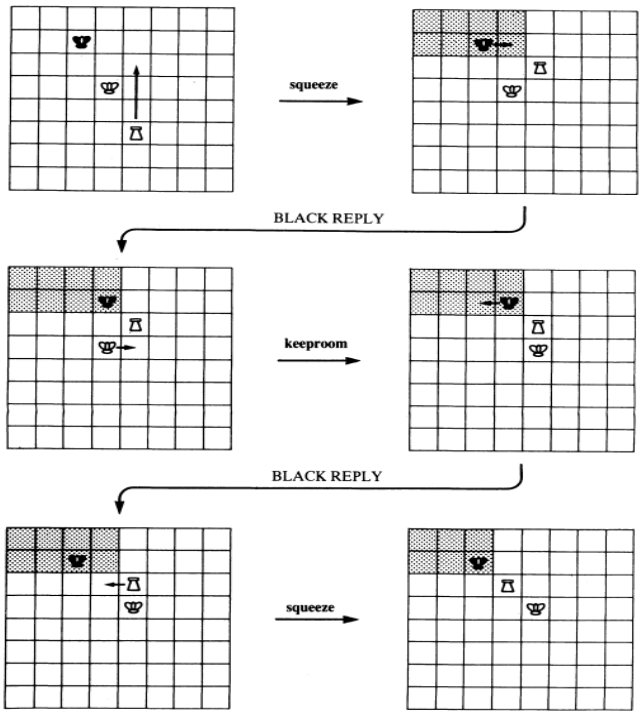
```
def minimax(state, max_turn):
    if state == 0:
        return 1 if max_turn else -1

    possible_new_states = [
        state - take for take in (1, 2, 3) if take <= state
    ]
    if max_turn:
        scores = [
            minimax(new_state, max_turn=False)
            for new_state in possible_new_states
        ]
        return max(scores)

    # ...
```

Try: You should play a few games of Nim to get a feel for how the new rules change the strategy. Try it with a different number of piles, say three, four, or five. You don't need a lot of counters in each pile. Between three and nine is a good starting point.

12.3 Advice Language 0

A broad strategy for winning with the king and rook against the sole opponent's king is to force the king to the edge, or into a corner if necessary, and deliver mate in a few moves. The goal is to develop a program that a play a game from a given starting position using knowledge representation in Advice Language 0.

Input: Goal predicates and Move constraint predicates.



Output: Predicate library for king and rock Vs king.

Hints

```
# minimax_simplenim.py

def minimax(state, max_turn):
    if state == 0:
        return 1 if max_turn else -1

    # ...

def minimax(state, max_turn):
    if state == 0:
        return 1 if max_turn else -1

    possible_new_states = [
        state - take for take in (1, 2, 3) if take <= state
    ]
    if max_turn:
        scores = [
            minimax(new_state, max_turn=False)
            for new_state in possible_new_states
        ]
        return max(scores)

    # ...
```

Try: Consider some other simple chess endgames, such as king and pawn vs. king, and write an ALO program (together with the corresponding predicate definitions) to play this endgame.

13. Exercises on Game Playing and Adversarial Search

13.1 Two Player Board Game as a Search Problem

Game playing assumes a multiple-agent environment, and thus offers ideal example for adversarial search. As the agents' goals are in conflict and they always plan against each other, the search space becomes complicated. Moreover, real games involve huge state spaces. The goal of this exercise is to find the optimal game strategies using MINIMAX algorithm.

Input:

1. Search problem with 4 components: Initial state, Successor function, Terminal test, Utility function.
A 3x3 grid is provided with the information of opener, and his/her symbol.

2. All nodes up to the terminals need to be generated, and assign them utilities (-1, 0, +1).

3. The MINIMAX values of non-terminals are computed up to the root, and the winning strategy is returned.

Output:

1. Allow the player MAX to search for the sequence of moves that leads to a terminal with maximum possible utility value, even if MIN plays in the best way.
2. Allow the MIN player to search for the opposite, that is, terminal with minimum possible utility.

Hints

```
/* DEFINING A DYNAMIC METHOD WILL BE USED IN THE PROGRAM */
:- dynamic settings/2.
/* SETTING USER DEFINED VALUES FOR SOME VARIABLES */
set(Key, Value) :-
        settings_value(Key, Value),
        !,
        retractall(settings(Key, _)),
        assert(settings(Key, Value)).
/* USER ENTERED SOME INVALID INPUTS */
set(Key, Value) :-
        write('Unknown value: '),
        write(set(Key, Value)),
        nl.
/* DEFAULT SETTINGS */
settings(statespaceDepth, 3).
settings(stones, 6).
settings(player1, minimax).
settings(player2, minimax).
settings(pauseDuration, 0.25).
```

```
/* BOUNDED VALUES FOR SOME SETTINGS */
settings_value(statespaceDepth, X) :-
        between(1, 10, X).
settings_value(stones, X) :-
        between(1, 10, X).
settings_value(player1, Value) :-
        playerSettings(Value).
settings_value(player2, Value) :-
        playerSettings(Value).
settings_value(pauseDuration, X) :-
        between(-1, 10, X).

/* DEFAULT ALGORITHMS THAT ARE IMPLEMENTED */
/* MANUAL MOVE BY USER */
playerSettings(manual).

/* PROGRAM MAKES USES MINMAX ALGORITHM */
playerSettings(minimax).

/* PROGRAM MAKES USE OF ALPHA-BETA PRUNING ALGORITHM */
playerSettings(alphabeta).

/*
--------------------------------------------------------------------------------
GAME PLAYING
--------------------------------------------------------------------------------
*/

/* GAME PLAYING FRAMEWORK */
play(Game) :-
        initialize(Game, Pos, Player),
        display(Pos, Player),
        displayPlayerAlgorithms,
        play(Pos, Player),
        true.

/* CHECK GAME END CONDITIONS */
play([Own, OwnKalaha, Opp, OppKalaha], Player) :-
        Own = [0, 0, 0, 0, 0, 0], /* OWN OR OPPONENTS SIDE HAS 0(ZERO) STONES IN ALL
OF THEIR HOLES */
        Opp = [0, 0, 0, 0, 0, 0],
        !,
        write('All stones moved to respective player kalaha..'),
        nl,
        write(Player),
        write(':'),
        write(OwnKalaha),
        write('points'),
        nl,
        nextPlayer(Player, Opponent),
        write(Opponent),
        write(':'),
        write(OppKalaha),
        write('points'),
        nl,
        nl,
```

```
        write('-x-x-x-x-x-x-x-x-x-x-x-x-x-x- GAME OVER -x-x-x-x-x-x-x-x-x-x-x-x-x-x-
x-x-x-x-'),
        nl,
        noprotocol.

/* PLAY THE GAME */
play([Own, OwnKalaha, Opp, OppKalaha], Player) :-
        chooseMove(Player, [Own, OwnKalaha, Opp, OppKalaha], [N|RN]),
        makeMove([NOwn, NOwnKalaha, NOpp, NOppKalaha], [Own, OwnKalaha, Opp,
OppKalaha], MoreMoves, N),
        display([NOpp, NOppKalaha, NOwn, NOwnKalaha], Player),
        makeMoveAgain([NOwn, NOwnKalaha, NOpp, NOppKalaha], Player, MoreMoves, RN).

/*
-------------------------------------------------------------------------------
CHOOSE A MOVE BASED ON ALGORITHMS
-------------------------------------------------------------------------------
*/

/* CHOOSE A MOVE. IMPLEMENT MINIMAX ALGO OR ALPHA-BETA PRUNING */
chooseMove(Player, Board, LN) :-
        write('Player = '),
        write(Player),
        nl,
        settings(Player, Alg),
        algorithm(Alg, Board, LN).

algorithm(manual, Board, LN) :-
        !,
        write('Make your move : '),
        performSearch(manual, Board, [gamemove(_, _, LN)|_]).

algorithm(Alg, Board, LN) :-
        !,
        write('just a moment please...'),
        nl,
        performSearch(Alg, Board, [gamemove(_, _, LN)|_]),
        write('...ok ready...'),
        nl,
        nl,
        LN=[N|_],
        write('My Move: '),
        write(N),
        write('.'),
        nl.

/*
-------------------------------------------------------------------------------
SEARCH FOR MOVES
-------------------------------------------------------------------------------
*/

/*
CHECK FOR LEGALALITY OF A MOVE...
JUST MAKING SURE THAT THE HOLE IS NOT EMPTY.
*/
checkLegalMove([X, _, _, _, _, _], 1) :- X > 0.
```

```
checkLegalMove([_, X, _, _, _, _], 2) :- X > 0.
checkLegalMove([_, _, X, _, _, _], 3) :- X > 0.
checkLegalMove([_, _, _, X, _, _], 4) :- X > 0.
checkLegalMove([_, _, _, _, X, _], 5) :- X > 0.
checkLegalMove([_, _, _, _, _, X], 6) :- X > 0.

/* FIND ALL MOVES WITH THE TERMINAL POSITION */
findAllMoves(Board, Bag) :-
        findall(gamemove(_, BP, BLM), goalMove(Board, BP, BLM), Bag),
        Bag \== [].

goalMove([A|RP], P, [N|RLM]) :-
        checkLegalMove(A, N),
        makeMove(NP, [A|RP], F, N),
        goalMoveTurn(NP, P, RLM, F).

goalMoveTurn(P, P, [], noMoreTurn).
goalMoveTurn(NP, P, LM, moreTurn) :-
        goalMove(NP, P, LM).

/* USER INPUT */
performSearch(manual, Board, [gamemove(unknown, _, [N])]) :-
        read(N),
        Board = [A|_],
        checkLegalMove(A, N),
        !.

performSearch(manual, Board, List) :-
        !,                                              /* RETRY USER INPUT */
        nl,
        write('Please try again : '),
        performSearch(manual, Board, List).

/* PERFORM SEARCH USING MINIMAX ALGORITHM */
performSearch(minimax, Board, BestList) :-
        !,                                              /*  THE  MIN  MAX  ALGORITHM
*/
        settings(statespaceDepth, D),
        settings(pauseDuration, PD),
        sleep(PD),
        findAllMoves(Board, Moves),
        settings(stones, S),
        MaxPts is S * 12,
        MinWin is S * 12 + 2,
        findMinimax(Moves, -1000000, D, MaxPts, MinWin),
        sort(Moves, BestListR),
        % reverze(BestListR, BestList).
        reverse(BestListR, BestList).

/* PERFORM SEARCH USING ALPHA-BETA PRUNING ALGORITHM */
performSearch(alphabeta, Board, BestList) :-
        !,                                          /* THE ALPHA BETA ALGORITHM */
        settings(statespaceDepth, D),
        settings(pauseDuration, PD),
        sleep(PD),
        findAllMoves(Board, Moves),
        settings(stones, S),
```

```
        MaxPts is S * 12,
        MinWin is S * 12 + 2,
        findAlphaBeta(Moves, -1000000, 1000000, D, MaxPts, MinWin),
        sort(Moves, BestListR),
        reverse(BestListR, BestList).

/*
-------------------------------------------------------------------------------
IMPLEMENTATIONS OF ALGORITHMS
-------------------------------------------------------------------------------
*/

/* MINIMAX ALGORITHM */
findMinimax([], _, _, _, _).

findMinimax([gamemove(NV, HP, _)|Moves], Pts, D, MP, MW) :-
        HP = [Own, OwnKalaha, Opp, OppKalaha],            /* ROTATE THE BOARD */
        miniMax([Opp, OppKalaha, Own, OwnKalaha], D, -1000000, V, MW),
        NV is -V,
        findMinimaxcases(Moves, Pts, D, NV, MP, MW).

findMinimaxcases(Moves, Pts, D, NV, MP, MW) :-
        Pts < NV,
        !,
        findMinimax(Moves,  NV, D, MP, MW).

findMinimaxcases(Moves, Pts, D,  _, MP, MW) :-
        !,
        findMinimax(Moves, Pts, D, MP, MW).

miniMax(P, D, Pts, V, MW) :-
        D > 0,
        P = [Own, OwnKalaha, _, OppKalaha],
        OwnKalaha < MW,
        OppKalaha < MW,
        /* NOT MORE THAN MinWin STONES */
        Own \== [0, 0, 0, 0, 0, 0],                       /* NOT REACHED THE END */
        !,
        findAllMoves(P, Moves),
        listMiniMax(Moves, D, Pts, V, MW).

miniMax(P, 0, _, V, _) :-
        !,
        P = [_, OwnKalaha, _, OppKalaha],
        V is OwnKalaha - OppKalaha.
        /* STATIC EVALUATION FUNCTION */

miniMax(P, _, _, V, _) :-
        !,
        P = [_, OwnKalaha, _, OppKalaha],         /* ...MORE THAN MinWin STONES */
        V is 1000 * (OwnKalaha - OppKalaha).      /* STATIC EVALUATION FUNCTION */

listMiniMax([], _, Pts, Pts, _).

listMiniMax([gamemove(_, HP, _)|Moves], D, Pts, V, MW) :-
        ND is D - 1,
        HP = [Own, OwnKalaha, Opp, OppKalaha],            /* ROTATE THE BOARD */
```

```
        miniMax([Opp, OppKalaha, Own, OwnKalaha], ND, -1000000, TV, MW),
        NV is - TV,
        listMiniMaxCases(Moves, D, Pts, V, NV, MW).

listMiniMaxCases(Moves, D, Pts, V, NV, MW) :-
        Pts < NV,
        !,
        listMiniMax(Moves, D, NV, V, MW).

listMiniMaxCases(Moves, D, Pts, V,  _, MW) :-
        !,
        listMiniMax(Moves, D, Pts, V, MW).

/* Alpha-Beta algorithm */
findAlphaBeta([], _, _, _, _, _).

findAlphaBeta([gamemove(NV, HP, _)|Moves], Alpha, Beta, D, MP, MW) :-
        NAlpha is - Beta,
        NBeta is - Alpha,
        HP = [Own, OwnKalaha, Opp, OppKalaha],                    /* ROTATE BOARD */
        alphaBeta([Opp, OppKalaha, Own, OwnKalaha], D, NAlpha, NBeta, V, MW),
        NV is -V,
        findAlphaBetaCases(Moves, Alpha, Beta, D, NV, MP, MW).

findAlphaBetaCases(Moves, Alpha, Beta, D, NV, MP, MW) :-
        Alpha < NV,
        !,
        findAlphaBeta(Moves, NV, Beta, D, MP, MW).

findAlphaBetaCases(Moves, Alpha, Beta, D,  _, MP, MW) :-
        !,
        findAlphaBeta(Moves, Alpha, Beta, D, MP, MW).

alphaBeta(P, D, Alpha, Beta, V, MW) :-
        D > 0,
        P = [Own, OwnKalaha, _, OppKalaha],
        OwnKalaha < MW,
        OppKalaha < MW,
        /* NOT MORE THAN MinWin STONES STONES IN SPACE */
        Own \== [0, 0, 0, 0, 0, 0],                        /* NOT REACHED END */
        !,
        findAllMoves(P, Moves),
        listAlphaBeta(Moves, D, Alpha, Beta, V, MW).

alphaBeta(P, 0, _, _, V, _) :-
        !,
        P = [_, OwnKalaha, _, OppKalaha],
        V is OwnKalaha - OppKalaha.
        /* STATIC EVALUATION FUNCTION */

alphaBeta(P, _, _, _, V, _) :-
        !,
        P = [_, OwnKalaha, _, OppKalaha],         /* ...MORE THAN MinWin STONES */
        V is 1000 * (OwnKalaha - OppKalaha).      /* STATIC EVALUATION FUNCTION */

listAlphaBeta([], _, Alpha, _, Alpha, _).
```

```
listAlphaBeta([gamemove(_, HP, _)|Moves], D, Alpha, Beta, V, MW) :-
        NAlpha is - Beta,
        NBeta is - Alpha,
        ND is D - 1,
        HP = [Own, OwnKalaha, Opp, OppKalaha],                    /* ROTATE BOARD */
        alphaBeta([Opp, OppKalaha, Own, OwnKalaha], ND, NAlpha, NBeta, TV, MW),
        NV is - TV,
        listAlphaBetaCases(Moves, D, Alpha, Beta, V, NV, MW).

listAlphaBetaCases(_, _, _, Beta, V, NV, _) :-
        Beta =< NV,
        !,
        V is NV + 1.

listAlphaBetaCases(Moves, D, Alpha, Beta, V, NV, MW) :-
        Alpha < NV,
        !,
        listAlphaBeta(Moves, D, NV, Beta, V, MW).

listAlphaBetaCases(Moves, D, Alpha, Beta, V,  _, MW) :-
        !,
        listAlphaBeta(Moves, D, Alpha, Beta, V, MW).

/*
-------------------------------------------------------------------------------
INITIALIZATION
-------------------------------------------------------------------------------
*/

initialize(kalaha, [[S, S, S, S, S, S], 0, [S, S, S, S, S, S], 0], player1) :-
        settings(stones, S).

/*
-------------------------------------------------------------------------------
DISPLAY BOARD ON THE SCREEN
-------------------------------------------------------------------------------
*/

/* SWAPPING THE OPPONENTS STONES WITH OWN STONES AND KALAHA */
swap([OwnStones, OwnKalaha, OppStones, OppKalaha], [OppStones, OppKalaha, OwnStones,
OwnKalaha]).

/* DISPLAYING THE CURRENT BOARD CONDITION */
display(Pos, player1) :-
        showgame(Pos).

/* FOR PLAYER2 WE NEED TO SWAP FIRST AND THEN DISPLAY THE BOARD. */
display(Pos, player2) :-
        swap(Pos, Pos1),
        showgame(Pos1).

/* PRINTS OUT THE CURRENT BOARD CONDITION ON THE SCREEN */
showgame([OwnStones, OwnKalaha, OppStones, OppKalaha]) :-
        reverse(OwnStones, OwnStonesRev),
        write('Player2  [6,5,4,3,2,1]'),
        nl,
        write('         '),
```

```
        write(OwnStonesRev),
        nl,
        write(OwnKalaha),
        write('                              '),
        write(OppKalaha),
        nl,
        write('         '),
        write(OppStones),
        nl,
        write('Player1  [1,2,3,4,5,6]'),
        nl,
        nl.

printAlgorithmName(alphabeta) :-
        write('Alpha-Beta Pruning Algorithm...').

printAlgorithmName(minimax) :-
        write('Minimax Algorithm...').

printAlgorithmName(manual) :-
        write('Manual : User makes his own moves...').

displayPlayerAlgorithms :-
        write('-------------------------------------------------'),
        nl,
        settings(player1, AlgPlayer1),
        write('Player1 algorithm : '),
        printAlgorithmName(AlgPlayer1),
        nl,
        settings(player2, AlgPlayer2),
        write('Player2 algorithm : '),
        printAlgorithmName(AlgPlayer2),
        nl,
        write('-------------------------------------------------'),
        nl,
        nl.


/*
-------------------------------------------------------------------------------
EXECUTING A MOVE
-------------------------------------------------------------------------------
*/

/* EXECUTING A MOVE */
makeMove([NOwn, NOwnKalaha, NOpp, NOppKalaha], [Own, OwnKalaha, Opp, OppKalaha],
MoreMoves, StartHole) :-
        pickStones(StartHole, Picked, TOwn, Own),
        seedStones(Board, [TOwn, OwnKalaha, Opp, OppKalaha], TMoreMoves, Picked,
StartHole),
        finalCheck([NOwn, NOwnKalaha, NOpp, NOppKalaha], Board, MoreMoves,
TMoreMoves).

/* PLAYER CAN PLAY ONE MORE TURN */
makeMoveAgain(P, Player, moreTurn, []) :-
        !,
        write('One more turn...'),
```

```
        play(P, Player).

/* CHECK FOR MANUAL MOVE TO BE MADE AGAIN */
makeMoveAgain(_, Player, moreTurn, _) :-
        settings(Player, manual),
        !,
        fail.

makeMoveAgain([Own, OwnKalaha, Opp, OppKalaha], Player, moreTurn, [H|T]) :-
        !,
        write('One more turn...'),
        nl,
        makeMove([NOwn, NOwnKalaha, NOpp, NOppKalaha], [Own, OwnKalaha, Opp,
OppKalaha], MoreMoves, H),
        write('My next move : '),
        write(H),
        write('.'),
        nl,
        display([NOpp, NOppKalaha, NOwn, NOwnKalaha], Player),
        makeMoveAgain([NOwn, NOwnKalaha, NOpp, NOppKalaha], Player, MoreMoves, T).

/* NO MORE TURNS FOR THE PLAYER */
makeMoveAgain([Own, OwnKalaha, Opp, OppKalaha], Player, noMoreTurn, []) :- !,
        nextPlayer(Player, Opponent),
        write('-------------------------------------------------'),
        nl,
        play([Opp, OppKalaha, Own, OwnKalaha], Opponent). /* Rotate the board for
opponent to play */

/*
GAME END CONDITION CHECK.
WHEN A PLAYER'S ALL HOLES ARE COMPLETELY EMPTY, THE GAME ENDS. THE PLAYER WHO
STILL HAS STONES LEFT IN HIS HOLES CAPTURES THOSE STONES AND PUTS THEM IN HIS
KALAHA. THE PLAYERS THEN COMPARE THEIR KALAHA. THE PLAYER WITH MOST STONES WINS
*/
finalCheck([NOwn, NOwnKalaha, NOpp, NOppKalaha], [[0, 0, 0, 0, 0, 0], OwnKalaha,
Opp, OppKalaha], noMoreTurn, _) :-
        !,
        Opp = [Opp1, Opp2, Opp3, Opp4, Opp5, Opp6],
        NOwn = [0, 0, 0, 0, 0, 0],
        NOpp = [0, 0, 0, 0, 0, 0],
        /*NOwnKalaha is OwnKalaha + Opp1 + Opp2 + Opp3 + Opp4 + Opp5 + Opp6.*/
        NOwnKalaha is OwnKalaha,
        NOppKalaha is OppKalaha + Opp1 + Opp2 + Opp3 + Opp4 + Opp5 + Opp6.

finalCheck([NOwn, NOwnKalaha, NOpp, NOppKalaha], [Own, OwnKalaha, [0, 0, 0, 0, 0,
0], OppKalaha], noMoreTurn, _) :-
        !,
        Own = [Own1, Own2, Own3, Own4, Own5, Own6],
        NOwn = [0, 0, 0, 0, 0, 0],
        NOpp = [0, 0, 0, 0, 0, 0],
        /*NOwnKalaha is OwnKalaha + Own1 + Own2 + Own3 + Own4 + Own5 + Own6.*/
        NOppKalaha is OppKalaha,
        NOwnKalaha is OwnKalaha + Own1 + Own2 + Own3 + Own4 + Own5 + Own6.

finalCheck(P, P, MoreMoves, MoreMoves).
```

```
/*
-------------------------------------------------------------------------------
GAME PLAYING RULES
-------------------------------------------------------------------------------
*/

/*
RULES AND GAME PLAYING MOVES
*/
/* PICKUP STONES AND START SEEDING */
pickStones(1, Picked, [0|L], [Picked|L]) :- !.
pickStones(N, Picked, [X|L2], [X|L1]) :-
        N > 1,
        M is N - 1,
        pickStones(M, Picked, L2, L1).

/*
BASIC RULE : STONES SHOULD BE SEEDED IN COUNTER-CLOCKWISE DIRECTION
*/
seedStones([NOwn, NOwnKalaha, NOpp, NOppKalaha], [Own, OwnKalaha, Opp, OppKalaha],
MoreMoves, Picked, StartHole) :-
        convertListToBoard(P, [Own, OwnKalaha, Opp, OppKalaha]),
        NextHole is StartHole + 1,
        simpleSeedStones(NP, P, MoreMoves, Picked, StartHole, NextHole),
        convertListToBoard(NP, [NOwn, NOwnKalaha, NOpp, NOppKalaha]).

/* CHECK ALL STONES SEEDED */

/*
SEED LAST IN OWN BOWL WITH ZERO STONES => DO A CAPTURE.
IF LAST SEED LANDS IN MY OWN HOLE WITH ZERO STONES,
THEN TAKE THIS STONE AND THE OPPOSITE HOLE'S STONES (OPPONENTS  SIDE)
AND PUT IT INTO MY KALAHA.
NOWNKALAHA IS NEW OWNKALAHA
*/
/*
IF LAST SEED LANDS IN MY OWN KALAHA THEN I GET ANOTHER TURN
*/

/*
NORMAL SEEDING OF THE LAST STONE
*/

/*
NORMAL SEEDING OF STONES (BUT NOT LAST)
*/

/*
IF LAST SEED ON OPPONENTS SIDE IN A BOWL WHERE THERE ARE 2 OR 3 STONES
THEN SHALL THOSE STONES BE MOVED TO MY KALAHA.  THIS SHALL BE REPEATED
CLOCKWISE UNTIL REACHING OWN KALAHA OR THERE THERE ARE NOT 2 OR 3
STONES IN THE BOWL.
*/

/*
```

```
SEARCH FOR START HOLE
*/
jump(1, P, P, NP, NP) :- !.

/*
-------------------------------------------------------------------------------
ALTERNATING BETWEEN PLAYER1 AND PLAYER2
-------------------------------------------------------------------------------
*/

nextPlayer(player1, player2).
nextPlayer(player2, player1).

/*
-------------------------------------------------------------------------------
GAME INSTRUCTIONS HEADER
-------------------------------------------------------------------------------
*/

:-
        protocol('gamelog.txt'),
        nl,
        write('----- Game Information -----'),
        nl,
        nl,
        write('This game is called Kalaha.'),
        nl,
        nl,
        write('----------------------------------------------------------------'),
        nl,
        write('The rules for this game are as follows:'),
        nl,
        nl,
        write('1.       A player can start his/her move from any non-empty pit from
his/her side of the board.'),
        nl,
        write('2.       The player cannot start his/her move using the pieces on the
opponents side of the board.'),
        nl,
        write('3.       The players Kalaha, the players pits and the opponents pits
are included in sowing. Opponents Kalaha is not included in sowing.'),
        nl,
        write('4.       Seeds  are  sowed  in  counter-clockwise  (anti-clockwise)
direction.'),
        nl,
        write('5.       The  player  cannot  sow  any  of  his/her  seeds  into  the
opponents Kalaha. i.e. we have to wrap around opponents Kalaha without placing any
stone there.'),
        nl,
        write('6.       If the last seed land in the players Kalaha, the players
score increases by 1 and he retain the right to continue playing.'),
        nl,
        write('7.       If the last seed does not end up in the players Kalaha, the
player loses his/her turn.'),
        nl,
```

```
        write('8.       If last seed on opponents side in a bowl where there are 2
or 3 stones then those stones be will moved to players Kalaha. This shall be repeated
clockwise until reaching own Kalaha or there are not 2 or 3 stones in the bowl.'),
        nl,
        write('9.       We cannot seed into the original hole from which the stones
were picked.'),
        nl,
        write('10.      If the last counter is put into an empty hole on the players
side of the board, a capture takes place: all stones in the opponents pit opposite
and the last stone of the sowing are put into the players kalaha and the opponent
moves next.'),
        nl,
        write('11.      If the last counter is put anywhere else, it is now the
opponents turn.'),
        nl,
        write('12.      The seeds that are being captured or the seeds that has
entered both players Kalaha do not re-enter the game. The game value of this game
thus depends on the configuration of the active seeds or seeds that are not
captured.'),
        nl,
        write('13.      When a players all holes are completely empty, the game
ends. The player who still has stones left in his holes captures those stones and
puts them in his Kalaha. The players then compare their Kalaha. The player with most
stones wins.!!'),
        nl,
        write('------------------------------------------------------------------------
-----------'),
        nl,
        nl,
        write('A log file of the current game is created and all moves will be
stored in this file at the end of the game...'),
        nl,
        write('The log file name is gamelog.txt'),
        nl,
        write('Check prolog current working directory using: working_directory(CWD,
CWD).'),
        nl,
        write('The log file is stored at working_directory(CWD, CWD) folder...'),
        nl,
        nl,
        write('To change number of stones - run set(stones, Integer).'),
        nl,
        write('To  change  state-space  search  depth  -  run  set(statespaceDepth,
Integer).'),
        nl,
        write('To change the pause duration between each move to have a look at the
moves made - run set(pauseDuration, Float).'),
        nl,
        write('To change player 1 alorithm - run set(player1, X).'),
        nl,
        write('To change player 2 alorithm - run set(player2, X).'),
        nl,
        write('  X can be either manual, minimax or alphabeta.'),
        nl, nl,
        write('The Default settings for the game are as follows: '),
        nl,
        write('stones = 6.'),
```

```
        nl,
        write('statespaceDepth = 3.'),
        nl,
        write('pauseDuration = 0.25'),
        nl,
        write('Player1 algorithm = manual.'),
        nl,
        write('Player2 algorithm = minimax.'),
        nl,
        nl,
        write('The game is started with play(kalaha).'),
        nl, nl.
```

Try:

1. Consider the same game playing and improve the performance of the MINIMAX search strategy by reducing the search space using alpha-beta pruning.
2. construct a pruned game tree using Alpha-Beta pruning. Take the sequence, [5, 3, 2, 4, 1, 3, 6, 2, 8, 7, 5, 1, 3, 4] of MINIMAX values for the nodes at the cutoff depth of 4 plies. Assume that branching factor is 2, MIN makes the first move, and nodes are generated from right to left.

13.2 Uncertainty and Probabilistic Reasoning

For decision making, rational agents are supposed to take help of probabilistic reasoning, beside utility theory for choosing from alternatives. Those tools are required for dealing with uncertainty due to partial knowledge of the environment, which is unavoidable. The goal of this exercise is to gain the insights of acting under uncertainty using probabilistic reasoning and implement simple environments for making probabilistic inference.

Input: Describe the domain using 3 Boolean random variables. Consider the below joint probabilities of the random variables, taken from a domain expert.

	A		¬A	
	C	¬C	C	¬C
B	0.108	0.012	0.072	0.008
¬B	0.016	0.064	0.144	0.576

Output: Compute the probability of any compound proposition from the given entries.

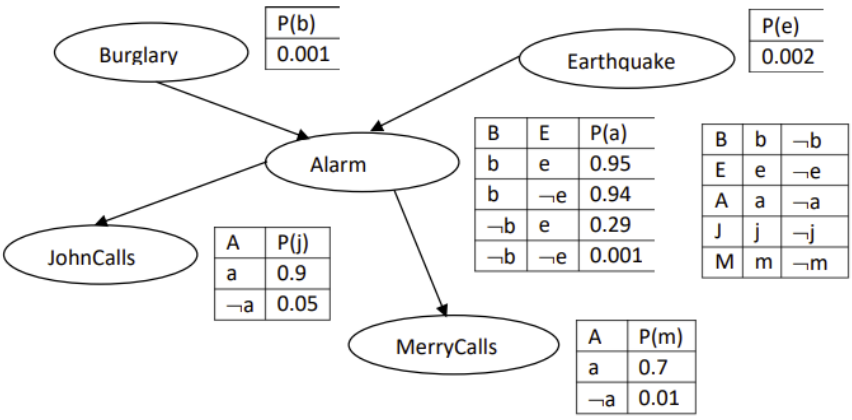
Hint

```
# We can define a predicate that implements the preceding independence-assumption
"and-combination" formula. It will take two arguments: an input list of
probabilities, and an output number for the combined probability.#

indep_andcombine([P],P).
indep_andcombine([P|PL],Ptotal) :- indep_andcombine(PL,P2), Ptotal is P2 * P.

# We just call this predicate as the last thing on the right side of rules, to
combine the "and"ed probabilities in a rule. If we had a rule without probabilities
like this:
```

```
f :- a, b, c.

# we would turn it into a rule with probabilities like this:

f(P) :- a(P1), b(P2), c(P3), indep_andcombine([P1,P2,P3],P).

#That addresses the third issue discussed in Section 8.1. Interestingly, indep_and
combine can also address the second issue discussed in Section 8.1, that of modeling
rule strengths. Suppose we have a rule:

g(P) :- d(P1), e(P2), indep_andcombine([P1,P2],P).

# The indep_andcombine handles uncertainty of d and e, but the rule itself may be
uncertain, meaning that the conclusion g has a probability less than 1 even when P1
and P2 are both 1. We could characterize this rule uncertainty itself with a
probability, the probability that the rule succeeds given complete certainty of all
terms "and"ed on its right side. If this probability were 0.7 for instance, we could
rewrite it:

g(P) :- d(P1), e(P2), indep_andcombine([P1,P2,0.7],P).

# In other words, rule uncertainty can be thought of as a "hidden" "and"ed predicate
expression) with an associated probability.
Here's an example. Suppose we have the following rule and facts:

f(P) :- a(P1), b, c(P2), indep_andcombine([P1,P2,0.8],P).
a(0.7).
b.
c(0.95).
Then for the query
?- f(X).

# P1 will be bound to 0.7, and P2 to 0.95. Predicate indep_andcombine computes |0.7 *
0.95 * 0.8 = 0.537|, and P is bound to that; so that's X, the total probability of
predicate f.
# For rules that refer only to things absolutely true and false, "and-combination" is
unnecessary. The rule-strength probability need only be on the left side, as for
instance:
f(0.7) :- a, b, c.
```

Try:

1. Consider the same example to provide complete and useful description of the domain with probabilistic reasoning using Bayesian networks.
2. Implement in Python or Prolog the environment for probabilistic inference using full joint-probability distribution as shown above.
3. Implement in Python or Prolog the environment for probabilistic inference using a Bayesian network as shown below.



14. Exercises on Expert Systems using Prolog/Python

14.1 Mental Health Disorder

Create an expert system in prolog to improve the understanding of declarative programming paradigm based on the logic rules. Develop the program with an idea of identifying the health disorder based on the database provide with mental health conditions.

Input: Set of logical rules with mental health conditions.

Output: Model the disorder of the patient health.

Hints

```
diagnose :-
 write('This is an expert system for dignosis of mental disorders.'), nl,
 write('There are several questions you need to answer for dignosis of mental
disorders.'), nl, nl,
 disorder(X),
 write('Condition was diagnosed as '),
 write(X),
 write('.').
diagnose :-
    write('The diagnose was not found.').


%The question predicate will have to determine from the user
%whether or not a given attribute-value pair is true
question(Attribute, Value):-
 retract(yes, Attribute, Value), !.
question(Attribute, Value):-
 retract(_, Attribute, Value),  !, fail.
question(Attribute, Value):-
 write('Is the '),
 write(Attribute),
 write(' - '),
 write(Value),
 write('?'),
```

```
 read(Y),
 asserta(retract(Y, Attribute, Value)),
 Y == yes.

%question with additional argument which contains
%a list of possible values for the attribute.
questionWithPossibilities(Attribute, Value, Possibilities) :-
 write('What is the patient`s '), write(Attribute), write('?'), nl,
 write(Possibilities), nl,
 read(X),
 check_val(X, Attribute, Value, Possibilities),
 asserta( retract(yes, Attribute, X) ),
 X == Value.

check_val(X, _, _, Possibilities) :-  member(X, Possibilities),
 !.
check_val(X, Attribute, Value, Possibilities) :-
 write(X), write(' is not a legal value, try again.'), nl,
 questionWithPossibilities(Attribute, Value, Possibilities).

%retract equips this system with a memory that remembers the facts that are already
%known because they were already entered by the user at some point during the
interaction.
:- dynamic(retract/3).

%. The program needs to be modified to specify which attributes are askable
food_amount(X) :- question(food_amount,X).
symptom(X) :- question(symptom,X).
mentality(X) :- question(mentality,X).
cause(X) :- question(cause, X).
indication(X) :- question(indication,X).
social_skill(X) :- question(social_skill,X).
condition(X) :- question(condition, X).
consequence(X) :- question(consquence,X).
specialty(X) :- question(specialty,X).
face_features(X) :- question(face_features,X).
ears_features(X) :- question(ears_features,X).
brain_function(X) :- question(brain_function,X).
perceptions(X) :- question(perceptions, X).
behavior(X)  :-  questionWithPossibilities(behavior,  X,  [repetitive_and_restricted,
narcissistic, aggresive]).


disorder(anorexia_nervosa) :- type(eating_disorder),
                              consequence(low_weight),
                              food_amount(food_restriction).

disorder(bulimia_nervosa) :- type(eating_disorder),
                             consequence(purging),
                     food_amount(binge_eating).

disorder(asperger_syndrome) :- type(neurodevelopmental_disorder),
                               specialty(psychiatry),
                               social_skill(low),
                               behavior(repetitive_and_restricted).

disorder(dyslexia) :- type(neurodevelopmental_disorder),
                      social_skill(normal),
```

```
                        perceptions(low),
                        symptom(trouble_reading).

disorder(autism) :- type(neurodevelopmental_disorder),
                    social_skill(low),
                    symptom(impaired_communication).


disorder(tourettes_syndrome) :- type(neurodevelopmental_disorder),
                                social_skill(normal),
                                specialty(neurology),
                                symptom(motor_tics).

disorder(bipolar_disorder) :- type(psychotic_disorder),
                              indication(elevated_moods).

disorder(schizophrenia) :- type(psychotic_disorder),
                           indication(hallucinations).

disorder(down_syndrome) :- type(genetic_disorder),
                           symptom(delayed_physical_growth),
                           face_features(long_and_narrow),
                           ears_features(large),
                           brain_function(intellectual_disability).

disorder(fragile_X_syndrome) :- type(genetic_disorder),
                                face_features(small_chin_and_slanted_eyes),
                                brain_function(intellectual_disability).


type(eating_disorder) :- symptom(abnormal_eating_habits),
                         mentality(strong_desire_to_be_thin).

type(neurodevelopmental_disorder) :-  condition(affected_nervous_system),
                                      brain_function(abnormal),
                                      cause(genetic_and_enviromental).

type(psychotic_disorder) :- symptom(false_beliefs),
                            mentality(manic_depressive),
                            cause(genetic_and_enviromental).

type(genetic_disorder) :- cause(abnormalities_in_genome).
```

Try: Create an Expert System suggesting the medical support for the diagnosis of kidney diseases. Include the features like:

1. Forward chaining reasoning
2. History and questions management
3. Backtrack and facts **revocation.**
4. Management of uncertainty through the CF approach proposed for the first time in the MyCIN expert **system.**
5. Explanation and translation form of the technical glossary

14.2 War Crimes Explorer

Create an expert system in prolog to improve the understanding of declarative programming paradigm based on the logic rules. Develop the program with an idea of:

- In-browser learning about genocide, war crimes, crimes against humanity, and aggression.
- Interactively enter facts and discover the laws that might have been broken.
- Possibly submit the information to the ICC (International Criminal Court) as a witness statement.

Input: Set of logical rules with laws and crimes within the jurisdiction of the International Criminal Court.

Output: Model the legal statutes.

Hints

```
/*
 * Crimes within the jurisdiction of the International Criminal Court.
 *
 *
https://world.public.law/rome_statute/article_5_crimes_within_the_jurisdiction_of_the
_court
 */
crime(genocide).
crime(war_crime).
crime(crime_against_humanity).
crime(crime_of_aggression).


/*
 * A first, simple attempt at Protected Persons under
 * the Geneva Conventions of 1949.
 */
protected_by_geneva_convention(P) :- civilian(P).
protected_by_geneva_convention(P) :- prisoner_of_war(P).
protected_by_geneva_convention(P) :- medical_personnel(P).
protected_by_geneva_convention(P) :- religious_personnel(P).


/*
 * D = Defendant
 * V = Victim
 */

/*
 * Genocide
 * https://world.public.law/rome_statute/article_6_genocide
 */
criminal_liability(genocide, Statute, D, V) :-
        elements(Statute, D, V).

/*
 * War crimes
 * https://world.public.law/rome_statute/article_8_war_crimes
```

```
 */
criminal_liability(war_crime, Statute, D, V) :-
        protected_by_geneva_convention(V),
        international_conflict(D, V),
        elements(Statute, D, V).


elements(article_8_2_a_i, D, V) :-
        act(D, killed, V).

elements(article_8_2_a_ii, D, V) :-
        act(D, tortured, V).
```

Try: Create an Expert System suggesting the medical support for the diagnosis of kidney diseases. Include the features like:

1. Forward chaining reasoning
2. History and questions management
3. Backtrack and facts **revocation.**
4. Management of uncertainty through the CF approach proposed for the first time in the MyCIN expert **system.**
5. Explanation and translation form of the technical glossary

14.3 DP Film Expert System

Create an expert system in prolog to improve the understanding of declarative programming paradigm based on the logic rules. Develop the program in such a way that it allows you to get film recommendations based on your answer and logic rules with a little film database.

Input: Set of logical rules with some film database.

Output: Recommend a file based on the persons name, mood, sex, time they have, and type of films interested.

Hints

```
# Sample database
/** DRAMA */
film('Zielona   mila','Frank   Darabont',1999,'drama',   'others',      188,'USA','Tom
Hanks',8.7,719).
film('Pif Paf! Jestes trup', 'Guy Ferland', 2002, 'drama', 'others',  87, 'Kanada',
'Ben Foster', 7.7, 16).
film('Dogville', 'Lars von Trier', 2003, 'drama', 'others', 178, 'Dania', 'Nicole
Kidman', 7.7, 45).
film('Z dystansu', 'Ton Kayne', 2011, 'drama', 'others', 100, 'USA', 'Adrien Brody',
7.9, 30).
film('Lista Schindlera', 'Steven Spielberg', 1993, 'drama', 'others', 195, 'USA',
'Adrien Brody', 8.4, 259).
film('Requiem dla snu', 'Darren Aronofsky', 2000, 'drama', 'others', 102, 'USA',
'Jared Leto', 7.9, 520).
film('Biutiful', 'Alejandro Gonzalez Inarritu', 2010, 'drama', 'others', 148,
'Hiszpania', 'Javier Bardem', 7.6, 12).
```

```
film('Czarny labedz', 'Darren Aronofsky', 2010, 'drama', 'others', 108, 'USA', 'Natalie Portman', 7.7, 248).
film('Gladiator', 'Ridley Scott', 2000, 'drama', 'others', 155, 'USA', 'Russell Crowe', 8.1, 552).
film('Dzien swira', 'Marek Koterski', 2002, 'drama', 'others', 123, 'Polska', 'Marek Kondrat', 7.8, 438).
film('Pianista', 'Roman Polanski', 2002, 'drama', 'others', 150, 'Polska', 'Adrien Brody', 8.3, 410).

/** COMEDY */
film('Seksmisja', 'Juliusz Machulski', 1984, 'comedy', 'others', 118, 'Polska', 'Jerzy Stuhr', 7.9, 420).
film('Forrest Gump', 'Robert Zemeckis', 1994, 'comedy', 'others', 144, 'USA', 'Tom Hanks', 8.6, 697).
film('Kac Vegas', 'Todd Phillips', 2009, 'comedy', 'others', 100, 'USA', 'Bradley Cooper', 7.3, 537).
film('Notykalni', 'Olivier Nakache', 2011, 'comedy', 'others', 112, 'Francja', 'Francois Cluzet', 8.7, 393).
film('Truman Show', 'Peter Weir', 1998, 'comedy', 'others', 103, 'USA', 'Jim Carrey', 7.4, 383).
film('Kiler', 'Juliusz Machulski', 1997, 'comedy', 'others', 104, 'Polska', 'Cezary Pazura', 7.7, 315).
film('Kevin sam w domu', 'Chris Columbus', 1990, 'comedy', 'others', 103, 'USA', 'Macaulay Culkin', 7.1, 297).
film('Mis', 'Stanislaw Bareja', 1980, 'comedy', 'others', 111, 'Polska', 'Stanislaw Tym', 7.8, 261).
film('Diabel ubiera sie u Prady', 'David Frankel', 2006, 'comedy', 'others', 109, 'USA', 'Meryl Streep', 6.9, 227).
film('Jak rozpetalem druga wojne swiatowa', 'Tadeusz Chmielewski', 1969, 'comedy', 'others', 236, 'Polska', 'Marian Kociniak', 7.9, 195).

# Write the predicates related to actors and create some helper functions

# Write the code to design an expert system asking some questions related to the mode, time available, gendre etc.
```

Try: Create an Expert System suggesting the medical support for the diagnosis of kidney diseases. Include the features like:

1. Forward chaining reasoning
2. History and questions management
3. Backtrack and facts **revocation.**
4. Management of uncertainty through the CF approach proposed for the first time in the MyCIN expert **system.**
5. Explanation and translation form of the technical glossary

15. Final Notes

The only way to learn programming is program, program, and program on challenging problems. The problems in this tutorial are certainly NOT challenging. There are tens of thousands of challenging problems available – used in training for various programming contests. Check out these sites:

1. Introduction to Artificial Intelligence with Python, Associated with Harvard University. CS50's Introduction to Artificial Intelligence with Python | Harvard University

2. NPTEL: An Introduction to Artificial Intelligence, https://nptel.ac.in/courses/106105077/

3. NPTEL: Artificial Intelligence Search Methods for Problem Solving, Artificial Intelligence Search Methods For Problem Solving - Course (nptel.ac.in).

4. IFACET (iitk.ac.in)

5. Introduction to Artificial Intelligence (AI) | Coursera in association with IBM.

6. http://www.ai.eecs.umich.edu

Student must have any one of the following certifications:

- Competitive Coding with AlphaCode Team - Competitive programming with AlphaCode (deepmind.com)
- IIIT Hyderabad Certification - Competitive programming with AlphaCode (deepmind.com)

V. TEXT BOOKS:

1. Elaine Rich, Kevin Knight, Shivashankar B Nair, "Artificial Intelligence", Tata McGraw Hill, 3rd edition, 2009.
2. Dan W. Patterson, "Introduction to Artificial Intelligence and Expert Systems", Prentice-Hall, 2007.

VI. REFERENCE BOOKS:

1. Nils J.Nilsson, "Principles of Artificial Intelligence", Narosa Publishing House, 1990.
2. Stuart Russell and Peter Norvig, "Artificial Intelligence A Modern Approach", Pearson Education, 2nd Edition, 2010.
3. VS Janakiraman K, Sarukesi Gopalakrishnan, "Foundations of Artificial Intelligence & Expert Systems", Macmillan.

VII. WEB REFERENCES:

1. Department of Computer Science, University of California, Berkeley, http://www.youtube.com/playlist?list=PLD52D2B739E4D1C5F
2. NPTEL: Artificial Intelligence, https://nptel.ac.in/courses/106105077/
3. http://www.udacity.com/
4. http://www.library.thinkquest.org/2705/
5. http://www.ai.eecs.umich.edu/