Experiment-1

Installation of gnu-prolog, Study of Prolog (gnu-prolog), its facts, and rules.

Aim:

Installation of gnu-prolog, Study of Prolog (gnu-prolog), its facts, and rules.

Prolog Version

In this tutorial, we are using GNU Prolog, Version: 1.4.5

Official Website

This is the official GNU Prolog website where we can see all the necessary details about GNU Prolog, and also get the download link.

http://www.gprolog.org/

Direct Download Link

Given below are the direct download links of GNU Prolog for Windows. For other operating systems like Mac or Linux, you can get the download links by visiting the official website (Link is given above) −

http://www.gprolog.org/setup-gprolog-1.4.5-mingw-x86.exe (32 Bit System)

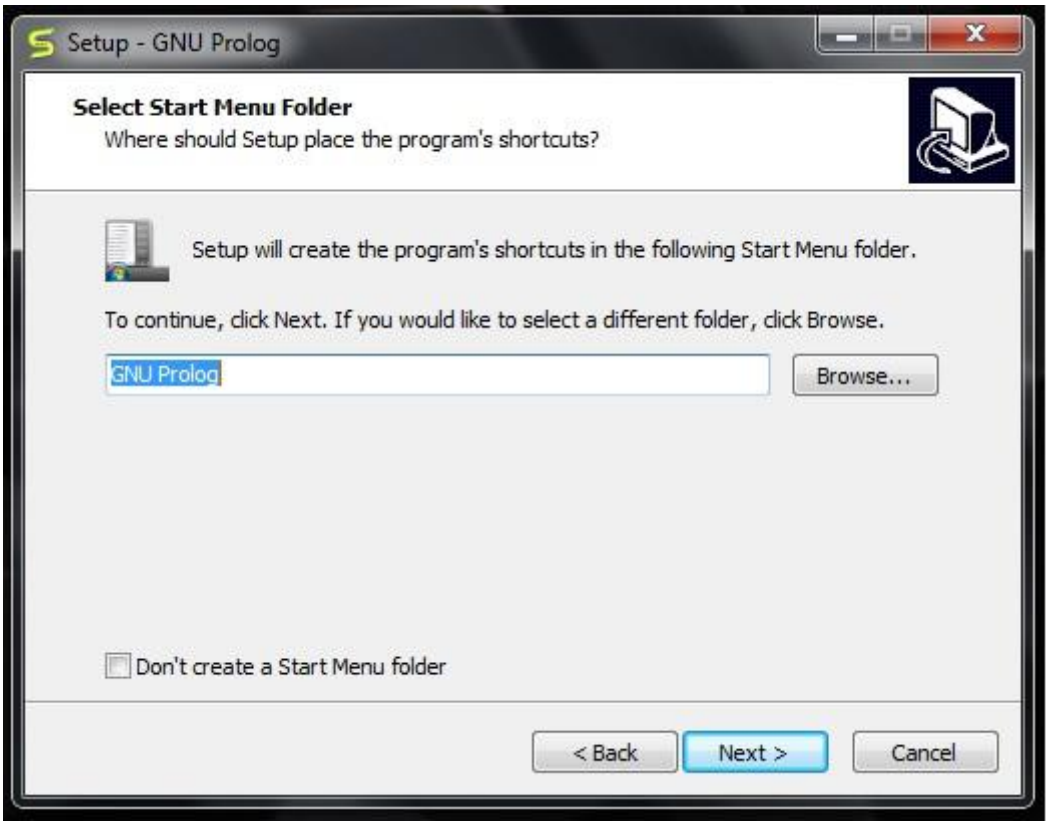
http://www.gprolog.org/setup-gprolog-1.4.5-mingw-x64.exe(64 Bit System)

Installation Guide

- Download the exe file and run it.
- You will see the window as shown below, then click on **next** −

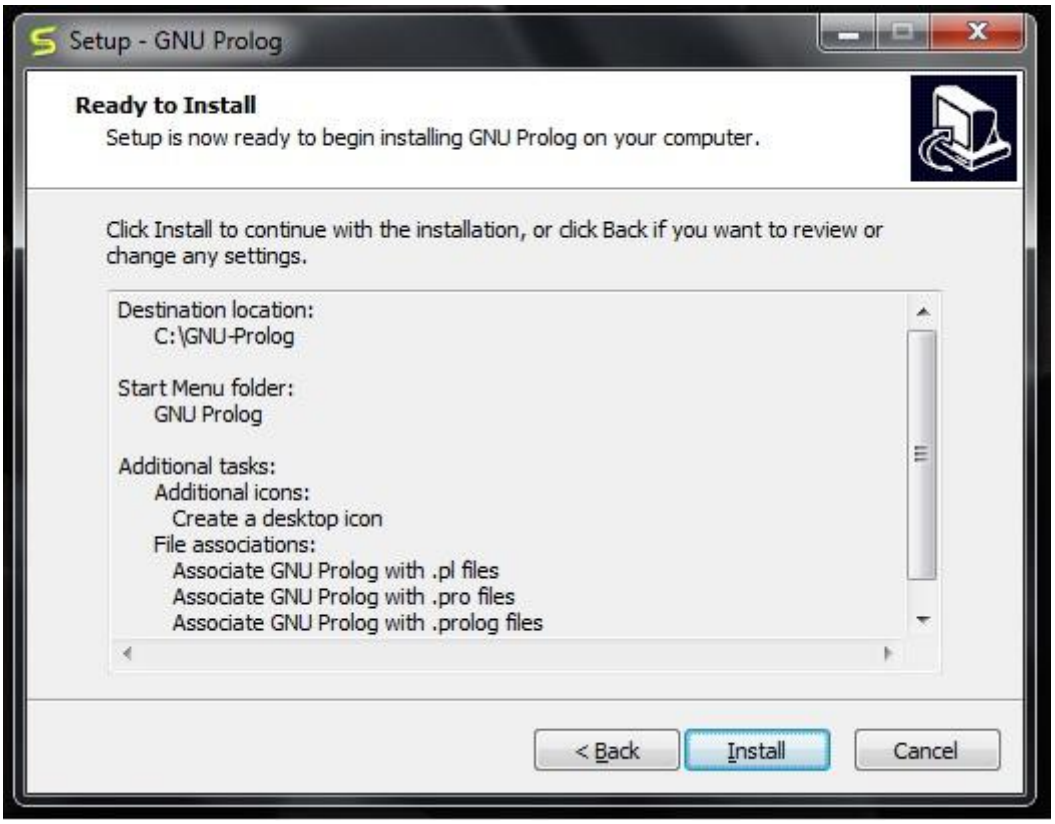
Select proper **directory** where you want to install the software, otherwise let it be installed on the default directory. Then click on **next**.

You will get the below screen, simply go to **next**.

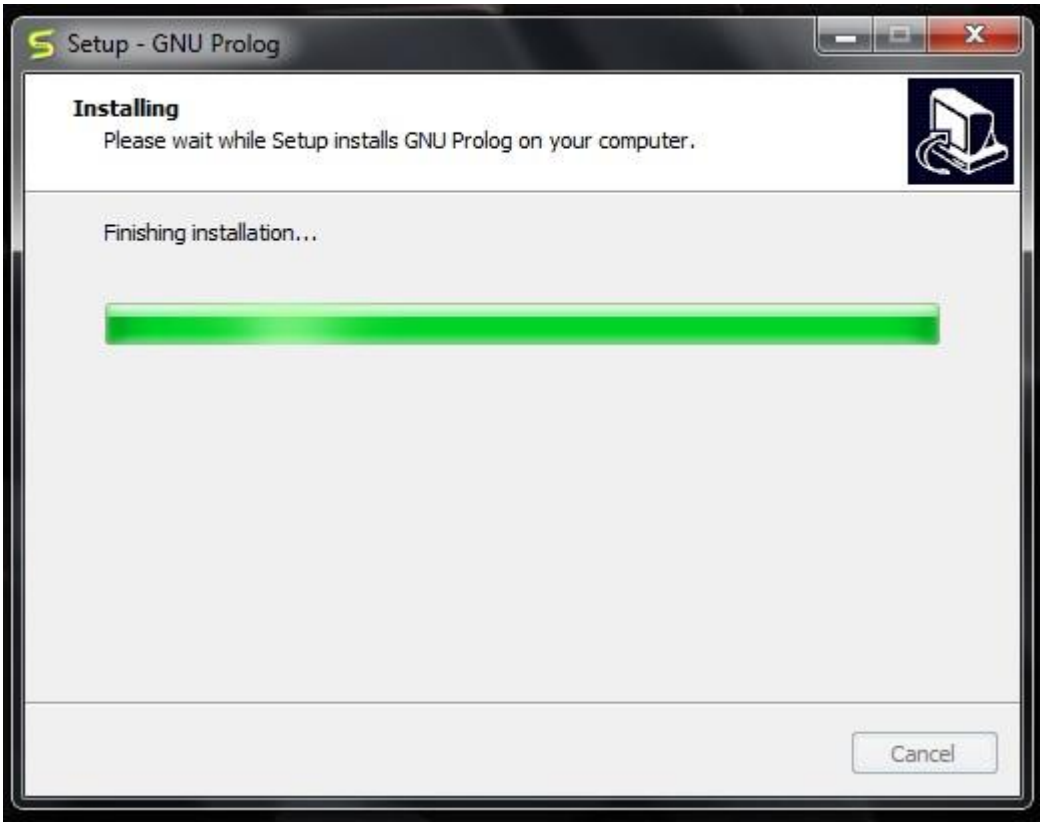


You can verify the below screen, and **check/uncheck** appropriate boxes, otherwise you can leave it as default. Then click on **next**.

In the next step, you will see the below screen, then click on **Install**.

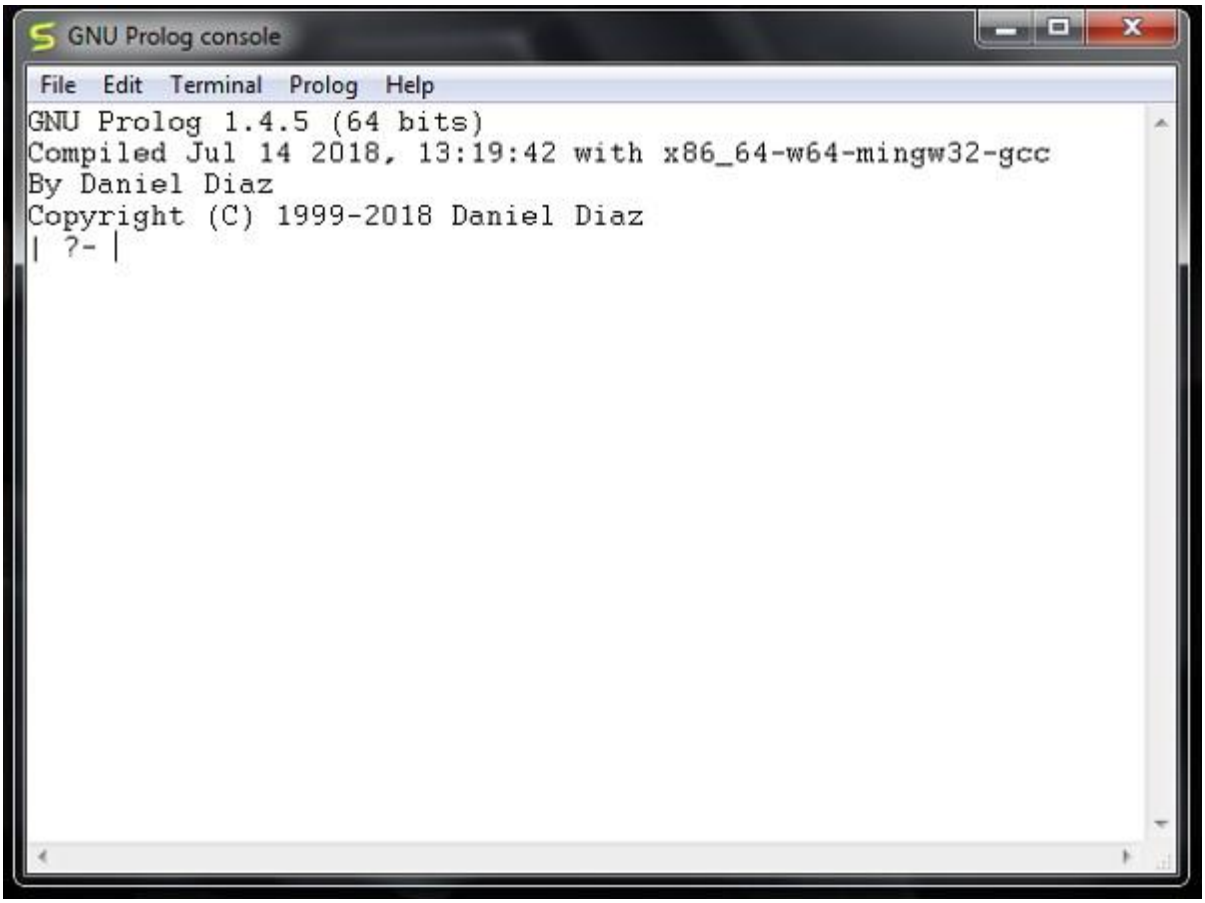


Then wait for the installation process to finish.



Finally click on **Finish** to start GNU Prolog.

The GNU prolog is installed successfully as shown below −


GNU Prolog console
File Edit Terminal Prolog Help
GNU Prolog 1.4.5 (64 bits)
Compiled Jul 14 2018, 13:19:42 with x86_64-w64-mingw32-gcc
By Daniel Diaz
Copyright (C) 1999-2018 Daniel Diaz
| ?-

Experiment-2

Write simple facts for the statements and querying it.

Aim:

Write simple facts for the statements and querying it.

Study of PROLOG. Write the following programs using PROLOG Using Turbo Prolog
Topics:
a)Basics of Turbo Prolog
b)Intro to Prolog programming
c) Running a simple program

- Prolog is a logical programming language and stands for PROgramming in LOGic Created around 1972.
- Preferred for AI programming and mainly used in such areas as:
  Theorem proving, expert systems, NLP, …

- Logical programming is the use of mathematical logic for computer programming.To start Turbo Prolog, open a MSDOS window and type:
  N> prolog followed by a carriage return.
- **The GUI:**
  - GUI is composed of four panels and a main menu bar.
  - The menu bar lists six options – Files, Edit, Run, Compile, Options, Setup.
  - The four panels are Editor, Dialog, Message and Trace.
- **MENU**
  - **Files** – Enables the user to load programs from disk, create new programs, save modified programs to disk, and to quit the program.
  - **Edit** – Moves user control to the Editor panel.
  - **Run** – Moves user control to the Dialog panel ; compiles the user program (if not already done so) in memory before running the program.
  - **Compile** – Provides the user with choices on how to save the compiled version of the program.
  - **Options** – Provides the user with choices on the type of compilation to be used.
  - Setup – Enables the user to change panel sizes, colors, and positions.

- **Dialog**
  - When a Prolog program is executing, output will be shown in the Dialog Panel
- **Message**
  - The Message Panel keeps the programmer up to date on processing activity.
- **Trace**
  - The Trace Panel is useful for finding problems in the programs you create.

Prolog Clauses

Any factual expression in Prolog is called a clause.
There are two types of factual expressions: facts and rules
There are three categories of statements in Prolog:

- **Facts:** Those are true statements that form the basis for the knowledge base.
- **Rules:** Similar to functions in procedural programming (C++, Java…) and has the form of if/then.
- **Queries:** Questions that are passed to the interpreter to access the knowledge base and start the program.

What is a Prolog program?

- Prolog is used for solving problems that involve objects and the relationships between objects.
- A program consists of a database containing one or more facts and zero or more rules(next week).
- A fact is a relationship among a collection of objects. A fact is a one-line statement that ends with a full-stop.
  parent (john, bart). parent (barbara, bart). male (john).
  dog(fido). >> Fido is a dog or It is true that fido is a dog sister(mary, joe). >> Mary is Joe's sister.
  play(mary, joe, tennis). >> It is true that Mary and Joe play tennis.
- Relationships can have any number of objects.
- Choose names that are meaningful – because in Prolog names are arbitrary strings but people will have to associate meaning to them.

Facts… Syntax rules:

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

EXAIPLE:

a) FACTS:

Some facts about family relationships could be written as:
sister(sue,bill)
parent(ann.sam)
male(jo)
female(riya)

b) RULES:

To represent the general rule for grandfather, we write:
grand f.gher(X2)
parent(X,Y)
parent(Y,Z)
male(X)

c) QUERIES:

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

?-parent(X,sam) Xann

?grandfather(X,Y)

X=jo, Y=sam

Terminology:

1. The names of the objects that are enclosed within the round brackets in each fact are called arguments.
2. The name of the relationship, which comes just before the round brackets, is called the predicate.
3. The arguments of a predicate can either be names of objects (constants) or variables.
4. When defining relationships between objects using facts, attention should be paid to the order in which the objects are listed. While programming the order is arbitrary, however the programmer must decide on some order and be consistent.
5. Ex. likes(tom, anna). >> The relationship defined has a different meaning if the order of the objects is changed. Here the first object is understood to be the "liker". If we wanted to state that Anna also likes Tom then we would have to add to our database – likes(anna, tom).
6. Remember that we must determine how to interpret the names of objects and relationships.

Constants & Variables Constants

- Constants are names that begin with lower case letters.
- Names of relationships are constants

Variables

- Variables take the place of constants in facts.
- Variables begin with upper case letters.

PROLOG IN DISGINING EXPERT SYSTEMS:

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

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

Write simple fact for following:

a. Ram likes mango.
b. Seema is a girl.
c. Bill likes Cindy.
d. Rose is red.
e. John owns gold.

Program:

```
Clauses likes(ram ,mango).
girl(seema).
red(rose).
likes(bill ,cindy).
owns(john ,gold).
```

Output:

```
Goal
queries
?-likes(ram,What).
What= mango
?-likes(Who,cindy).
Who= cindy
?-red(What).
What= rose
?-owns(Who,What).
Who= john
What= gold.
```

Experiment-6

Write a program which behaves a small expert for medical Diagnosis.

Aim: to Write a program which behaves a small expert for medical Diagnosis Using Prolog

```
Domains:
disease,indication=symbol
name-string

Predicates:
hypothesis(name,disease)
symptom(name,indication)
response(char)
go
goonce
clauses
go:-
goonce
write("will you like to try again (y/n)?"),
response(Reply),
Reply='n'.
go.
goonce:-
write("what is the patient's name"),nl,
readln(Patient),
hypothesis(Patient,Disease),!,
write(Patient,"probably has",Disease),!,
goonce:-
write("sorry, i am not ina position to diagnose"),
write("the disease").
symptom(Patient,fever):-
write("does",Patient,"has a fever (y/n)?"),nl,
response(Reply),
Reply='y',nl.
symptom(Patient,rash):-
write ("does", Patient,"has a rash (y/n)?" ),nl,
response(Reply),
Reply='y',
symptom(Patient_body,ache):-
write("does",Patient,"has a body ache (y/n)?"),nl,
response(Reply).
Reply='y',nl.
symptom(Patient,runny_nose):-
write("does",Patient,"has a runny_nose (y/n)?"),
response(Reply),
Reply='y'
hypothesis(Patient,flu):-
symptom(Patient,fever),
```

```
symptom(Patient,body_ache),
hypothesis(Patient,common_cold):-
symptom(Patient,body_ache),
Symptom(Patient,runny_nose).
response(Reply):-
readchar(Reply),
write(Reply).
```

Output:

```
makewindow(1,7,7"Expert Medical Diagnosis",2,2,23,70)
go
```

Experiment-7

Write programs for computation of recursive functions like factorial Fibonacci numbers, etc.

Aim: to write programs for computation of recursive functions like factorial Fibonacci numbers, etc.

A) Factorial of a Number Using Recursion

```
# Factorial of a number using recursion

def recur_factorial(n):
   if n == 1:
      return n
   else:
      return n*recur_factorial(n-1)

num = 6

# check if the number is negative
if num < 0:
   print("factorial does not exist for negative numbers")
elif num == 0:
   print("The factorial of 0 is 1")
else:
   print("The factorial of", num, "is", recur_factorial(num))
```

Output

Factorial of 6 is 720

B) Fibonnaci of a Number Using Recursion

```
# Python program to display the Fibonacci sequence
def recur_fibo(n):
   if n <= 1:
      return n
   else:
      return(recur_fibo(n-1) + recur_fibo(n-2))

nterms = 10

# check if the number of terms is valid
if nterms <= 0:
   print("Plese enter a positive integer")
else:
   print("Fibonacci sequence:")
   for i in range(nterms):
      print(recur_fibo(i), end=" ")
```

Out Put:

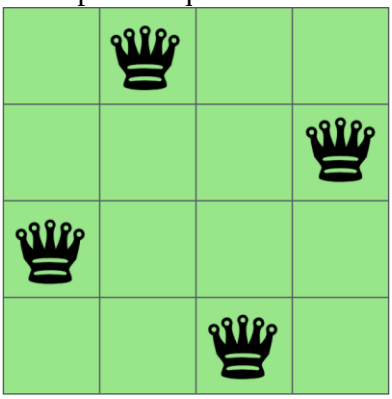
Fibonacci sequence:
0 1 1 2 3 5 8 13 21 34

Experiment-8

Write a program to solve 8-queens problem.

Aim: To write a program to solve 8-queens problem using Python.

The eight queens puzzle, or the eight queens problem, asks how to place eight queens on a chessboard without attacking each other. If you never played chess before, a queen can move in any direction (horizontally, vertically and diagonally) any number of places. In the next figure, you can see two queens with their attack patterns:
In the figure below you can see how to place 4 queens on a 4×4 board.



For Source code check in canvas

Experiment-10

Write a program for travelling salesman program.

Traveler needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once. What is the shortest possible route that he visits each city exactly once and returns to the origin city?

Travelling salesman problem is the most notorious computational problem. We can use brute-force approach to evaluate every possible tour and select the best one. For **n** number of vertices in a graph, there are **(*n* - 1)!** number of possibilities.

Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.

Let us consider a graph ***G = (V, E)***, where ***V*** is a set of cities and ***E*** is a set of weighted edges. An edge ***e(u, v)*** represents that vertices ***u*** and ***v*** are connected. Distance between vertex ***u*** and ***v*** is ***d(u, v)***, which should be non-negative.

Suppose we have started at city ***1*** and after visiting some cities now we are in city ***j***. Hence, this is a partial tour. We certainly need to know ***j***, since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem.

For a subset of cities ***S Є {1, 2, 3, ... , n}*** that includes ***1***, and ***j Є S***, let ***C(S, j)*** be the length of the shortest path visiting each node in **S** exactly once, starting at ***1*** and ending at ***j***.
When $|S| > 1$, we define ***C(S, 1)*** $= \propto$ since the path cannot start and end at **1**.

Now, let express **C(S, j)** in terms of smaller sub-problems. We need to start at ***1*** and end at **j**. We should select the next city in such a way that

$C(S,j)=\min C(S-\{j\},i)+d(i,j)$ where $i \in S$ and $i \neq j$

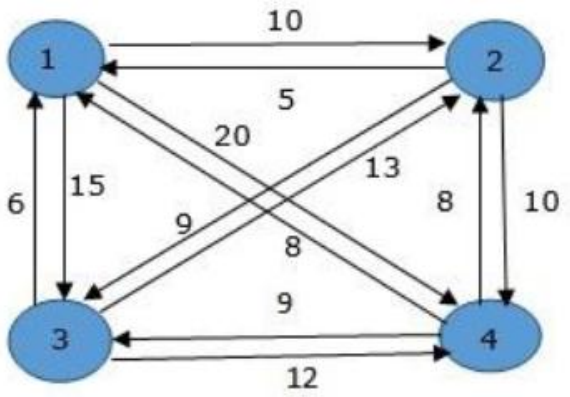
$C(S,j)=\min C(s-\{j\},i)+d(i,j)$ where $i \in S$ and $i \neq j$

$C(S,j)=\min C(S-\{j\},i)+d(i,j)$ where $i \in S$ and $i \neq j$

$C(S,j)=\min C(s-\{j\},i)+d(i,j)$ where $i \in S$ and $i \neq j$

Example

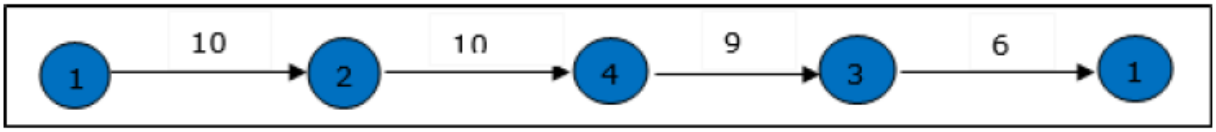
In the following example, we will illustrate the steps to solve the travelling salesman problem.



From the above graph, the following table is prepared.

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

The minimum cost path is 35.



For Source code check in canvas

Experiment-11

Case study of standard AI programs like Mycin and AI Shell.

Aim: To Case study of standard AI programs like Mycin and AI Shell.

MYCIN: A Case Study

No course on Expert systems is complete without a discussion of Mycin. As mentioned above, Mycin was one of the earliest expert systems, and its design has strongly influenced the design of commercial expert systems and expert system shells.

Mycin was an expert system developed at Stanford in the 1970s. Its job was to diagnose and recommend treatment for certain blood infections. To do the diagnosis ``properly'' involves growing cultures of the infecting organism. Unfortunately this takes around 48 hours, and if doctors waited until this was complete their patient might be dead! So, doctors have to come up with quick guesses about likely problems from the available data, and use these guesses to provide a ``covering'' treatment where drugs are given which should deal with any possible problem.

Mycin was developed partly in order to explore how human experts make these rough (but important) guesses based on partial information. However, the problem is also a potentially important one in practical terms - there are lots of junior or non-specialised doctors who sometimes have to make such a rough diagnosis, and if there is an expert tool available to help them then this might allow more effective treatment to be given. In fact, Mycin was never actually used in practice. This wasn't because of any weakness in its performance - in tests it outperformed members of the Stanford medical school. It was as much because of ethical and legal issues related to the use of computers in medicine - if it gives the wrong diagnosis, who do you sue?

Anyway Mycin represented its knowledge as a set of IF-THEN rules with certainty factors. The following is an English version of one of Mycin's rules:

1. IF the infection is pimary-bacteremia
2. AND the site of the culture is one of the sterile sites
3. AND the suspected portal of entry is the gastrointestinal tract
4. THEN there is suggestive evidence (0.7) that infection is bacteroid.

The 0.7 is roughly the certainty that the conclusion will be true given the evidence. If the evidence is uncertain the certainties of the bits of evidence will be combined with the certainty of the rule to give the certainty of the conclusion.

Mycin was written in Lisp, and its rules are formally represented as Lisp expressions. The action part of the rule could just be a conclusion about the problem being solved, or it could be an

arbitary lisp expression. This allowed great flexibility, but removed some of the modularity and clarity of rule-based systems, so using the facility had to be used with care.

Anyway, Mycin is a (primarily) goal-directed system, using the basic backward chaining reasoning strategy that we described above. However, Mycin used various heuristics to control the search for a solution (or proof of some hypothesis). These were needed both to make the reasoning efficient and to prevent the user being asked too many unnecessary questions.

One strategy is to first ask the user a number of more or less preset questions that are always required and which allow the system to rule out totally unlikely diagnoses. Once these questions have been asked the system can then focus on particular, more specific possible blood disorders, and go into full backward chaining mode to try and prove each one. This rules out alot of unecessary search, and also follows the pattern of human patient-doctor interviews.

The other strategies relate to the way in which rules are invoked. The first one is simple: given a possible rule to use, Mycin first checks all the premises of the rule to see if any are known to be false. If so there's not much point using the rule. The other strategies relate more to the certainty factors. Mycin will first look at rules that have more certain conclusions, and will abandon a search once the certainties involved get below 0.2.

A dialogue with Mycin is somewhat like the mini dialogue we gave in section 5.3, but of course longer and somewhat more complex. There are three main stages to the dialogue. In the first stage, initial data about the case is gathered so the system can come up with a very broad diagnosis. In the second more directed questions are asked to test specific hypotheses. At the end of this section a diagnosis is proposed. In the third section questions are asked to determine an appropriate treatment, given the diagnosis and facts about the patient. This obviously concludes with a treatment recommendation. At any stage the user can ask why a question was asked or how a conclusion was reached, and when treatment is recommended the user can ask for alternative treatments if the first is not viewed as satisfactory.

Mycin, though pioneering much expert system research, also had a number of problems which were remedied in later, more sophisticated architectures. One of these was that the rules often mixed domain knowledge, problem solving knowledge and ``screening conditions" (conditions to avoid asking the user silly or awkward questions - e.g., checking patient is not child before asking about alcoholism). A later version called NEOMYCIN attemped to deal with these by having an explicit disease taxonomy (represented as a frame system) to represent facts about different kinds of diseases. The basic problem solving strategy was to go down the disease tree, from general classes of diseases to very specific ones, gathering information to differentiate between two disease subclasses (ie, if disease1 has subtypes disease2 and disease3, and you know that the patient has the disease1, and subtype disease2 has symptom1 but not disease3, then ask about symptom1.)

There were many other developments from the MYCIN project. For example, EMYCIN was really the first expert shell developed from Mycin. A new expert system called PUFF was developed using EMYCIN in the new domain of heart disorders. And systom called NEOMYCIN was developed for training doctors, which would take them through various example cases, checking their conclusions and explaining where they went wrong.

We should make it clear at this point that not all expert systems are Mycin-like. Many use different approaches to both problem solving and knowledge representation. A full course on expert systems would consider the different approaches used, and when each is appropriate.