Lab Manual

CSC462 - Artificial Intelligence

CUI

Department of Computer Science
Islamabad Campus

Lab Contents:

Topics include: Introduction to python; How to Implement Algorithms; Intelligent Agents; Depth First Search; Breadth First Search; Iterative Deepening Search; Uniform Cost Search; A* Algorithm; Hill Climbing; Constraint Satisfaction Problems, First Order Logic; Inference in First Order Logic; and Expert Systems.

Student Outcomes (SO)

S.#	Description
2	Identify, formulate, research literature, and solve *complex* computing problems reaching substantiated conclusions using fundamental principles of mathematics, computing sciences, and relevant domain disciplines.
4	Create, select, adapt, and apply appropriate techniques, resources, and modern computing tools to *complex* computing activities, with an understanding of the limitations.

Intended Learning Outcomes

Sr.#	Description	Blooms Taxonomy Learning Level	SO
CLO-6	Implement various searching technique, CSP and knowledge-based system to solve a problem.	*Applying*	2,4

Lab Assessment Policy

The lab work done by the student is evaluated using Psycho-motor rubrics defined by the course instructor, viva-voce, project work/performance. Marks distribution is as follows:

Assignments	Lab Mid Term Exam	Lab Terminal Exam	Total
25	25	50	100

Note: Midterm and Final term exams must be computer based.

Table of Contents

Lab #	Main Topic	Page #
Lab 01	Python revision	**1**
Lab 02	Python Collections	**14**
Lab 03	Breadth first search	**38**
Lab 04	Depth first search	**49**
Lab 05	Iterative deepening search	**57**
Lab 06	Uniform Cost search	**62**
Lab 07	A* algorithm	**70**
Lab 08	Hill Climbing	**78**
Lab 09	**Mid Term Exam**	
Lab 10	Genetic Algorithn	**84**
Lab 11	Constrain Satisfaction Problem	**91**
Lab 12	Introduction to Prolog	**97**
Lab 13	Complex knowledge base	**104**
Lab 14	Expert Systems	**114**
Lab 15	**Final Term Exam**	

Lab 01
Python Introduction

Objective:

This lab is an introductory session on Python. The lab will equip students with necessary concepts needed to build algorithms in Python.

Activity Outcomes:

The lab will teach students to:

- Basic Operations on strings and numbers
- Basic use of conditionals
- Basic use of loops

Instructor Note:

As a pre-lab activity, read Chapters 3-5 from the book (Introduction to Programming Using Python - Y. Liang (Pearson, 2013)) to gain an insight about python programming and its fundamentals.

1) Useful Concepts

Python refers to the Python programming language (with syntax rules for writing what is considered valid Python code) and the Python interpreter software that reads source code (written in the Python language) and performs its instructions.
Python is a general-purpose programming language. That means you can use Python to write code for any programming task. Python is now used in the Google search engine, in mission-critical projects at NASA, and in transaction processing at the New York Stock Exchange.
Python's most obvious feature is that it uses indentation as a control structure. Indentation is used in Python to delimit blocks. **The number of spaces** is variable, but all statements within the same block must be indented the same amount. The header line for compound statements, such as if, while, def, and class should be terminated with a colon (**:**)

Example: Indentation as a control-structure

```
for i in range(20):
        if i%3 == 0:
                print i
        if i%5 == 0:
                print "Bingo!"
        print "---"
```

Variables
Python is dynamically typed. You do not need to declare variables! The declaration happens automatically when you assign a value to a variable. Variables can change type, simply by assigning them a new value of a different type. Python allows you to assign a single value to several variables simultaneously. You can also assign multiple objects to multiple variables.

Data types in Python
Every value in Python has a datatype. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.
There are various data types in Python. Some of the important types are listed below.

Integers
In Python 3, there is effectively no limit to how long an integer value can be. Of course, it is constrained by the amount of memory your system has, as are all things, but beyond that an integer can be as long as you need it to be:

```
>>> print(123123123123123123123123123123123123123123123123 + 1)
123123123123123123123123123123123123123123123124
```

Python interprets a sequence of decimal digits without any prefix to be a decimal number:

```
>>> print(10)
10
```

The following strings can be prepended to an integer value to indicate a base other than 10:

Prefix	Interpretation	Base
0b (zero + lowercase letter 'b') 0B (zero + uppercase letter 'B')	Binary	2
0o (zero + lowercase letter 'o') 0O (zero + uppercase letter 'O')	Octal	8
0x (zero + lowercase letter 'x') 0X (zero + uppercase letter 'X')	Hexadecimal	16

For example:

```
>>> print(0o10)
8
>>> print(0x10)
16
>>> print(0b10)
2
```

The underlying type of a Python integer, irrespective of the base used to specify it, is called int:

```
>>> type(10)
<class 'int'>
>>> type(0o10)
<class 'int'>
>>> type(0x10)
<class 'int'>
```

Floating-Point Numbers

The float type in Python designates a floating-point number. float values are specified with a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation:

```
>>> 4.2
4.2
>>> type(4.2)
<class 'float'>
>>> 4.
4.0
>>> .2
0.2

>>> .4e7
4000000.0
>>> type(.4e7)
<class 'float'>
>>> 4.2e-4
0.00042
```

Floating-Point Representation

The following is a bit more in-depth information on how Python represents floating-point numbers internally. You can readily use floating-point numbers in Python without understanding them to this level, so don't worry if this seems overly complicated. The information is presented here in case you are curious.

Almost all platforms represent Python float values as 64-bit "double-precision" values, according to the IEEE 754 standard. In that case, the maximum value a floating-point number can have is approximately 1.8×10^{308}. Python will indicate a number greater than that by the string inf:

```
>>> 1.79e308
1.79e+308
>>> 1.8e308
Inf
```

The closest a nonzero number can be to zero is approximately 5.0×10^{-324}. Anything closer to zero than that is effectively zero:

```
>>> 5e-324
5e-324
>>> 1e-325
0.0
```

Floating point numbers are represented internally as binary (base-2) fractions. Most decimal fractions cannot be represented exactly as binary fractions, so in most cases the internal representation of a floating-point number is an approximation of the actual value. In practice, the difference between the actual value and the represented value is very small and should not usually cause significant problems.

Complex Numbers

Complex numbers are specified as <real part>+<imaginary part>j. For example:

```
>>> 2+3j
(2+3j)
>>> type(2+3j)
<class 'complex'>
```

Strings

Strings are sequences of character data. The string type in Python is called str.

String literals may be delimited using either single or double quotes. All the characters between the opening delimiter and matching closing delimiter are part of the string:

```
>>> print("I am a string.")
I am a string.
>>> type("I am a string.")
<class 'str'>

>>> print('I am too.')
I am too.
>>> type('I am too.')
<class 'str'>
```

A string in Python can contain as many characters as you wish. The only limit is your machine's memory resources. A string can also be empty:

```
>>> ''
''
```

What if you want to include a quote character as part of the string itself? Your first impulse might be to try something like this:

```
>>> print('This string contains a single quote (') character.')
SyntaxError: invalid syntax
```

As you can see, that doesn't work so well. The string in this example opens with a single quote, so Python assumes the next single quote, the one in parentheses which was intended to be part of the string, is the closing delimiter. The final single quote is then a stray and causes the syntax error shown.
If you want to include either type of quote character within the string, the simplest way is to delimit the string with the other type. If a string is to contain a single quote, delimit it with double quotes and vice versa:

```
>>> print("This string contains a single quote (') character.")
This string contains a single quote (') character.

>>> print('This string contains a double quote (") character.')
This string contains a double quote (") character.
```

Escape Sequences in Strings

Sometimes, you want Python to interpret a character or sequence of characters within a string differently. This may occur in one of two ways:

- You may want to suppress the special interpretation that certain characters are usually given within a string.
- You may want to apply special interpretation to characters in a string which would normally be taken literally.

You can accomplish this using a backslash (\) character. A backslash character in a string indicates that one or more characters that follow it should be treated specially. (This is referred to as an escape sequence, because the backslash causes the subsequent character sequence to "escape" its usual meaning.)

Suppressing Special Character Meaning

You have already seen the problems you can come up against when you try to include quote characters in a string. If a string is delimited by single quotes, you can't directly specify a single quote character as part of the string because, for that string, the single quote has special meaning—it terminates the string:

```
>>> print('This string contains a single quote (') character.')
SyntaxError: invalid syntax
```

Specifying a backslash in front of the quote character in a string "escapes" it and causes Python to suppress its usual special meaning. It is then interpreted simply as a literal single quote character:

```
>>> print('This string contains a single quote (\') character.')
This string contains a single quote (') character.
```

The same works in a string delimited by double quotes as well:

```
>>> print("This string contains a double quote (\") character.")
This string contains a double quote (") character.
```

The following is a table of escape sequences which cause Python to suppress the usual special interpretation of a character in a string:

Escape Sequence	Usual Interpretation of Character(s) After Backslash	"Escaped" Interpretation
\'	Terminates string with single quote opening delimiter	Literal single quote (') character
\"	Terminates string with double quote opening delimiter	Literal double quote (") character
\<newline>	Terminates input line	Newline is ignored
\\	Introduces escape sequence	Literal backslash (\) character

Ordinarily, a newline character terminates line input. So pressing Enter in the middle of a string will cause Python to think it is incomplete:

```
>>> print('a

SyntaxError: EOL while scanning string literal
```

To break up a string over more than one line, include a backslash before each newline, and the newlines will be ignored:

```
>>> print('a\
... b\
... c')
Abc
```

To include a literal backslash in a string, escape it with a backslash:

```
>>> print('foo\\bar')
foo\bar
```

Applying Special Meaning to Characters

Next, suppose you need to create a string that contains a tab character in it. Some text editors may allow you to insert a tab character directly into your code. But many programmers consider that poor practice, for several reasons:

- The computer can distinguish between a tab character and a sequence of space characters, but you can't. To a human reading the code, tab and space characters are visually indistinguishable.
- Some text editors are configured to automatically eliminate tab characters by expanding them to the appropriate number of spaces.
- Some Python REPL environments will not insert tabs into code.

In Python (and almost all other common computer languages), a tab character can be specified by the escape sequence \t:

```
>>> print('foo\tbar')
foo    bar
```

The escape sequence \t causes the t character to lose its usual meaning, that of a literal t. Instead, the combination is interpreted as a tab character.

Here is a list of escape sequences that cause Python to apply special meaning instead of interpreting literally:

Escape Sequence	"Escaped" Interpretation
\a	ASCII Bell (BEL) character
\b	ASCII Backspace (BS) character
\f	ASCII Formfeed (FF) character
\n	ASCII Linefeed (LF) character
\N{<name>}	Character from Unicode database with given <name>
\r	ASCII Carriage Return (CR) character
\t	ASCII Horizontal Tab (TAB) character
\uxxxx	Unicode character with 16-bit hex value xxxx
\Uxxxxxxxx	Unicode character with 32-bit hex value xxxxxxxx
\v	ASCII Vertical Tab (VT) character
\ooo	Character with octal value ooo
\xhh	Character with hex value hh

Examples:

```
>>> print("a\tb")
a    b
>>> print("a\141\x61")
aaa
>>> print("a\nb")
a
b
>>> print('\u2192 \N{rightwards arrow}')
→ →
```

This type of escape sequence is typically used to insert characters that are not readily generated from the keyboard or are not easily readable or printable.

Raw Strings

A raw string literal is preceded by r or R, which specifies that escape sequences in the associated string are not translated. The backslash character is left in the string:

```
>>> print('foo\nbar')
```

```
foo
bar
>>> print(r'foo\nbar')
foo\nbar

>>> print('foo\\bar')
foo\bar
>>> print(R'foo\\bar')
foo\\bar
```

Triple-Quoted Strings

There is yet another way of delimiting strings in Python. Triple-quoted strings are delimited by matching groups of three single quotes or three double quotes. Escape sequences still work in triple-quoted strings, but single quotes, double quotes, and newlines can be included without escaping them. This provides a convenient way to create a string with both single and double quotes in it:

```
>>> print('''This string has a single (') and a double (") quote.''')
This string has a single (') and a double (") quote.
```

Because newlines can be included without escaping them, this also allows for multiline strings:

```
>>> print("""This is a
string that spans
across several lines""")
This is a
string that spans
across several lines
```

You will see in the upcoming tutorial on Python Program Structure how triple-quoted strings can be used to add an explanatory comment to Python code.

Boolean Type, Boolean Context, and "Truthiness"

Python 3 provides a Boolean data type. Objects of Boolean type may have one of two values, True or False:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

As you will see in upcoming tutorials, expressions in Python are often evaluated in Boolean context, meaning they are interpreted to represent truth or falsehood. A value that is true in Boolean context is sometimes said to be "truthy," and one that is false in Boolean context is said to be "falsy." (You may also see "falsy" spelled "falsey.")

The "truthiness" of an object of Boolean type is self-evident: Boolean objects that are equal to True are truthy (true), and those equal to False are falsy (false). But non-Boolean objects can be evaluated in Boolean context as well and determined to be true or false.

2) Solved Lab Activites

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*5*	*Low*	*CLO-6*
2	*5*	*Low*	*CLO-6*
3	*10*	*Medium*	*CLO-6*
4	*5*	*Low*	*CLO-6*
5	*5*	*Low*	*CLO-6*
6	*5*	*Low*	*CLO-6*
7	*10*	*Medium*	*CLO-6*

Activity 1:

Let us take an integer from user as input and check whether the given value is even or not. If the given value is not even then it means that it will be odd. So here we need to use if-else statement an demonstrated below

A. Create a new Python file from Python Shell and type the following code.
B. Run the code by pressing F5.

```
n=input("Enter a number ")
if int(n)%2==0:
    print("The given number is an even number")
else:
    print("The given number is an odd number")
```

Output

```
Enter a number 11
The given number is an odd number
>>>
```

Activity 2:

Write a Python code to keep accepting integer values from user until 0 is entered. Display sum of the given values.

Solution:

```
sum=0
s=input("Enter an integer value...")
n=int(s)
while n!=0:
    sum=sum+n
    s=input("Enter an integer value...")
    n=int(s)
print("Sum of given values is ",sum)
```

Output

```
Enter an integer value...10
Enter an integer value...521
Enter an integer value...5
Enter an integer value...22
Enter an integer value...0
Sum of given values is  558
>>>
```

Activity 3:

Write a Python code to accept an integer value from user and check that whether the given value is prime number or not.

Solution:

```
isPrime = True
i=2
n=int(input("enter a number"))
while i<n:
    remainder=n%i
    if remainder==0:
        isPrime=False
        break
    else:
        i=i+1

if isPrime:
    print("Number is Prime")
else:
    print("Number is not Prime")
```

Activity 4:

Accept 5 integer values from user and display their sum. Draw flowchart before coding in python.

Solution:

Create a new Python file from Python Shell and type the following code. Run the code by pressing F5.

```
summ = 0
i=0
while i<=4:
    s=input("enter a number")
    n=int(s)
    summ=summ+n
    i=i+1

print("sum is ",summ)
```

You will get the following output.

```
enter a number1
enter a number2
enter a number3
enter a number4
enter a number5
sum is  15
>>>
```

Activity 5:

Calculate the sum of all the values between 0-10 using while loop.

Solution:

Create a new Python file from Python Shell and type the following code.
Run the code by pressing F5.

```
summation = 0
i=1
while i<=10:
    summation=summation+i
    i=i+1

print("sum is ",summation)
```

You will get the following output.

```
sum is  55
>>>
```

Activity 6:

Take input from the keyboard and use it in your program.

Solution:

In Python and many other programming languages you can get user input. In Python the input() function will ask keyboard input from the user.The input function prompts text if a parameter is given. The function reads input from the keyboard, converts it to a string and removes the newline (Enter).Type and experiment with the script below.

```
#!/usr/bin/env python3

name = input('What is your name? ')
print('Hello ' + name)

job = input('What is your job? ')
print('Your job is ' + job)

num = input('Give me a number? ')
print('You said: ' + str(num))
```

Activity 7:

Generate a random number between 1 and 9 (including 1 and 9). Ask the user to guess the number, then tell them whether they guessed too low, too high, or exactly right. (Hint: remember to use the user input lessons from the very first exercise)

Extras:

Keep the game going until the user types "exit"

Keep track of how many guesses the user has taken, and when the game ends, print this out.

Solution:

```
import random
# Awroken

MINIMUM = 1
MAXIMUM = 9
NUMBER = random.randint(MINIMUM, MAXIMUM)
GUESS = None
ANOTHER = None
TRY = 0
RUNNING = True

print "Alright..."

while RUNNING:
   GUESS = raw_input("What is your lucky number? ")
   if int(GUESS) < NUMBER:
      print "Wrong, too low."
   elif int(GUESS) > NUMBER:
      print "Wrong, too high."
   elif GUESS.lower() == "exit":
      print "Better luck next time."
   elif int(GUESS) == NUMBER:
      print "Yes, that's the one, %s." % str(NUMBER)
      if TRY < 2:
         print "Impressive, only %s tries." % str(TRY)
      elif TRY > 2 and TRY < 10:
         print "Pretty good, %s tries." % str(TRY)
      else:
         print "Bad, %s tries." % str(TRY)
      RUNNING = False
   TRY += 1
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Lab Task 1:

Write a program that prompts the user to input an integer and then outputs the number with the digits reversed. For example, if the input is 12345, the output should be 54321.

Lab Task 2:

Write a program that reads a set of integers, and then prints the sum of the even and odd integers.

Lab Task 3:

Fibonacci series is that when you add the previous two numbers the next number is formed. You have to start from 0 and 1.
E.g. 0+1=1 → 1+1=2 → 1+2=3 → 2+3=5 → 3+5=8 → 5+8=13

So the series becomes
0 1 1 2 3 5 8 13 21 34 55 ……

Steps: You have to take an input number that shows how many terms to be displayed. Then use loops for displaying the Fibonacci series up to that term e.g. input no is =6 the output should be
0 1 1 2 3 5

Lab Task 4:

Write a Python code to accept marks of a student from 1-100 and display the grade according to the following formula.

Grade F if marks are less than 50
Grade E if marks are between 50 to 60
Grade D if marks are between 61 to 70
Grade C if marks are between 71 to 80
Grade B if marks are between 81 to 90
Grade A if marks are between 91 to 100

Lab Task 5:

Write a program that takes a number from user and calculate the factorial of that number.

Lab 02
Python Lists and Dictionaries

Objective:

This lab will give you practical implementation of different types of **sequences** including **lists, tuples, sets and dictionaries**. We will use lists alongside loops in order to know about indexing individual items of these containers. This lab will also allow students to write their own **functions.**

Activity Outcomes:

This lab teaches you the following topics:

- How to use lists, tuples, sets and dictionaries How to use loops with lists
- How to write customized functions

Instructor Note:

As a pre-lab activity, read Chapters 6, 10 and 14 from the book (*Introduction to Programming Using Python - Y. Liang (Pearson, 2013)*) to gain an insight about python programming and its fundamentals.

1) Useful Concepts

Python provides different types of data structures as sequences. In a sequence, there are more than one values and each value has its own index. The first value will have an index 0 in python, the second value will have index 1 and so on. These indices are used to access a particular value in the sequence.

Python Lists:

Lists are just like dynamically sized arrays, declared in other languages (vector in C++ and ArrayList in Java). Lists need not be homogeneous always which makes it the most powerful tool in Python. A single list may contain DataTypes like Integers, Strings, as well as Objects. Lists are mutable, and hence, they can be altered even after their creation.

List in Python are ordered and have a definite count. The elements in a list are indexed according to a definite sequence and the indexing of a list is done with 0 being the first index. Each element in the list has its definite place in the list, which allows duplicating of elements in the list, with each element having its own distinct place and credibility.

Creating a List

Lists in Python can be created by just placing the sequence inside the square brackets[]. Unlike Sets, a list doesn't need a built-in function for the creation of a list.

```
# Python program to demonstrate
# Creation of List

# Creating a List
List = []
print("Blank List: ")
print(List)

# Creating a List of numbers
List = [10, 20, 14]
print("\nList of numbers: ")
print(List)

# Creating a List of strings and accessing
# using index
List = ["Geeks", "For", "Geeks"]
print("\nList Items: ")
print(List[0])
print(List[2])

# Creating a Multi-Dimensional List
# (By Nesting a list inside a List)
List = [['Geeks', 'For'], ['Geeks']]
print("\nMulti-Dimensional List: ")
print(List)
```

Output:
Blank List:
[]

List of numbers:
[10, 20, 14]

List Items
Geeks
Geeks

Multi-Dimensional List:
[['Geeks', 'For'], ['Geeks']]

Creating a list with multiple distinct or duplicate elements

A list may contain duplicate values with their distinct positions and hence, multiple distinct or duplicate values can be passed as a sequence at the time of list creation.

```
# Creating a List with
# the use of Numbers
# (Having duplicate values)
List = [1, 2, 4, 4, 3, 3, 3, 6, 5]
print("\nList with the use of Numbers: ")
print(List)

# Creating a List with
# mixed type of values
# (Having numbers and strings)
List = [1, 2, 'Geeks', 4, 'For', 6, 'Geeks']
print("\nList with the use of Mixed Values: ")
print(List)
```

Output:
List with the use of Numbers:
[1, 2, 4, 4, 3, 3, 3, 6, 5]

List with the use of Mixed Values:
[1, 2, 'Geeks', 4, 'For', 6, 'Geeks']

Knowing the size of List

```
# Creating a List
List1 = []
```

```
print(len(List1))

# Creating a List of numbers
List2 = [10, 20, 14]
print(len(List2))
```

Output:
0
3

Adding Elements to a List
Using append() method
Elements can be added to the List by using the built-in **append()** function. Only one element at a time can be added to the list by using the append() method, for the addition of multiple elements with the append() method, loops are used. Tuples can also be added to the list with the use of the append method because tuples are immutable. Unlike Sets, Lists can also be added to the existing list with the use of the append() method.

```
# Python program to demonstrate
# Addition of elements in a List

# Creating a List
List = []
print("Initial blank List: ")
print(List)

# Addition of Elements
# in the List
List.append(1)
List.append(2)
List.append(4)
print("\nList after Addition of Three elements: ")
print(List)

# Adding elements to the List
# using Iterator
for i in range(1, 4):
    List.append(i)
print("\nList after Addition of elements from 1-3: ")
print(List)

# Adding Tuples to the List
List.append((5, 6))
print("\nList after Addition of a Tuple: ")
```

```
print(List)

# Addition of List to a List
List2 = ['For', 'Geeks']
List.append(List2)
print("\nList after Addition of a List: ")
print(List)
```

Output:

```
Initial blank List:
[]

List after Addition of Three elements:
[1, 2, 4]

List after Addition of elements from 1-3:
[1, 2, 4, 1, 2, 3]

List after Addition of a Tuple:
[1, 2, 4, 1, 2, 3, (5, 6)]

List after Addition of a List:
[1, 2, 4, 1, 2, 3, (5, 6), ['For', 'Geeks']]
```

Using insert() method

append() method only works for the addition of elements at the end of the List, for the addition of elements at the desired position, insert() method is used. Unlike append() which takes only one argument, the insert() method requires two arguments(position, value).

```
# Python program to demonstrate
# Addition of elements in a List

# Creating a List
List = [1,2,3,4]
print("Initial List: ")
print(List)

# Addition of Element at
# specific Position
# (using Insert Method)
List.insert(3, 12)
List.insert(0, 'Geeks')
print("\nList after performing Insert Operation: ")
print(List)
```

Output:
Initial List:
[1, 2, 3, 4]

List after performing Insert Operation:
['Geeks', 1, 2, 3, 12, 4]

Using extend() method

Other than append() and insert() methods, there's one more method for the Addition of elements, **extend()**, this method is used to add multiple elements at the same time at the end of the list.

```
# Python program to demonstrate
# Addition of elements in a List

# Creating a List
List = [1, 2, 3, 4]
print("Initial List: ")
print(List)

# Addition of multiple elements
# to the List at the end
# (using Extend Method)
List.extend([8, 'Geeks', 'Always'])
print("\nList after performing Extend Operation: ")
print(List)
```

Output:
Initial List:
[1, 2, 3, 4]

List after performing Extend Operation:
[1, 2, 3, 4, 8, 'Geeks', 'Always']

Accessing elements from the List

In order to access the list items refer to the index number. Use the index operator [ ] to access an item in a list. The index must be an integer. Nested lists are accessed using nested indexing.

```
# Python program to demonstrate
# accessing of element from list

# Creating a List with
# the use of multiple values
List = ["Geeks", "For", "Geeks"]
```

```
 # accessing a element from the
# list using index number
print("Accessing a element from the list")
print(List[0])
print(List[2])

# Creating a Multi-Dimensional List
# (By Nesting a list inside a List)
List = [['Geeks', 'For'], ['Geeks']]

# accessing an element from the
# Multi-Dimensional List using
# index number
print("Accessing a element from a Multi-Dimensional list")
print(List[0][1])
print(List[1][0])
```

Output:

```
Accessing a element from the list
Geeks
Geeks
Accessing a element from a Multi-Dimensional list
For
Geeks
```

Negative indexing

In Python, negative sequence indexes represent positions from the end of the array. Instead of having to compute the offset as in List[len(List)-3], it is enough to just write List[-3]. Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second-last item, etc.

```
List = [1, 2, 'Geeks', 4, 'For', 6, 'Geeks']

# accessing an element using negative indexing
print("Accessing element using negative indexing")

# print the last element of list
print(List[-1])

# print the third last element of list
print(List[-3])
```

Output:

```
Accessing element using negative indexing
Geeks
For
```

Removing Elements from the List

Using remove() method

Elements can be removed from the List by using the built-in **remove()** function but an Error arises if the element doesn't exist in the list. Remove() method only removes one element at a time, to remove a range of elements, the iterator is used. The remove() method removes the specified item.

```
# Python program to demonstrate
# Removal of elements in a List

# Creating a List
List = [1, 2, 3, 4, 5, 6,
        7, 8, 9, 10, 11, 12]
print("Initial List: ")
print(List)

# Removing elements from List
# using Remove() method
List.remove(5)
List.remove(6)
print("\nList after Removal of two elements: ")
print(List)

# Removing elements from List
# using iterator method
for i in range(1, 5):
    List.remove(i)
print("\nList after Removing a range of elements: ")
print(List)
```

Output:

```
Initial List:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

List after Removal of two elements:
[1, 2, 3, 4, 7, 8, 9, 10, 11, 12]

List after Removing a range of elements:
[7, 8, 9, 10, 11, 12]
```

Using pop() method

Pop() function can also be used to remove and return an element from the list, but by default it removes only the last element of the list, to remove an element from a specific position of the List, the index of the element is passed as an argument to the pop() method.

```
List = [1,2,3,4,5]
 # Removing element from the
```

```
# Set using the pop() method
List.pop()
print("\nList after popping an element: ")
print(List)

# Removing element at a
# specific location from the
# Set using the pop() method
List.pop(2)
print("\nList after popping a specific element: ")
print(List)
```

Output:

List after popping an element:
[1, 2, 3, 4]

List after popping a specific element:
[1, 2, 4]

Slicing of a List

In Python List, there are multiple ways to print the whole List with all the elements, but to print a specific range of elements from the list, we use the Slice operation. Slice operation is performed on Lists with the use of a colon(:). To print elements from beginning to a range use [: Index], to print elements from end-use [:-Index], to print elements from specific Index till the end use [Index:], to print elements within a range, use [Start Index:End Index] and to print the whole List with the use of slicing operation, use [:]. Further, to print the whole List in reverse order, use [::-1].

Note – To print elements of List from rear-end, use Negative Indexes.

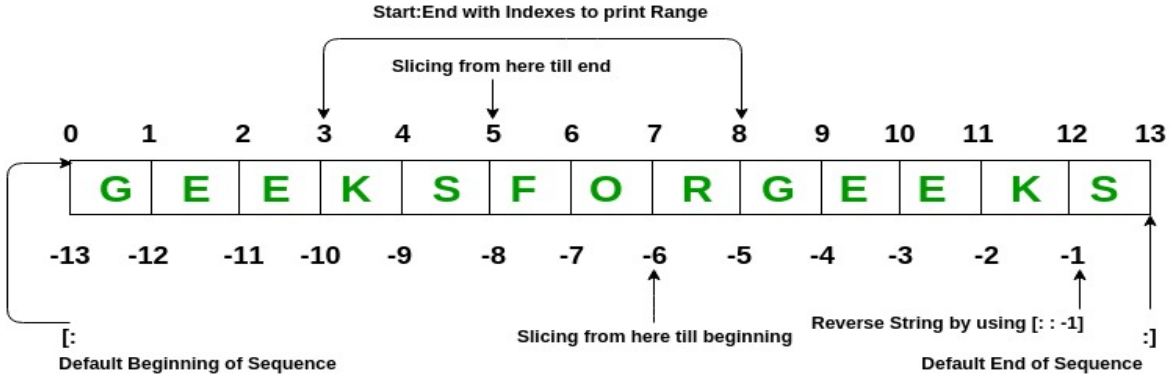


Figure 1 - List

```
# Python program to demonstrate
# Removal of elements in a List

# Creating a List
List = ['G', 'E', 'E', 'K', 'S', 'F',
        'O', 'R', 'G', 'E', 'E', 'K', 'S']
```

```
print("Initial List: ")
print(List)

# Print elements of a range
# using Slice operation
Sliced_List = List[3:8]
print("\nSlicing elements in a range 3-8: ")
print(Sliced_List)

# Print elements from a
# pre-defined point to end
Sliced_List = List[5:]
print("\nElements sliced from 5th "
      "element till the end: ")
print(Sliced_List)

# Printing elements from
# beginning till end
Sliced_List = List[:]
print("\nPrinting all elements using slice operation: ")
print(Sliced_List)
```

Output:

```
Initial List:
['G', 'E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

Slicing elements in a range 3-8:
['K', 'S', 'F', 'O', 'R']

Elements sliced from 5th element till the end:
['F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

Printing all elements using slice operation:
['G', 'E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']
```

Negative index List slicing

```
# Creating a List
List = ['G', 'E', 'E', 'K', 'S', 'F',
        'O', 'R', 'G', 'E', 'E', 'K', 'S']
print("Initial List: ")
print(List)

# Print elements from beginning
# to a pre-defined point using Slice
```

```
Sliced_List = List[:-6]
print("\nElements sliced till 6th element from last: ")
print(Sliced_List)

# Print elements of a range
# using negative index List slicing
Sliced_List = List[-6:-1]
print("\nElements sliced from index -6 to -1")
print(Sliced_List)

# Printing elements in reverse
# using Slice operation
Sliced_List = List[::-1]
print("\nPrinting List in reverse: ")
print(Sliced_List)
```

Output:

```
Initial List:
['G', 'E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

Elements sliced till 6th element from last:
['G', 'E', 'E', 'K', 'S', 'F', 'O']

Elements sliced from index -6 to -1
['R', 'G', 'E', 'E', 'K']

Printing List in reverse:
['S', 'K', 'E', 'E', 'G', 'R', 'O', 'F', 'S', 'K', 'E', 'E', 'G']
```

List Comprehension

List comprehensions are used for creating new lists from other iterables like tuples, strings, arrays, lists, etc.

A list comprehension consists of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element.

Syntax:

newList = [expression(element) for element in oldList if condition]

Example:

```
# Python program to demonstrate list comprehension in Python
 # below list contains square of all odd numbers from range 1 to 10

odd_square = [x ** 2 for x in range(1, 11) if x % 2 == 1]
print(odd_square)
```

Output:

```
[1, 9, 25, 49, 81]
```

For better understanding, the above code is similar to –

```
# for understanding, above generation is same as,
odd_square = []

for x in range(1, 11):
   if x % 2 == 1:
      odd_square.append(x**2)

print(odd_square)
```

Output:

[1, 9, 25, 49, 81]

Dictionary in Python is an unordered collection of data values, used to store data values like a map, which, unlike other Data Types that hold only a single value as an element, Dictionary holds **key:value** pair. Key-value is provided in the dictionary to make it more optimized.

Note – Keys in a dictionary don't allow Polymorphism.

Disclamer: *It is important to note that Dictionaries have been modified to maintain insertion order with the release of Python 3.7, so they are now ordered collection of data values.*

Creating a Dictionary

In Python, a Dictionary can be created by placing a sequence of elements within curly **{}** braces, separated by 'comma'. Dictionary holds pairs of values, one being the Key and the other corresponding pair element being its **Key:value**. Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated and must be *immutable*.

```
# Creating a Dictionary
# with Integer Keys
Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
print("\nDictionary with the use of Integer Keys: ")
print(Dict)

# Creating a Dictionary
# with Mixed keys
Dict = {'Name': 'Geeks', 1: [1, 2, 3, 4]}
print("\nDictionary with the use of Mixed Keys: ")
print(Dict)
```

Output:

Dictionary with the use of Integer Keys:

{1: 'Geeks', 2: 'For', 3: 'Geeks'}

Dictionary with the use of Mixed Keys:

{1: [1, 2, 3, 4], 'Name': 'Geeks'}

Dictionary can also be created by the built-in function dict(). An empty dictionary can be created by just placing to curly braces{}.

```
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)

# Creating a Dictionary
# with dict() method
Dict = dict({1: 'Geeks', 2: 'For', 3:'Geeks'})
print("\nDictionary with the use of dict(): ")
print(Dict)

# Creating a Dictionary
# with each item as a Pair
Dict = dict([(1, 'Geeks'), (2, 'For')])
print("\nDictionary with each item as a pair: ")
print(Dict)
```

Output:

```
Empty Dictionary:
{}
Dictionary with the use of dict():
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
Dictionary with each item as a pair:
{1: 'Geeks', 2: 'For'}
```

Nested Dictionary:

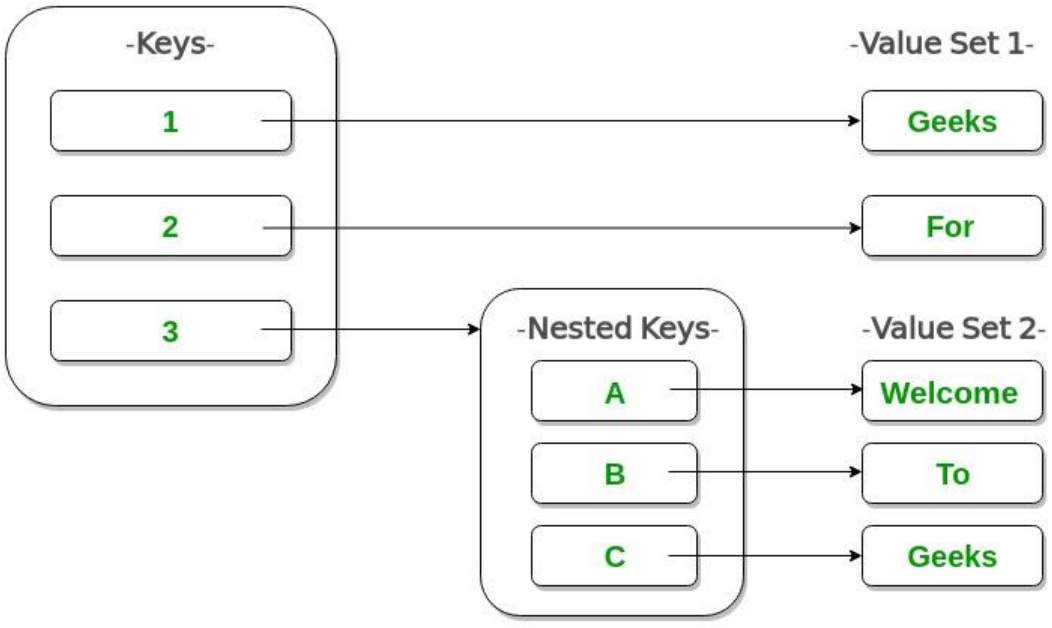


Figure 2 - Dictionary

```
# Creating a Nested Dictionary
# as shown in the below image
Dict = {1: 'Geeks', 2: 'For',
       3:{'A' : 'Welcome', 'B' : 'To', 'C' : 'Geeks'}}

print(Dict)
```

Output:

```
{1: 'Geeks', 2: 'For', 3: {'A': 'Welcome', 'B': 'To', 'C': 'Geeks'}}
```

Adding elements to a Dictionary

In Python Dictionary, the Addition of elements can be done in multiple ways. One value at a time can be added to a Dictionary by defining value along with the key e.g. Dict[Key] = 'Value'. Updating an existing value in a Dictionary can be done by using the built-in **update()** method. Nested key values can also be added to an existing Dictionary.

Note- While adding a value, if the key-value already exists, the value gets updated otherwise a new Key with the value is added to the Dictionary.

```
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)

# Adding elements one at a time
Dict[0] = 'Geeks'
Dict[2] = 'For'
Dict[3] = 1
print("\nDictionary after adding 3 elements: ")
print(Dict)

# Adding set of values
# to a single Key
Dict['Value_set'] = 2, 3, 4
print("\nDictionary after adding 3 elements: ")
print(Dict)

# Updating existing Key's Value
Dict[2] = 'Welcome'
print("\nUpdated key value: ")
print(Dict)

# Adding Nested Key value to Dictionary
Dict[5] = {'Nested' :{'1' : 'Life', '2' : 'Geeks'}}
```

```
print("\nAdding a Nested Key: ")
print(Dict)
```

Output:

```
Empty Dictionary:
{}

Dictionary after adding 3 elements:
{0: 'Geeks', 2: 'For', 3: 1}

Dictionary after adding 3 elements:
{0: 'Geeks', 2: 'For', 3: 1, 'Value_set': (2, 3, 4)}

Updated key value:
{0: 'Geeks', 2: 'Welcome', 3: 1, 'Value_set': (2, 3, 4)}

Adding a Nested Key:
{0: 'Geeks', 2: 'Welcome', 3: 1, 5: {'Nested': {'1': 'Life', '2': 'Geeks'}}, 'Value_set': (2, 3, 4)}
```

Accessing elements from a Dictionary

In order to access the items of a dictionary refer to its key name. Key can be used inside square brackets.

```
# Python program to demonstrate
# accessing a element from a Dictionary

# Creating a Dictionary
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

# accessing a element using key
print("Accessing a element using key:")
print(Dict['name'])

# accessing a element using key
print("Accessing a element using key:")
print(Dict[1])
```

Output:

```
Accessing a element using key:
For
Accessing a element using key:
Geeks
```

There is also a method called **get()** that will also help in accessing the element from a dictionary.

```
# Creating a Dictionary
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

# accessing a element using get()
# method
print("Accessing a element using get:")
print(Dict.get(3))
```

Output:

Accessing a element using get:
Geeks

Accessing an element of a nested dictionary

In order to access the value of any key in the nested dictionary, use indexing [] syntax.

```
# Creating a Dictionary
Dict = {'Dict1': {1: 'Geeks'},
        'Dict2': {'Name': 'For'}}

# Accessing element using key
print(Dict['Dict1'])
print(Dict['Dict1'][1])
print(Dict['Dict2']['Name'])
```

Output:

{1: 'Geeks'}
Geeks
For

Removing Elements from Dictionary

Using del keyword

In Python Dictionary, deletion of keys can be done by using the **del** keyword. Using the del keyword, specific values from a dictionary as well as the whole dictionary can be deleted. Items in a Nested dictionary can also be deleted by using the del keyword and providing a specific nested key and particular key to be deleted from that nested Dictionary.

```
# Initial Dictionary
Dict = { 5 : 'Welcome', 6 : 'To', 7 : 'Geeks',
        'A' : {1 : 'Geeks', 2 : 'For', 3 : 'Geeks'},
        'B' : {1 : 'Geeks', 2 : 'Life'}}
print("Initial Dictionary: ")
print(Dict)

# Deleting a Key value
del Dict[6]
```

```
print("\nDeleting a specific key: ")
print(Dict)

# Deleting a Key from
# Nested Dictionary
del Dict['A'][2]
print("\nDeleting a key from Nested Dictionary: ")
print(Dict)
```

Output:

Initial Dictionary:
{'A': {1: 'Geeks', 2: 'For', 3: 'Geeks'}, 'B': {1: 'Geeks', 2: 'Life'}, 5: 'Welcome', 6: 'To', 7: 'Geeks'}

Deleting a specific key:
{'A': {1: 'Geeks', 2: 'For', 3: 'Geeks'}, 'B': {1: 'Geeks', 2: 'Life'}, 5: 'Welcome', 7: 'Geeks'}

Deleting a key from Nested Dictionary:
{'A': {1: 'Geeks', 3: 'Geeks'}, 'B': {1: 'Geeks', 2: 'Life'}, 5: 'Welcome', 7: 'Geeks'}

Using pop() method

Pop() method is used to return and delete the value of the key specified.

```
# Creating a Dictionary
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

# Deleting a key
# using pop() method
pop_ele = Dict.pop(1)
print('\nDictionary after deletion: ' + str(Dict))
print('Value associated to poped key is: ' + str(pop_ele))
```

Output:

Dictionary after deletion: {3: 'Geeks', 'name': 'For'}
Value associated to poped key is: Geeks

Using popitem() method

The popitem() returns and removes an arbitrary element (key, value) pair from the dictionary.

```
# Creating Dictionary
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

# Deleting an arbitrary key
# using popitem() function
pop_ele = Dict.popitem()
print("\nDictionary after deletion: " + str(Dict))
```

```
print("The arbitrary pair returned is: " + str(pop_ele))
```

Output:

```
Dictionary after deletion: {3: 'Geeks', 'name': 'For'}
The arbitrary pair returned is: (1, 'Geeks')
```

Using clear() method

All the items from a dictionary can be deleted at once by using **clear()** method.

```
# Creating a Dictionary
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}


# Deleting entire Dictionary
Dict.clear()
print("\nDeleting Entire Dictionary: ")
print(Dict)
```

Output:

```
Deleting Entire Dictionary:
{}
```

Dictionary Methods

Methods	Description
copy()	They copy() method returns a shallow copy of the dictionary.
clear()	The clear() method removes all items from the dictionary.
pop()	Removes and returns an element from a dictionary having the given key.
popitem()	Removes the arbitrary key-value pair from the dictionary and returns it as tuple.
get()	It is a conventional method to access a value for a key.
dictionary_name.values()	returns a list of all the values available in a given dictionary.
str()	Produces a printable string representation of a dictionary.
update()	Adds dictionary dict2's key-values pairs to dict
setdefault()	Set dict[key]=default if key is not already in dict
keys()	Returns list of dictionary dict's keys
items()	Returns a list of dict's (key, value) tuple pairs
has_key()	Returns true if key in dictionary dict, false otherwise
fromkeys()	Create a new dictionary with keys from seq and values set to value.
type()	Returns the type of the passed variable.
cmp()	Compares elements of both dict.

2) Solved Lab Activites:

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*10*	*Low*	*CLO-6*
2	*10*	*Low*	*CLO-6*
3	*10*	*Low*	*CLO-6*
4	*15*	*Medium*	*CLO-6*
5	*15*	*Medium*	*CLO-6*
6	*15*	*Medium*	*CLO-6*

Activity 1

Accept two lists from user and display their join.

Solution:

```
myList1=[]
print("Enter objects of first list...")
for i in range(5):
    val=input("Enter a value:")
    n=int(val)
    myList1.append(n)

myList2=[]
print("Enter objects of second list...")
for i in range(5):
    val=input("Enter a value:")
    n=int(val)
    myList2.append(n)

list3=myList1+myList2;
print(list3)
```

You will get the following output.

```
Enter objects of first list...
Enter a value:1
Enter a value:2
Enter a value:3
Enter a value:4
Enter a value:5
Enter objects of second list...
Enter a value:6
Enter a value:7
Enter a value:8
Enter a value:9
Enter a value:0
[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>>
```

Activity 2:

A palindrome is a string which is same read forward or backwards.
For example: "dad" is the same in forward or reverse direction. Another example is "aibohphobia" which literally means, an irritable fear of palindromes.
Write a function in python that receives a string and returns True if that string is a palindrome and False otherwise. Remember that difference between upper and lower case characters are ignored during this determination.

Solution:

```
def isPalindrome(word):
    temp=word[::-1]
    if temp.capitalize()==word.capitalize():
        return True
    else:
        return False


print(isPalindrome("deed"))
```

Activity 3:

Imagine two matrices given in the form of 2D lists as under; a = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
b = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

Write a python code that finds another matrix/2D list that is a product of and b, i.e., C=a*b
Solution:

```
for indrow in range (3):
    c.append ([])
    for indcol in range(3):
        c[indrow].append (0)
        for indaux in range (3):
            c[indrow][indcol] += a[indrow][indaux] * b[indcol][inc

print (c)
```

Activity 4:

A closed polygon with N sides can be represented as a list of tuples of N connected coordinates, i.e., [(x_1,y_1), (x_2,y_2), (x_3,y_3), . . . , (x_N,y_N)]. A sample polygon with 6 sides (N=6) is shown below.

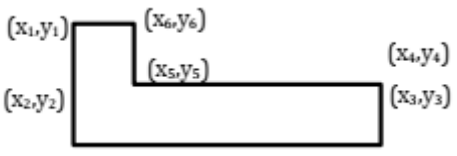


Figure 3 - Polygon

Write a python function that takes a list of N tuples as input and returns the perimeter of the polygon. Remember that your code should work for any value of N.

Hint: A perimeter is the sum of all sides of a polygon.

Solution:

```
def perimeter(listing):
    leng=len(listing)
    perimeter=0;
    for i in range(0,leng-1):
        dist = (((listing[i][0]-listing[i+1][0])**2)+
                ((listing[i][1]-listing[i+1][1])**2))**0.5
        perimeter = perimeter + dist
    perimeter = perimeter + (((listing[0][0]-listing[leng-1][0])**2)
                             +((listing[0][1]-listing[leng-1][1])**2))**0.5
    return perimeter

L = [(1,3), (2,7), (3,9), (-1,8)]
print(perimeter(L))
```

Activity 5:

Imagine two sets A and B containing numbers. Without using built-in set functionalities, write your own function that receives two such sets and returns another set C which is a symmetric difference of the two input sets. (A symmetric difference between A and B will return a set C which contains only those items that appear in one of A or B. Any items that appear in both sets are not included in C). Now compare the output of your function with the following built-in functions/operators.

- ✓ ***A.symmetric_difference(B)***
- ✓ ***B.symmetric_difference(A)***
- ✓ ***A ^ B***
- ✓ ***B ^ A***

Solution:

```
#Function defined
def symmDiff(a,b):
    e=set()   #empty set
    for i in a:   #for loop used to access in a
        if i  not in b:
            e.add(i)
    for i in b: #for loop used to access in b
        if i  not in a:
            e.add(i)
    return e


set1={0,1,2,4,5}
set2={4,5,7,8,9}
print(symmDiff(set1,set2))

#verification using inbuilt function
print(set1.symmetric_difference(set2))
print(set2.symmetric_difference(set1))
print(set1^set2)
print(set2^set1)
```

Activity 6:

Create a Python program that contains a dictionary of names and phone numbers. Use a tuple of separate first and last name values for the key field. Initialize the dictionary with at least three names and numbers. Ask the user to search for a phone number by entering a first and last name. Display the matching number if found, or a message if not found.

Solution:

```
sample={("sohaib","ali"):"0246585468445", ("aib","li"):"02465854645",
        ("sib","ai"):"0246585468445",}
firstName = input("enter first name")
lastName = input("enter last name")

searchTuple = (firstName, lastName)
if searchTuple in sample:
    print(sample[searchTuple])
else:
    print("name not found")
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1:

Create two lists based on the user values. Merge both the lists and display in sorted order.

Lab Task 2:

Repeat the above activity to find the smallest and largest element of the list. (Suppose all the elements are integer values)

Lab Task 3:

The derivate of a function f(x) is a measurement of how quickly the function f changes with respect to change in its domain x. This measurement can be approximated by the following relation,

$$\frac{d}{dx}f(x) = \frac{f(x+h) - \mathrm{f}(x)}{h}$$

Where h represents a small increment in x. You have to prove the following relation

$$\frac{d}{dx}\sin(x) = \cos(x)$$

Imagine x being a list that goes from –pi to pi with an increment of 0.001. You can approximate the derivative by using the following approximation,

$$\frac{d}{dx}\sin(x) = \frac{sin(x+h) - \sin(x)}{h}$$

*In your case, assume h = 0.001. That is at each point in x, compute the right hand side of above equation and compare whether the output value is equivalent to cos(x). Also print the corresponding values of ( ) and cos(x) for every point. Type ‘’from math import *’’ at the start of your program to use predefined values of pi, and sin and cos functions. What happens if you increase the interval h from 0.001 to 0.01 and then to 0.1?*

Lab Task 4:

For this exercise, you will keep track of when our friend’s birthdays are, and be able to find that information based on their name. Create a dictionary (in your file) of names and birthdays. When you run your program it should ask the user to enter a name, and return the birthday of that person back to them. The interaction should look something like this:

>>> Welcome to the birthday dictionary. We know the birthdays of:
Albert Einstein
Benjamin Franklin
Ada Lovelace

>>> Who's birthday do you want to look up?
Benjamin Franklin
>>> Benjamin Franklin's birthday is 01/17/1706.

Lab Task 5:

Create a dictionary by extracting the keys from a given dictionary

Write a Python program to create a new dictionary by extracting the mentioned keys from the below dictionary.

Given dictionary:

```
sample_dict = {
   "name": "Kelly",
   "age": 25,
   "salary": 8000,
   "city": "New york"}

# Keys to extract
keys = ["name", "salary"]
```

Expected output:

{'name': 'Kelly', 'salary': 8000}

Lab 03
Breadth First Search

Objective:

This lab will introduce students to search problems. We will first start by representing problems in terms of state space graph. Given a state space graph, starting state and a goal state, students will then perform a basic **Breadth First Search** solution within that graph. The output will be a set of actions or a path that will begin from initial state/node and end in the goal node**.**

Activity Outcomes:

This lab teaches you the following topics:

- How to represent problems in terms of state space graph
- How to use find a solution of those problems using Breadth First Search.

Instructor Note:

As pre-lab activity, read Chapter 3 from the book (Artificial Intelligence, A Modern Approach by Peter Norvig, 3rd edition) to know the basics of search algorithms.

1) Useful Concepts

Sometimes, very different-sounding problems turn out to be similar when you think about how to solve them. What do Pac-Man, the royal family of Britain, and driving to Orlando have in common? They all involve route-finding or path-search problems:

How is the current Prince William related to King William III, who endowed the College of William and Mary in 1693?

What path should a ghost follow to get to Pac-Man as quickly as possible? What's the best way to drive from Dallas, Texas to Orlando, Florida?

We have to be given some information to answer any of these questions. This information about each question can be represented in terms of graph. This lab will enable students to represent problems in terms of graph and then finding a solution in terms of a path using breadth first search

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

Consider the following diagram.

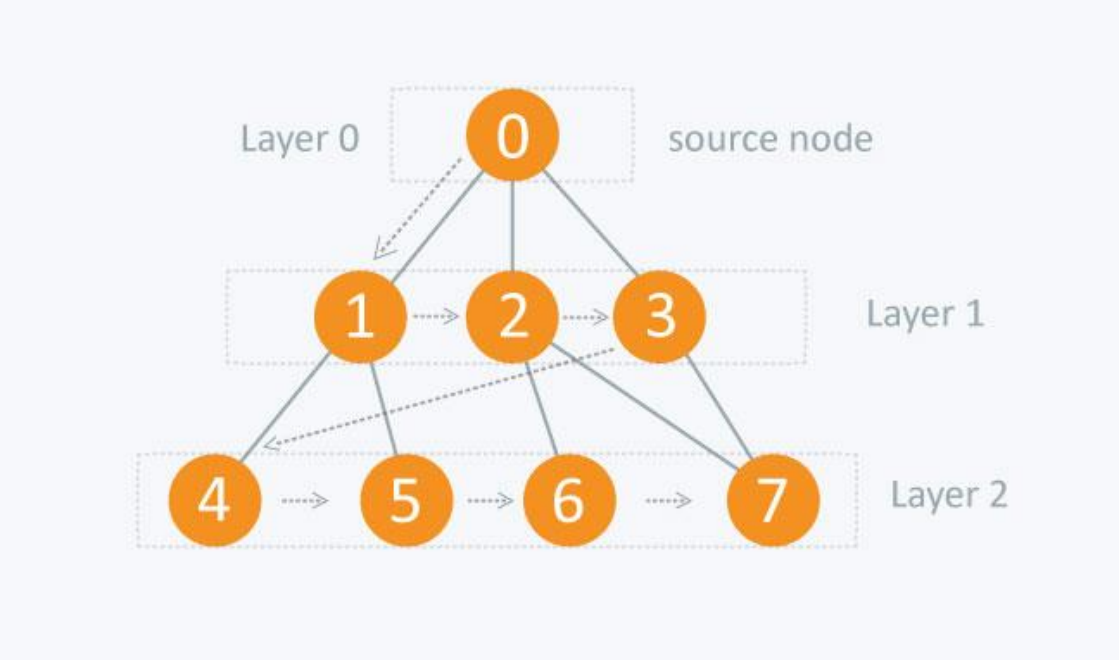


Figure 4 - Graph

The distance between the nodes in layer 1 is comparitively lesser than the distance between the nodes in layer 2. Therefore, in BFS, you must traverse all the nodes in layer 1 before you move to the nodes in layer 2.

Traversing child nodes

A graph can contain cycles, which may bring you to the same node again while traversing the graph. To avoid processing of same node again, use a boolean array which marks the node after it is

processed. While visiting the nodes in the layer of a graph, store them in a manner such that you can traverse the corresponding child nodes in a similar order.
In the earlier diagram, start traversing from 0 and visit its child nodes 1, 2, and 3. Store them in the order in which they are visited. This will allow you to visit the child nodes of 1 first (i.e. 4 and 5), then of 2 (i.e. 6 and 7), and then of 3 (i.e. 7) etc.
To make this process easy, use a queue to store the node and mark it as 'visited' until all its neighbours (vertices that are directly connected to it) are marked. The queue follows the First In First Out (FIFO) queuing method, and therefore, the neigbors of the node will be visited in the order in which they were inserted in the node i.e. the node that was inserted first will be visited first, and so on.

Pseudocode

```
BFS (G, s)                  //Where G is the graph and s is the source node
    let Q be queue.
    Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.

    mark s as visited.
    while ( Q is not empty)
        //Removing that vertex from queue,whose neighbour will be visited now
        v  =  Q.dequeue( )

      //processing all the neighbours of v
      for all neighbours w of v in Graph G
          if w is not visited
                  Q.enqueue( w )          //Stores w in Q to further visit its neighbour
                  mark w as visited.
```

Traversing process

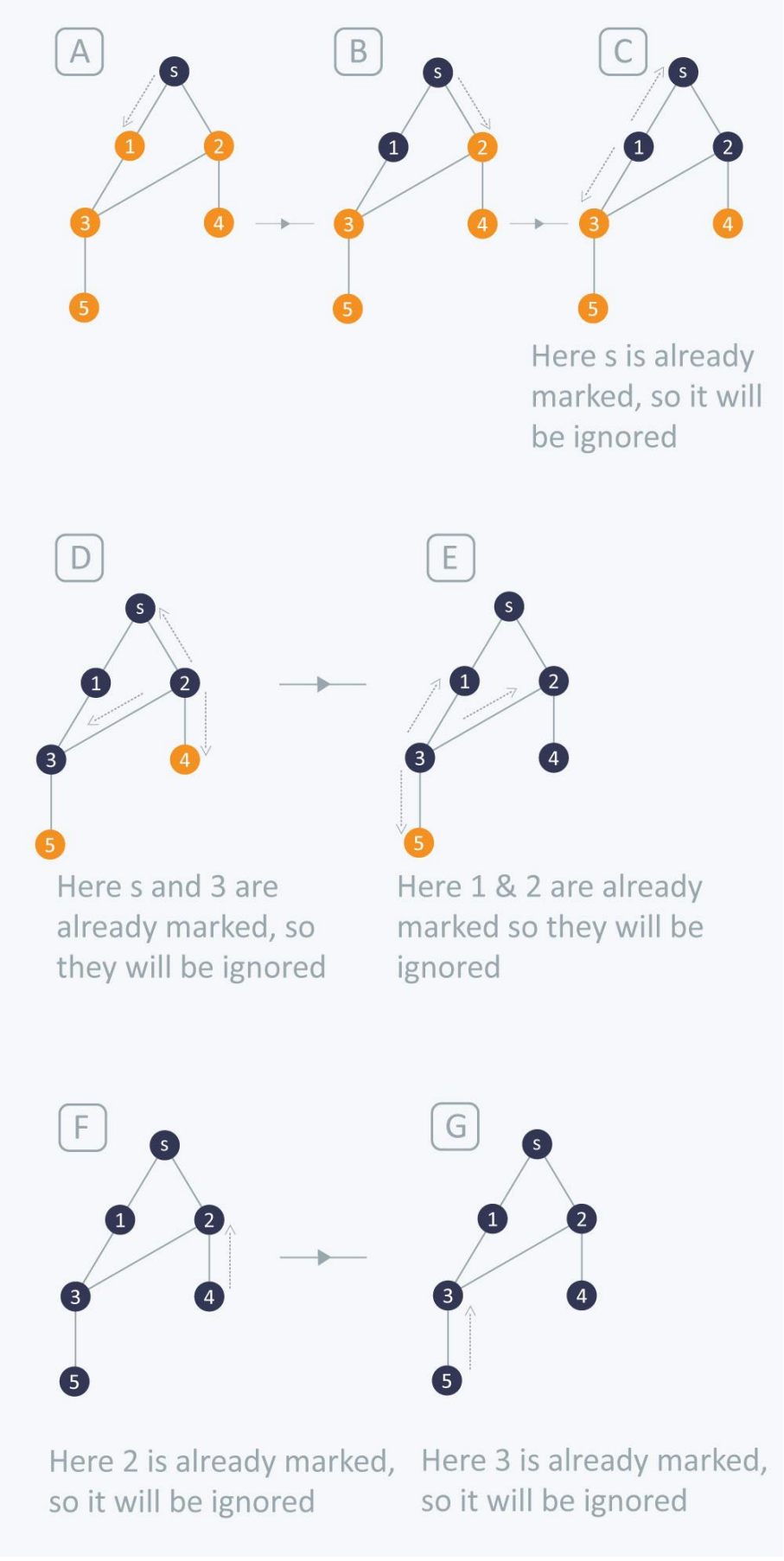


Figure 5 - Traversal

The traversing will start from the source node and push *s* in queue. *s* will be marked as 'visited'.

First iteration

- s will be popped from the queue
- Neighbors of s i.e. 1 and 2 will be traversed
- 1 and 2, which have not been traversed earlier, are traversed. They will be:
  - Pushed in the queue
  - 1 and 2 will be marked as visited

Second iteration

- 1 is popped from the queue
- Neighbors of 1 i.e. s and 3 are traversed
- s is ignored because it is marked as 'visited'
- 3, which has not been traversed earlier, is traversed. It is:
  - Pushed in the queue
  - Marked as visited

Third iteration

- 2 is popped from the queue
- Neighbors of 2 i.e. s, 3, and 4 are traversed
- 3 and s are ignored because they are marked as 'visited'
- 4, which has not been traversed earlier, is traversed. It is:
  - Pushed in the queue
  - Marked as visited

Fourth iteration

- 3 is popped from the queue
- Neighbors of 3 i.e. 1, 2, and 5 are traversed
- 1 and 2 are ignored because they are marked as 'visited'
- 5, which has not been traversed earlier, is traversed. It is:
  - Pushed in the queue
  - Marked as visited

Fifth iteration

- 4 will be popped from the queue
- Neighbors of 4 i.e. 2 is traversed
- 2 is ignored because it is already marked as 'visited'

Sixth iteration

- 5 is popped from the queue
- Neighbors of 5 i.e. 3 is traversed
- 3 is ignored because it is already marked as 'visited'

The queue is empty and it comes out of the loop. All the nodes have been traversed by using BFS. If all the edges in a graph are of the same weight, then BFS can also be used to find the minimum distance between the nodes in a graph.

Example

As in this diagram, start from the source node, to find the distance between the source node and node 1. If you do not follow the BFS algorithm, you can go from the source node to node 2 and then to node 1. This approach will calculate the distance between the source node and node 1 as 2, whereas, the minimum distance is

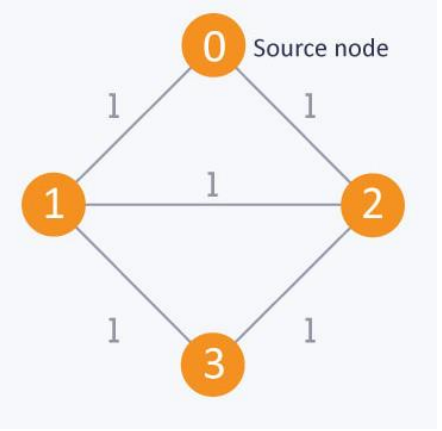


Figure 6 - Graph

actually 1. The minimum distance can be calculated correctly by using the BFS algorithm.

Complexity

The time complexity of BFS is **O(V + E)**, where V is the number of nodes and E is the number of edges.

Applications

1. How to determine the level of each node in the given tree?

As you know in BFS, you traverse level wise. You can also use BFS to determine the level of each node.

Implementation

```
vector <int> v[10] ;   //Vector for maintaining adjacency list explained above
 int level[10]; //To determine the level of each node
 bool vis[10]; //Mark the node if visited
 void bfs(int s) {
    queue <int> q;
    q.push(s);
    level[ s ] = 0 ;  //Setting the level of the source node as 0
    vis[ s ] = true;
    while(!q.empty())
    {
       int p = q.front();
       q.pop();
       for(int i = 0;i < v[ p ].size() ; i++)
       {
          if(vis[ v[ p ][ i ] ] == false)
          {
       //Setting the level of each node with an increment in the level of parent node
             level[ v[ p ][ i ] ] = level[ p ]+1;
              q.push(v[ p ][ i ]);
              vis[ v[ p ][ i ] ] = true;
  }
       }
    }
 }
```

This code is similar to the BFS code with only the following difference:

level[v[p][i]] = level[p]+1;

In this code, while you visit each node, the level of that node is set with an increment in the level of its parent node. This is how the level of each node is determined.

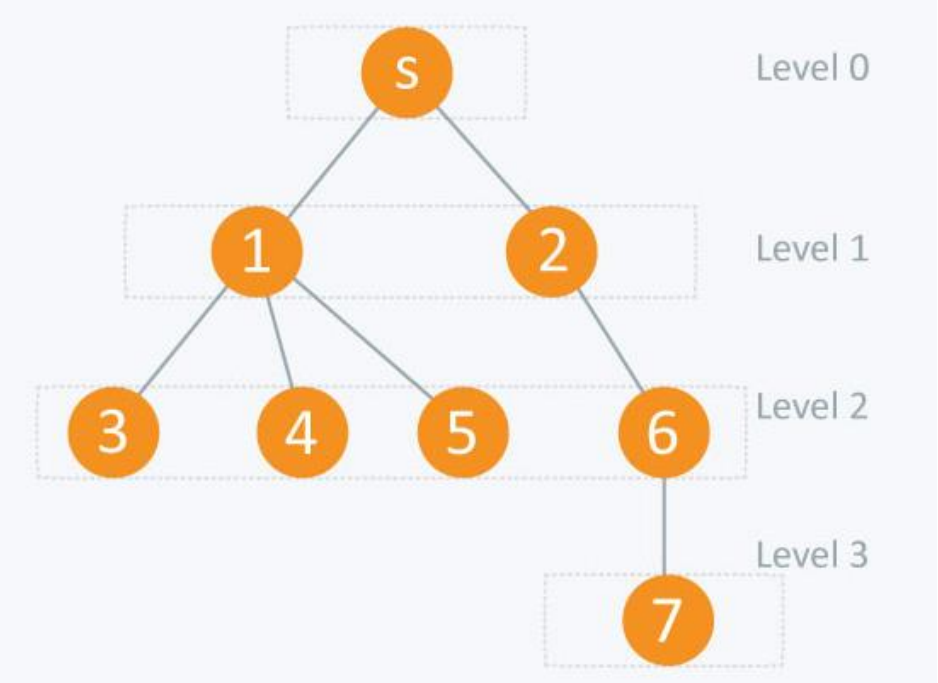


Figure 7 - Graph

```
node                level [node]

s (source node)         0
1                   1
2                   1
3                   2
4                   2
5                   2
6                   2
7                   3
```

2) Solved Lab Activities:

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*45*	*High*	*CLO-6*

Activity 1:

Consider a toy problem that can be represented as a following graph. How would you represent this graph in python?

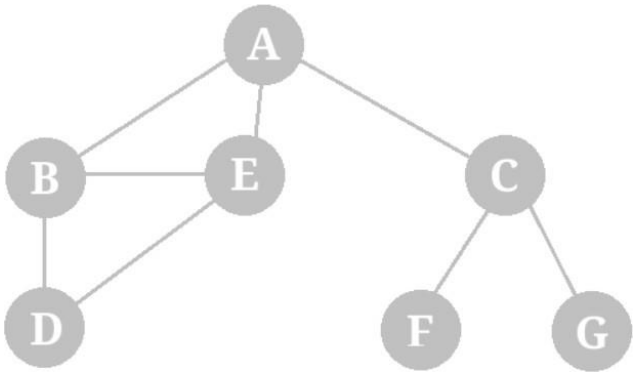


Figure 8 - Graph

Solution:

Remember a node of a tree can be represented by four attributes. We consider node as a class having four attributes namely,

1. State of the node
2. Parent of the node
3. Actions applicable to that node
4. The total path cost of that node starting from the initial state until that particular node is reached

```
class Node:
    #state = state          # class variable shared by all instances
    def __init__(self, state, parent, actions, totalCost):
        self.state = state     # instance variable unique to each insta
        self.parent = parent
        self.actions = actions # we are not saving actions themselves,
                               # only output states of those actions
        self.totalCost = totalCost
```

We can now implement this class in a dictionary. This dictionary will represent our state space graph. As we will traverse through the graph, we will keep updating parent and cost of each node.

```
# we think of a graph as a dictionary, items comprise of nodes, where
# each node has a key and a value. Key is simply the state of the node
# and value are actual attributes that node object

graph = {'A': Node('A', None, ['B', 'C', 'E'], None),
         'B': Node('B', None, ['A','D', 'E'], None),
         'C': Node('C', None, ['A', 'F', 'G'], None),
         'D': Node('D', None, ['B', 'E'], None),
         'E': Node('E', None, ['A', 'B','D'], None),
         'F': Node('F', None, ['C'], None),
         'G': Node('G', None, ['C'], None)}
```

Activity 1 - b

For the graph in previous activity, imagine node A as starting node and your goal is to reach F. Keeping breadth first search in mind, describe a sequence of actions that you must take to reach that goal state.

Solution:
Remember that in theory class, we discussed the following implementation of breadth first search.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

What follows is python implementation of above mentioned algorithm,

```
def BFS():
    initialState = 'D'
    goalState = 'F'

    # we think of a graph as a dictionary, items comprise of nodes, where
    # each node has a key and a value. Key is simply the state of the node
    # and value are actual attributes that node object

    graph = {'A': Node('A', None, ['B', 'C', 'E'], None),
             'B': Node('B', None, ['A','D', 'E'], None),
             'C': Node('C', None, ['A', 'F', 'G'], None),
             'D': Node('D', None, ['B', 'E'], None),
             'E': Node('E', None, ['A', 'B','D'], None),
             'F': Node('F', None, ['C'], None),
             'G': Node('G', None, ['C'], None)}

    frontier = [initialState]
    explored=[]
while len(frontier)!=0:
    currentNode = frontier.pop(0)
    explored.append(currentNode)
    for child in graph[currentNode].actions:
        if child not in frontier and child not in explored:
            graph[child].parent=currentNode
            if graph[child].state==goalState:
                return actionSequence(graph, initialState, goalState)
            frontier.append(child)
```

Now the function definition is complete which can be called as follows,

```
solution = BFS()
print(solution)
```

There is one additional line of graph[child].parent=currentNode in python code which is missing in pseudocode. This allows us to update each parent of a node as we traverse the graph. Afterwards, the function actionSequence() is called which returns a series of actions when a goal state is reached. Given a start state, end state and a graph, this function recursively iterated through each parent until the starting state is reached.

```
def actionSequence(graph, initialState, goalState):
    # returns a list of states starting from goal state moving upwards
    # parents until root node is reached
    solution=[goalState]
    currentParent=graph[goalState].parent
    while currentParent!=None:
        solution.append(currentParent)
        currentParent = graph[currentParent].parent
    solution.reverse()
    return solution
```

Calling BFS() will return the following solution, ['A', 'C', 'F']

1. Change initial state to D and set goal state as C. What will be resulting path of BFS search?

Solution:

['D', 'B', 'A', 'C']

3) Graded Lab Tasks

Lab Task 1:

Imagine going from Arad to Bucharest in the following map. Implement a BFS to find the corresponding path.

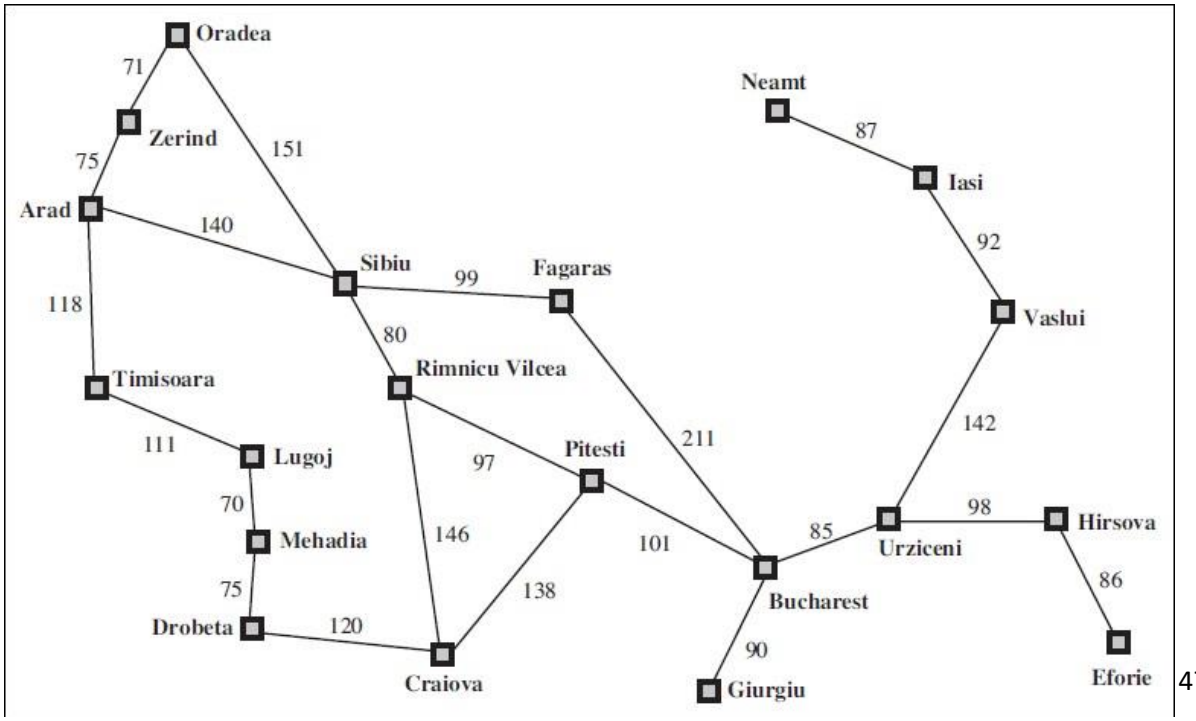


Figure 9 - Map of Romania

Lab Task 2:

Consider a maze as shown below. Each empty tile represents a separate node in the graph. There are maximum of four possible actions i.e., to move up, down, left or right on any given tile/node. Using BFS, find out how to get out of the maze if you're in the start position depicted below.

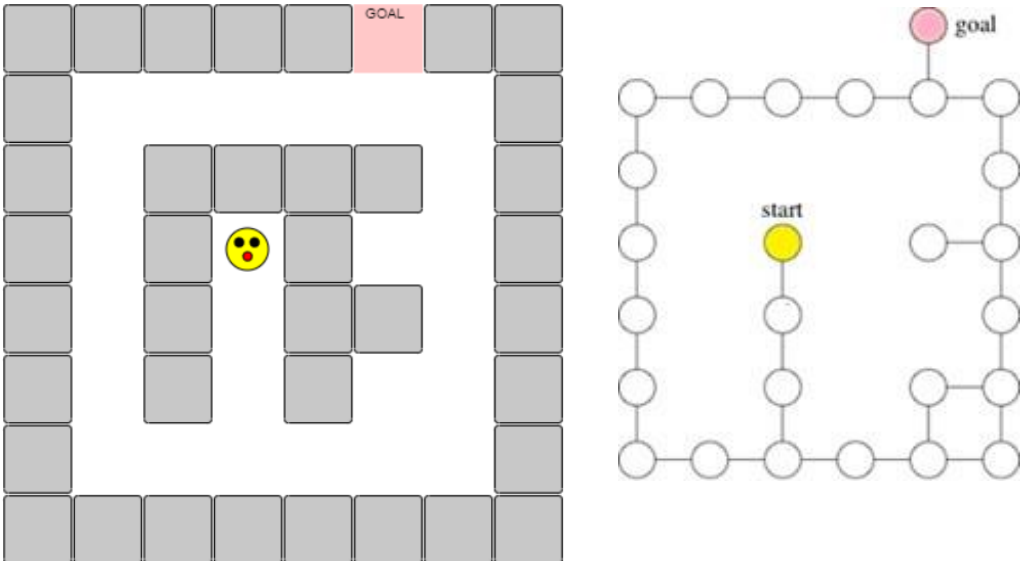


Figure 10 - Puzzle

Lab 04
Depth First Search

Objective:

This lab will introduce students to search problems. We will first start by representing problems in terms of state space graph. Given a state space graph, starting state and a goal state, students will then perform a basic **Depth First Search** solution within that graph. The output will be a set of actions or a path that will begin from initial state/node and end in the goal node.

Activity Outcomes:

This lab teaches you the following topics:

- How to represent problems in terms of state space graph
- How to find a solution of those problems using Depth First Search.

Instructor Note:

As pre-lab activity, read Chapter 3 from the book (Artificial Intelligence, A Modern Approach by Peter Norvig, 3rd edition) to know the basics of search algorithms.

1) Useful Concepts:

In the previous lab we studied breadth first search, which is the most naïve way of searching trees. The path returned by breadth search is not optimal. Uniform cost search always returns the optimal path. In this lab students will be able to implement a uniform cost search approach. Students will also be able to apply depth first search to their problems

Depth First Search (DFS)

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows: Pick a starting node and push all its adjacent nodes into a stack. Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack. Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

Pseudocode

```
DFS-iterative (G, s):                          //Where G is graph and s is source vertex
  let S be stack
  S.push( s )          //Inserting s in stack
  mark s as visited.
  while ( S is not empty):
     //Pop a vertex from stack to visit next
     v  =  S.top( )
    S.pop( )
    //Push all the neighbours of v in stack that are not visited
   for all neighbours w of v in Graph G:
      if w is not visited :
              S.push( w )
             mark w as visited


DFS-recursive(G, s):
   mark s as visited
   for all neighbours w of s in Graph G:
      if w is not visited:
         DFS-recursive(G, w)
```

The following image shows how DFS works.

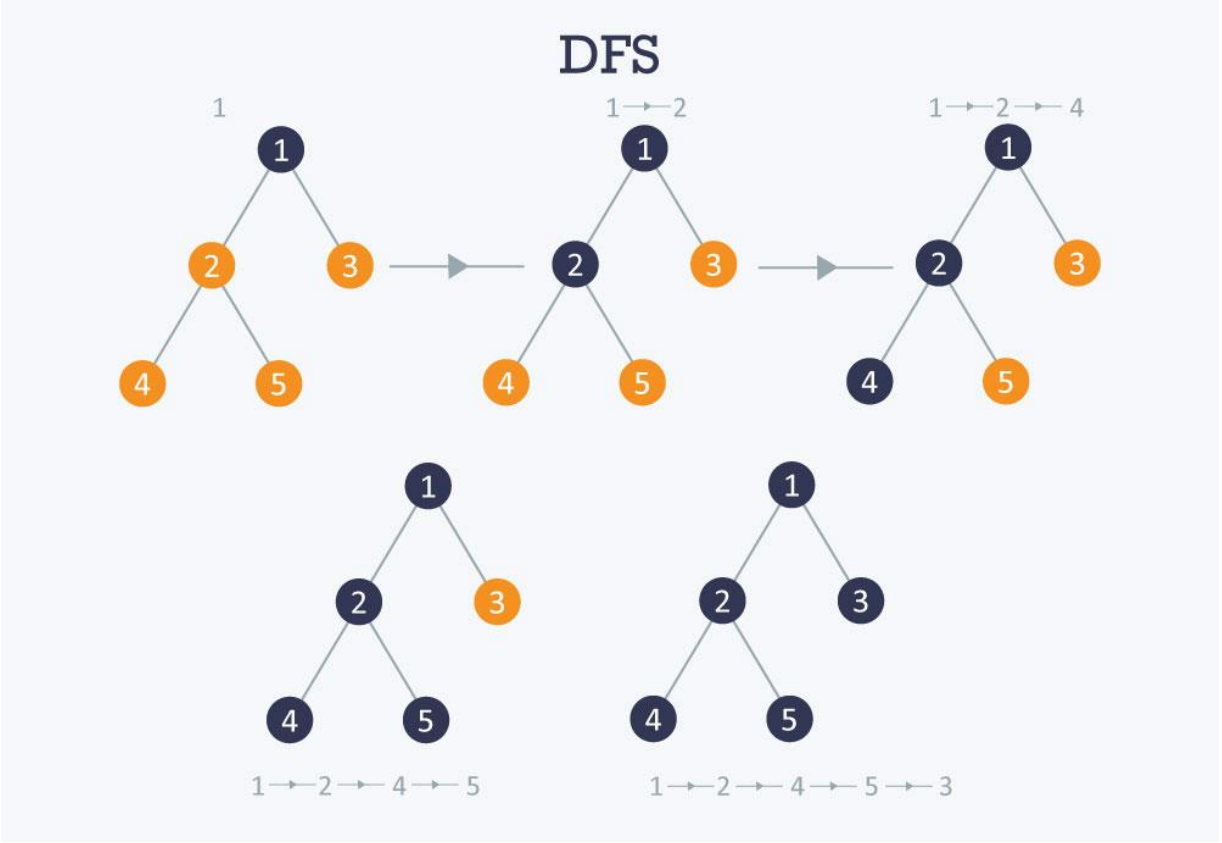


Figure 11 - Traversal

Time complexity O(V+E), when implemented using an adjacency list.

Applications

How to find connected components using DFS?

A graph is said to be disconnected if it is not connected, i.e. if two nodes exist in the graph such that there is no edge in between those nodes. In an undirected graph, a connected component is a set of vertices in a graph that are linked to each other by paths.

Consider the example given in the diagram. Graph G is a disconnected graph and has the following 3 connected components.

- First connected component is 1 -> 2 -> 3 as they are linked to each other
- Second connected component 4 -> 5
- Third connected component is vertex 6

In DFS, if we start from a start node it will mark all the nodes connected to the start node as visited. Therefore, if we choose any node in a connected component and run DFS on that node it will mark the whole connected component as visited.

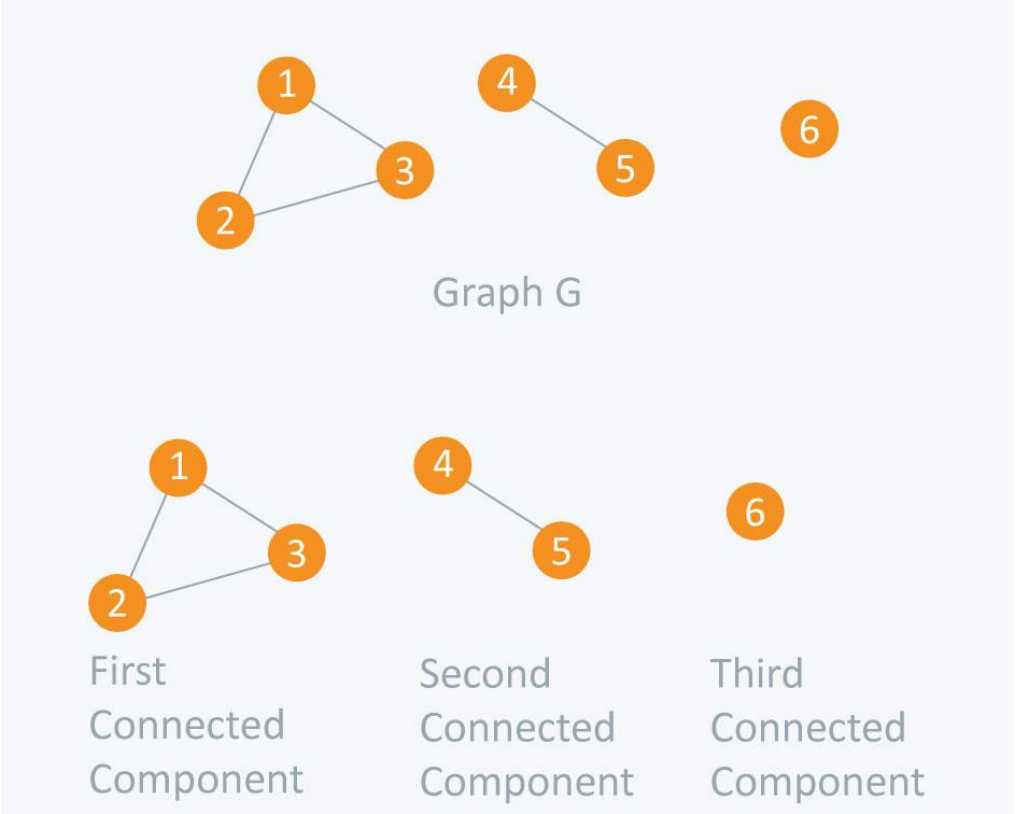


Figure 12 - Trvaersal

Input **File**

6
4
1 2
2 3
1 3
4 5

Code

```
#include <iostream>
#include <vector>
 using namespace std;

 vector <int> adj[10];
 bool visited[10];

 void dfs(int s) {
    visited[s] = true;
    for(int i = 0;i < adj[s].size();++i)    {
     if(visited[adj[s][i]] == false)
         dfs(adj[s][i]);
    }
 }

 void initialize() {
```

```
    for(int i = 0;i < 10;++i)
     visited[i] = false;
}

int main() {
    int nodes, edges, x, y, connectedComponents = 0;
    cin >> nodes;                    //Number of nodes
    cin >> edges;                    //Number of edges
    for(int i = 0;i < edges;++i) {
     cin >> x >> y;
 //Undirected Graph
     adj[x].push_back(y);                //Edge from vertex x to vertex y
     adj[y].push_back(x);                //Edge from vertex y to vertex x
    }

    initialize();                    //Initialize all nodes as not visited

    for(int i = 1;i <= nodes;++i) {
     if(visited[i] == false)     {
        dfs(i);
        connectedComponents++;
     }
    }
    cout << "Number of connected components: " << connectedComponents << endl;
    return 0;
}
```

Output

Number of connected components: 3

2) Solved Lab Activities:

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*45*	*High*	*CLO-6*

Activity 1:

Consider a toy problem that can be represented as a following graph. How would you represent this graph in python?

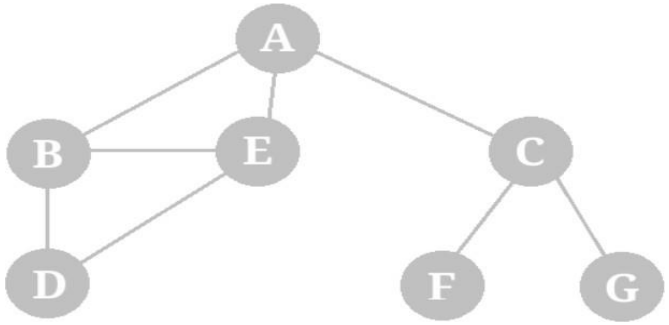


Figure 13 - Graph

Solution:

Remember a node of a tree can be represented by four attributes. We consider node as a class having four attributes namely,

1. State of the node
2. Parent of the node
3. Actions applicable to that node
4. The total path cost of that node starting from the initial state until that particular node is reached

```
class Node:
    #state = state          # class variable shared by all instances
    def __init__(self, state, parent, actions, totalCost):
        self.state = state     # instance variable unique to each insta
        self.parent = parent
        self.actions = actions # we are not saving actions themselves,
                               # only output states of those actions
        self.totalCost = totalCost
```

We can now implement this class in a dictionary. This dictionary will represent our state space graph. As we will traverse through the graph, we will keep updating parent and cost of each node.

```
# we think of a graph as a dictionary, items comprise of nodes, where
# each node has a key and a value. Key is simply the state of the node
# and value are actual attributes that node object

graph = {'A': Node('A', None, ['B', 'E', 'C'], None),
         'B': Node('B', None, ['D', 'E', 'A'], None),
         'C': Node('C', None, ['A', 'F', 'G'], None),
         'D': Node('D', None, ['B', 'E'], None),
         'E': Node('E', None, ['A', 'B','D'], None),
         'F': Node('F', None, ['C'], None),
         'G': Node('G', None, ['C'], None)}
```

For the graph in previous activity, imagine node A as starting node and your goal is to reach F. Keeping depth first search in mind, describe a sequence of actions that you must take to reach that goal state.

Remember that we can implement depth first search simply by using LIFO approach instead of FIFO that was used in breadth first search. Additionally we also don't keep leaf nodes (nodes without children) in explored set.

```
def DFS():
    initialState = 'A'
    goalState = 'D'

    # we think of a graph as a dictionary, items comprise of nodes, where
    # each node has a key and a value. Key is simply the state of the node
    # and value are actual attributes that node object

    graph = {'A': Node('A', None, ['B', 'E', 'C'], None),
             'B': Node('B', None, ['D', 'E', 'A'], None),
             'C': Node('C', None, ['A', 'F', 'G'], None),
             'D': Node('D', None, ['B', 'E'], None),
             'E': Node('E', None, ['A', 'B','D'], None),
             'F': Node('F', None, ['C'], None),
             'G': Node('G', None, ['C'], None)}

    frontier = [initialState]
    explored=[]
    while len(frontier)!=0:
        currentNode = frontier.pop(len(frontier)-1)
        print(currentNode)
        explored.append(currentNode)
        currentChildren=0
        for child in graph[currentNode].actions:
            if child not in frontier and child not in explored:
                graph[child].parent=currentNode
                if graph[child].state==goalState:
                    print(explored)
                    return actionSequence(graph, initialState, goalState)
                currentChildren=currentChildren+1
                frontier.append(child)
        if currentChildren==0:
            del explored[len(explored)-1]
```

Now the function definition is complete which can be called as follows,

```
solution = DFS()
print(solution)
```

Notice the difference in two portions of the code between breadth first search and depth first search. In the first we just pop out the last entry from the queue and in the 2nd difference we delete leaf nodes from the graph.

```
def actionSequence(graph, initialState, goalState):
    # returns a list of states starting from goal state moving upwards
    # parents until root node is reached
    solution=[goalState]
    currentParent=graph[goalState].parent
    while currentParent!=None:
        solution.append(currentParent)
        currentParent = graph[currentParent].parent
    solution.reverse()
    return solution
```

Calling DFS() will return the following solution, ['A', 'E', 'D']

Activity 2:

Change initial state to D and set goal state as C. What will be resulting path of BFS search? What will be the sequence of nodes explored?

Solution:

Final path is ['D', 'B', 'A', 'C'] Explored node sequence is D, E, A.

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1:

Imagine going from Arad to Bucharest in the following map. Your goal is to minimize the distance mentioned in the map during your travel. Implement a depth first search to find the corresponding path.

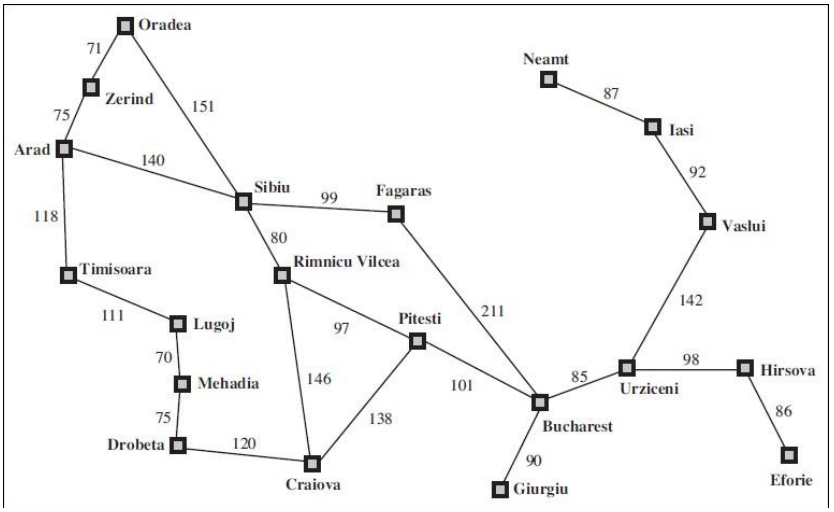


Figure 14 - Map of Romania

Lab Task 2:

Generate a list of possible words from a character matrix

Given an M × N boggle board, find a list of all possible words that can be formed by a sequence of adjacent characters on the board.

We are allowed to search a word in all eight possible directions, i.e., North, West, South, East, North-East, North-West, South-East, South-West, but a word should not have multiple instances of the same cell.

Consider the following the traditional 4 × 4 boggle board. If the input dictionary is [START, NOTE, SAND, STONED], the valid words are [NOTE, SAND, STONED].

Boggle Board

M	S	E	F
R	A	T	D
L	O	N	E
K	A	F	B

Figure 15 - 4x4 Boggle Board

Lab 05
Iterative Deepening Search

Objective:

This lab will introduce students to search problems. We will first start by representing problems in terms of state space graph. Given a state space graph, starting state and a goal state, students will then perform a basic **iterative deepening** solution within that graph. The output will be a set of actions or a path that will begin from initial state/node and end in the goal node.

Activity Outcomes:

This lab teaches you the following topics:

- How to represent problems in terms of state space graph
- How to find a solution of those problems using iterative deepening Search.

Instructor Note:

As pre-lab activity, read Chapter 3 from the book (Artificial Intelligence, A Modern Approach by Peter Norvig, 3rd edition) to know the basics of search algorithms.

1) Useful concepts:

The Iterative Deepening Depth-First Search (also ID-DFS) algorithm is an algorithm used to find a node in a tree. The basic principle of the algorithm is to start with a start node, and then look at the first child of this node. It then looks at the first child of that node (grandchild of the start node) and so on, until a node has no more children (we've reached a leaf node). It then goes up one level, and looks at the next child. If there are no more children, it goes up one more level, and so on, until it find more children or reaches the start node. If hasn't found the goal node after returning from the last child of the start node, the goal node cannot be found, since by then all nodes have been traversed.
So far this has been describing Depth-First Search (DFS). Iterative deepening adds to this, that the algorithm not only returns one layer up the tree when the node has no more children to visit, but also when a previously specified maximum depth has been reached. Also, if we return to the start node, we increase the maximum depth and start the search all over, until we've visited all leaf nodes (bottom nodes) and increasing the maximum depth won't lead to us visiting more nodes.

Specifically, these are the steps:
1. For each child of the current node
2. If it is the target node, return
3. If the current maximum depth is reached, return
4. Set the current node to this node and go back to 1.
5. After having gone through all children, go to the next child of the parent (the next sibling)
6. After having gone through all children of the start node, increase the maximum depth and go back to 1.
7. If we have reached all leaf (bottom) nodes, the goal node doesn't exist.

Example of the Algorithm

Consider the following tree:

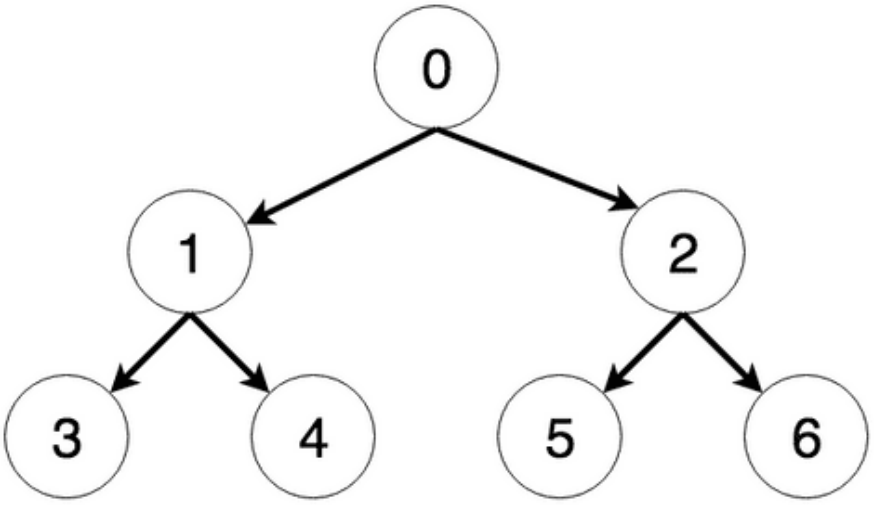


Figure 16 - Graph

The steps the algorithm performs on this tree if given node 0 as a starting point, in order, are:

1. Visiting Node 0
2. Visiting Node 1
3. Current maximum depth reached, returning…

4. Visiting Node 2
5. Current maximum depth reached, returning…
6. Increasing depth to 2
7. Visiting Node 0
8. Visiting Node 1
9. Visiting Node 3
10. Current maximum depth reached, returning…
11. Visiting Node 4
12. Current maximum depth reached, returning…
13. Visiting Node 2
14. Visiting Node 5
15. Current maximum depth reached, returning…
16. Visiting Node 6
17. Found the node we're looking for, returning…

Runtime of the Algorithm

If we double the maximum depth each time we need to go deeper, the runtime complexity of Iterative Deepening Depth-First Search (ID-DFS) is the same as regular Depth-First Search (DFS), since all previous depths added up will have the same runtime as the current depth ($1/2 + 1/4 + 1/8 + \ldots < 1$). The runtime of regular Depth-First Search (DFS) is $O(|N|)$ ($|N|$ = number of Nodes in the tree), since every node is traversed at most once. The number of nodes is equal to b^d, where b is the branching factor and d is the depth, so the runtime can be rewritten as $O(b^d)$.

Space of the Algorithm

The space complexity of Iterative Deepening Depth-First Search (ID-DFS) is the same as regular Depth-First Search (DFS), which is, if we exclude the tree itself, $O(d)$, with d being the depth, which is also the size of the call stack at maximum depth. If we include the tree, the space complexity is the same as the runtime complexity, as each node needs to be saved.

2) Solved Lab Activties

Sr.No	***Allocated Time***	***Level of Complexity***	***CLO Mapping***
1	***45***	***High***	***CLO-6***

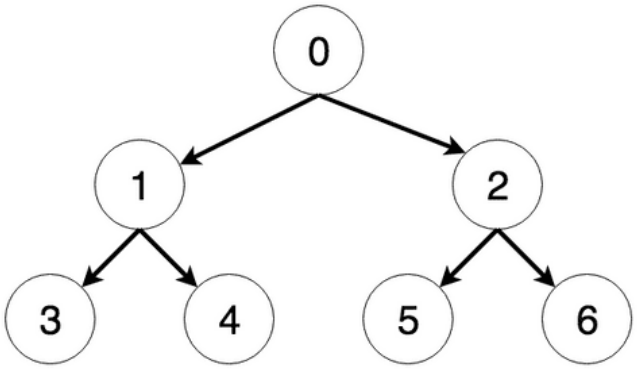


Figure 17 - Graph

```python
def iterative_deepening_dfs(start, target):
    """
    Implementation of iterative deepening DFS (depth-first search) algorithm to find the shortest path from a start to a target node..
    Given a start node, this returns the node in the tree below the start node with the target value (or null if it doesn't exist)
    Runs in O(n), where n is the number of nodes in the tree, or O(b^d), where b is the branching factor and d is the depth.
    :param start:  the node to start the search from
    :param target: the value to search for
    :return: The node containing the target value or null if it doesn't exist.
    """
    # Start by doing DFS with a depth of 1, keep doubling depth until we reach the "bottom" of the tree or find the node we're searching for
    depth = 1
    bottom_reached = False  # Variable to keep track if we have reached the bottom of the tree
    while not bottom_reached:
        # One of the "end nodes" of the search with this depth has to still have children and set this to False again
        result, bottom_reached = iterative_deepening_dfs_rec(start, target, 0, depth)
        if result is not None:
            # We've found the goal node while doing DFS with this max depth
            return result

        # We haven't found the goal node, but there are still deeper nodes to search through
        depth *= 2
        print("Increasing depth to " + str(depth))

    # Bottom reached is True.
    # We haven't found the node and there were no more nodes that still have children to explore at a higher depth.
    return None
```

```python
def iterative_deepening_dfs_rec(node, target, current_depth, max_depth):
    print("Visiting Node " + str(node["value"]))

    if node["value"] == target:
        # We have found the goal node we we're searching for
        print("Found the node we're looking for!")
        return node, True

    if current_depth == max_depth:
        print("Current maximum depth reached, returning...")
        # We have reached the end for this depth...
        if len(node["children"]) > 0:
            # ...but we have not yet reached the bottom of the tree
            return None, False
        else:
            return None, True

    # Recurse with all children
    bottom_reached = True
    for i in range(len(node["children"])):
        result, bottom_reached_rec = iterative_deepening_dfs_rec(node["children"][i], target, current_depth + 1,
                                                                 max_depth)
        if result is not None:
            # We've found the goal node while going down that child
            return result, True
        bottom_reached = bottom_reached and bottom_reached_rec

    # We've gone through all children and not found the goal node
    return None, bottom_reached
```

```
start = {
    "value": 0, "children": [
        {"value": 1, "children": [
            {"value": 3, "children": []},
            {"value": 4, "children": []}
        ]}, {
            "value": 2, "children": [
                {"value": 5, "children": []},
                {"value": 6, "children": []}
            ]
        }
    ]
}


print(iterative(start, 6)["value"])
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1:

Imagine going from Arad to Bucharest in the following map. Your goal is to minimize the distance mentioned in the map during your travel. Implement a iterative deepening search to find the corresponding path.

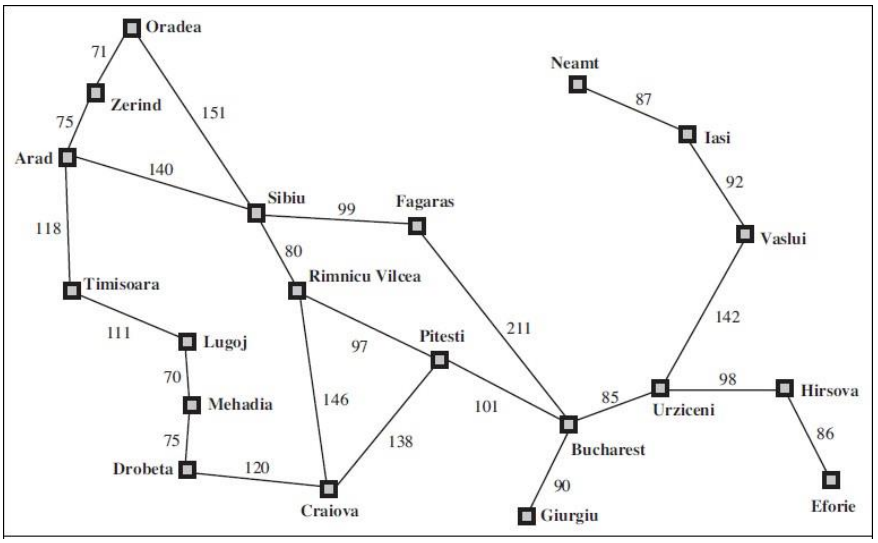


Figure 18 - Map of Romania

Lab Task 2:

Generate a list of possible words from a character matrix

Given a 8 × 8 boggle board, find a list of all possible words that can be formed by a sequence of adjacent characters on the board.

We are allowed to search a word in all eight possible directions, i.e., North, West, South, East, North-East, North-West, South-East, South-West, but a word should not have multiple instances of the same cell.

Consider the following the traditional 4 × 4 boggle board. If the input dictionary is [START, NOTE, SAND, STONED], the valid

M	S	E	F
R	A	T	D
L	O	N	E
K	A	F	B

Figure 19 - 4x4 Boggle Board

words are [NOTE, SAND, STONED]. ***With iterative deepening, create words of length 5, 6, 7 and 8 through each iteration.***

Lab 06
Uniform Cost Search

Objective:

This lab will introduce students to search problems. We will first start by representing problems in terms of state space graph. Given a state space graph, starting state and a goal state, students will then perform a basic **Uniform Cost Search** solution within that graph. The output will be a set of actions or a path that will begin from initial state/node and end in the goal node.

Activity Outcomes:

This lab teaches you the following topics:

- How to represent problems in terms of state space graph
- How to find a solution of those problems using Uniform Cost Search.

Instructor Note:

As pre-lab activity, read Chapter 3 from the book (Artificial Intelligence, A Modern Approach by Peter Norvig, 3rd edition) to know the basics of search algorithms.

1) Useful Concepts:

Uniform Cost Search is the best algorithm for a search problem, which does not involve the use of heuristics. It can solve any general graph for optimal cost. Uniform Cost Search as it sounds searches in branches, which are more or less the same in cost.
Uniform Cost Search again demands the use of a priority queue. Recall that Depth First Search used a priority queue with the depth up to a particular node being the priority and the path from the root to the node being the element stored. The priority queue used here is similar with the priority being the cumulative cost up to the node. Unlike Depth First Search where the maximum depth had the maximum priority, Uniform Cost Search gives the minimum cumulative cost the maximum priority.

Uniform Cost Search is an algorithm used to move around a directed weighted search space to go from a start node to one of the ending nodes with a minimum cumulative cost. This search is an uninformed search algorithm since it operates in a brute-force manner, i.e. it does not take the state of the node or search space into consideration. It is used to find the path with the lowest cumulative cost in a weighted graph where nodes are expanded according to their cost of traversal from the root node. This is implemented using a priority queue where lower the cost higher is its priority.

Algorithm of Uniform Cost Search
Below is the algorithm to implement Uniform Cost Search in Artificial Intelligence:-centre;”>**Algorithm for USC**

- Insert RootNode into the queue.
- Repeat till queue is not empty:
- Remove the next element with the highest priority from the queue.
- If the node is a destination node, then print the cost and the path and exit

else insert all the children of removed elements into the queue with their cumulative cost as their priorities.
Here rootNode is the starting node for the path, and a priority queue is being maintained to maintain the path with the least cost to be chosen for the next traversal. In case 2 paths have the same cost of traversal, nodes are considered alphabetically.

Example of Uniform Cost Search
Consider the below example, where we need to reach any one of the destination node{G1, G2, G3} starting from node S. Node{A, B, C, D, E and F} are the intermediate nodes. Our motive is to find the path from S to any of the destination state with the least cumulative cost. Each directed edge represents the direction of movement allowed through that path, and its labelling represents the cost is one travels through that path. Thus Overall cost of the path is a sum of all the paths.
For e.g. – a path from S to G1- {S->A -> G1} whose cost is SA +AG1 = 5 + 9 = 14

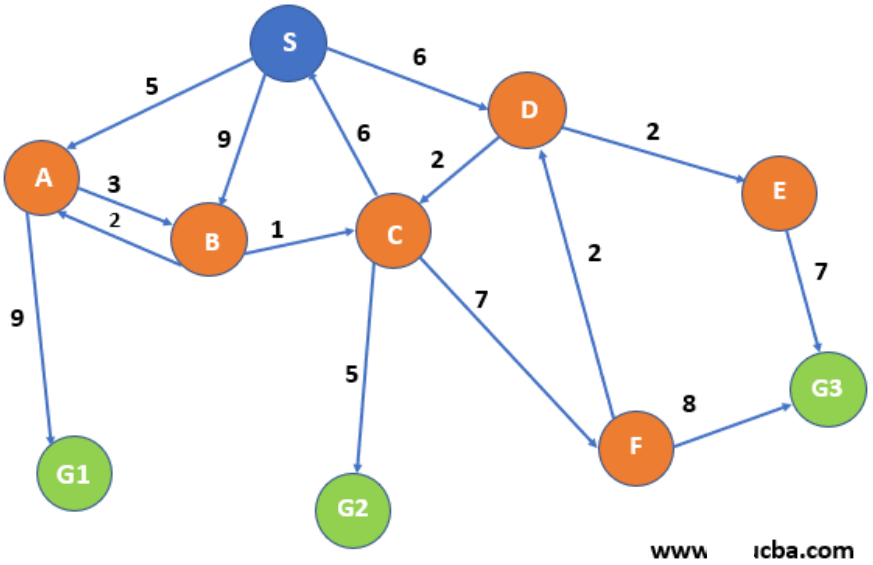
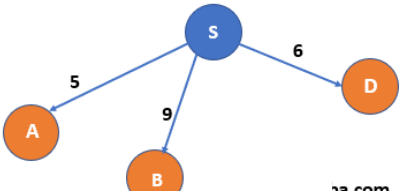
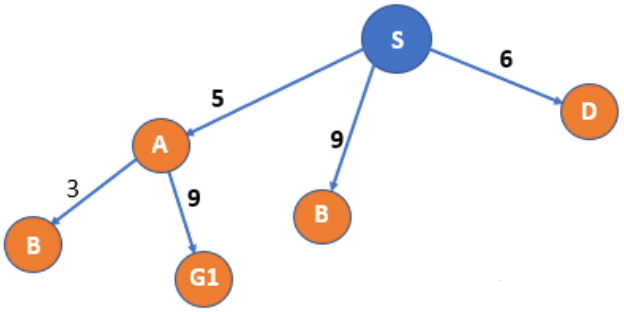
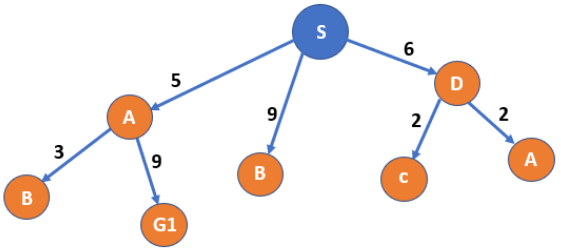
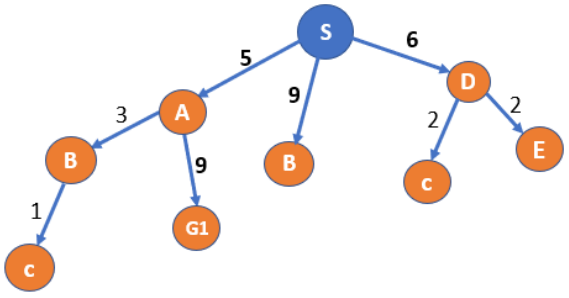
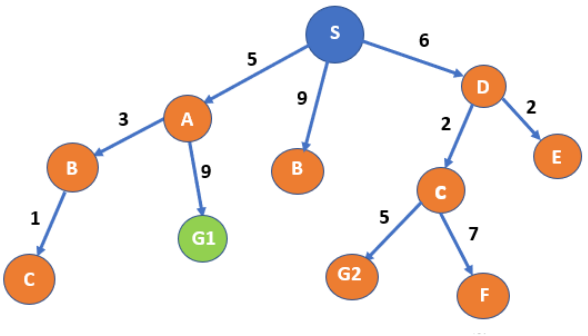
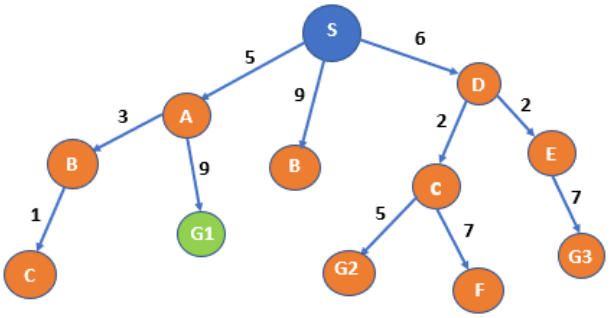
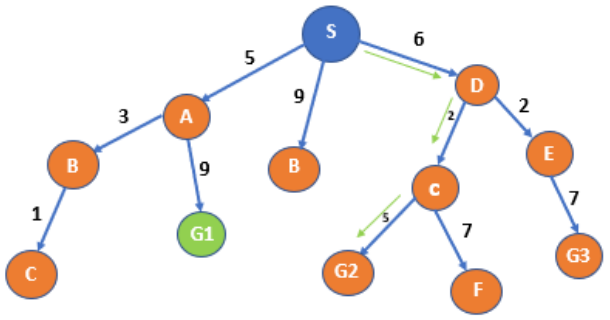


Figure 20 - Gra

Here we will maintain a priority queue the same as BFS with the cost of the path as its priority, lower the cost higher is the priority.

Explanation	Flow	Visited List
Step 1- We will start with start node and check if we have reached any of the destination nodes, i.e. No thus continue.		
Step 2– We reach all the nodes that can be reached from S I .eA, B, D. And Since node S has been visited thus added to the visited List. Now we select the cheapest path first for further expansion, i.e. A		S
Step 3 – Node B and G1 can be reached from A and since node A is visited thus move to the visited list. Since G1 is reached but for the optimal solution, we need to consider every possible case; thus, we will expand the next cheapest path, i.e. S->D.		S, A

Step 4– Now node D has been visited thus it goes to visited list and now since we have three paths with the same cost, we will choose alphabetically thus will expand node B		S,A,D
Step 5-: From B, we can only reach node C. Now the path with minimum weight is S->D->C, i.e. 8. Thus expand C. And B has now visited node.		S,A,D,B
Step 6:- From C we can reach G2 and F node with 5 and 7 weights respectively. Since S is present in the visited list thus, we are not considering the C->S path. Now C will enter the visited list. Now the next node with the minimum total path is S->D->E, i.e. 8. Thus we will expand E.		D,B,C
Step 7:- From E we can reach only G3. E will move to the visited list.		S,A,D,B,C,E

Step 8 – In the last, we have 6 active paths . S->B – B is in the visited list; thus will be marked as a dead end. b.Same for S->A->B->C – C has already been visited thus is considered a dead end. Out of the remaining S->A->G1 S->D->C->G2 S->D->C->F S->D->E->G3 Minimum is S->D->C->G2 And also, G2 is one of the destination nodes. Thus, we found our path.		S,A,D,B,C,E

2) Solved Lab Activities:

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*45*	*High*	*CLO-6*

Activity 1

Imagine the same tree but this time we also mention the cost of each edge.

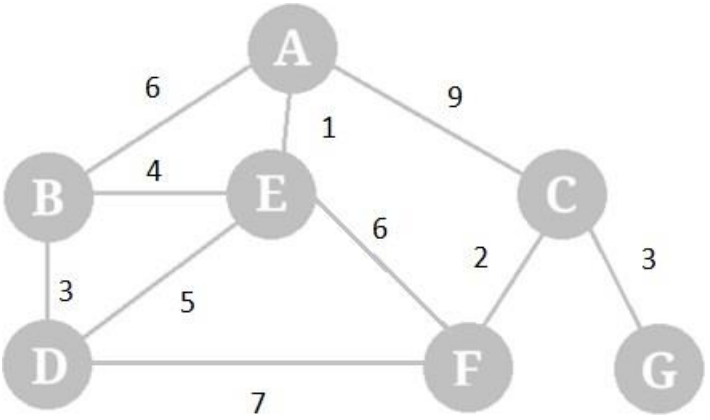


Figure 21 - Graph

Implement a uniform cost solution to find the path from C to B.

Solution:

First we modify our graph structure so that in Node.actions is an array of tuples where each tuple contains a vertex and its associated weight.

```
graph = {'A': Node('A', None, [('B',6), ('C',9), ('E',1)], 0),
         'B': Node('B', None, [('A',6), ('D',3), ('E',4)], 0),
         'C': Node('C', None, [('A',9), ('F',2), ('G',3)], 0),
         'D': Node('D', None, [('B',3), ('E',5), ('F',7)], 0),
         'E': Node('E', None, [('A',1), ('B',4), ('D',5), ('F',6)], 0),
         'F': Node('F', None, [('C',2), ('E',6), ('D',7)], 0),
         'G': Node('G', None, [('C',3)], 0)}
```

We also modify the frontier format which will now be a dictionary. This dictionary will contain each node (the state of the node will act as a key and its parent and accumulated cost from the initial state will be two attributes of a particular key). We now define a function which will give the node/key for which the cost is minimum. This will be implementation of pop method from the priority queue.

```
import math

def findMin(frontier):
    # returns that node in the frontier which has a lowest cost
    minV=math.inf
    node=''
    for i in frontier:
        if minV>frontier[i][1]:
            minV=frontier[i][1]
            node = i
    return node
```

The rest of the functions are the same as in BFS i.e.,

```
def actionSequence(graph, initialState, goalState):
    # returns a list of states starting from goal state moving upwards towards
    # parents until root node is reached
    solution=[goalState]
    currentParent=graph[goalState].parent
    while currentParent!=None:
        solution.append(currentParent)
        currentParent = graph[currentParent].parent
    solution.reverse()
    return solution

class Node:
    #state = state          # class variable shared by all instances
    def __init__(self, state, parent, actions, totalCost):
        self.state = state    # instance variable unique to each instance
        self.parent = parent
        self.actions = actions # we are not saving actions themselves,
                               # only output states of those actions
        self.totalCost = totalCost
```

Finally we define UCS()

```
def UCS():
    initialState = 'C'
    goalState = 'B'

    # we think of a graph as a dictionary, items comprise of nodes, wher
    # each node has a key and a value. Key is simply the state of the no
    # and value are actual attributes that node object
##
    graph = {'A': Node('A', None, [('B',6), ('C',9), ('E',1)], 0),
             'B': Node('B', None, [('A',6), ('D',3), ('E',4)], 0),
             'C': Node('C', None, [('A',9), ('F',2), ('G',3)], 0),
             'D': Node('D', None, [('B',3), ('E',5), ('F',7)], 0),
             'E': Node('E', None, [('A',1), ('B',4), ('D',5), ('F',6)],
             'F': Node('F', None, [('C',2), ('E',6), ('D',7)], 0),
             'G': Node('G', None, [('C',3)], 0)}

     frontier = dict()
     frontier[initialState]=(None, 0)
     explored=[] # parent of initial node is None and its cost is 0
```

```
while len(frontier)!=0:
    currentNode =findMin(frontier)
    del frontier[currentNode]
    if graph[currentNode].state==goalState:
        return actionSequence(graph, initialState, goalState)
    explored.append(currentNode)
    for child in graph[currentNode].actions:
        currentCost=child[1] + graph[currentNode].totalCost
        if child[0] not in frontier and child[0] not in explored:
            graph[child[0]].parent=currentNode
            graph[child[0]].totalCost=currentCost
            frontier[child[0]]=(graph[child[0]].parent, graph[child[0]].totalCost)
        elif child[0] in frontier:
            if frontier[child[0]][1] < currentCost:
                graph[child[0]].parent=frontier[child[0]][0]
                graph[child[0]].totalCost=frontier[child[0]][1]
            else:
                frontier[child[0]]=(currentNode, currentCost)
                graph[child[0]].parent=frontier[child[0]][0]
                graph[child[0]].totalCost=frontier[child[0]][1]
```

solution = UCS() print(solution)

The above two lines will print ['C', 'F', 'D', 'B']

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Imagine going from Arad to Bucharest in the following map. Your goal is to minimize the distance mentioned in the map during your travel. Implement a uniform cost search to find the corresponding

path.

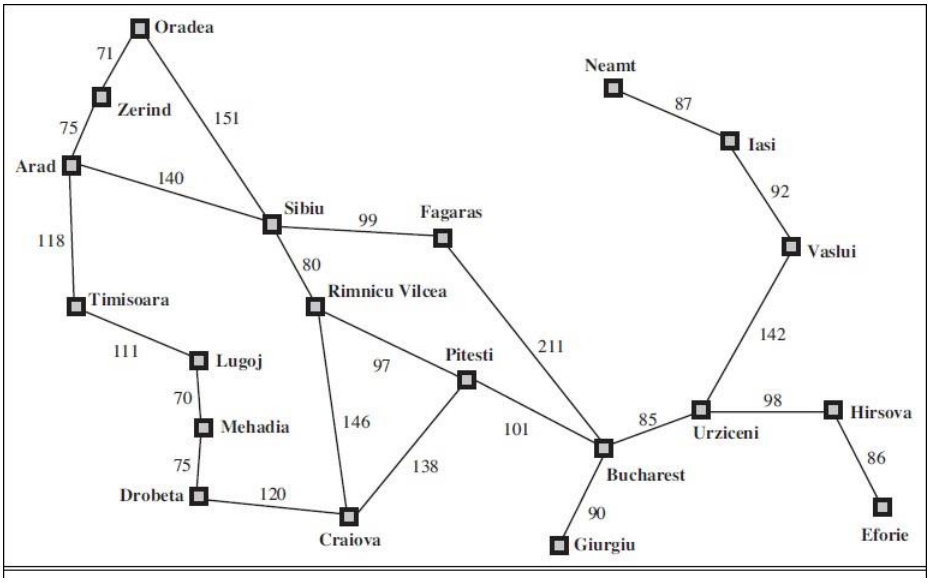


Figure 22 - Map of Romania

Lab 07
A-Star Search

Objective:

This lab will introduce students to heuristic and local search methods. In particular we will implement a particular variant of heuristic search known as A* search.

Activity Outcomes:

This lab teaches you the following topics:

- How to represent problems in terms of state space graph How to find a solution of those problems using A* search.

Instructor Note:

As pre-lab activity, read Chapter 3 from the book (Artificial Intelligence, A Modern Approach by Peter Norvig, 3rd edition) to know the basics of search algorithms.

1) Useful concepts:

So far we have studied uninformed search strategies. There can also be occasions where we are given some extra information related to a particular goal in the form of heuristics. We can use such heuristics in our search strategy. A particular form of this strategy known as A* search uses the total cost of a solution via a particular node as that node's evaluation criteria. We will see its implementation in detail.

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

Why A* Search Algorithm?

Informally speaking, A* Search algorithms, unlike other traversal techniques, it has "brains". What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections. And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

Explanation

Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* Search Algorithm comes to the rescue. What A* Search Algorithm does is that at each step it picks the node according to a value-'**f**' which is a parameter equal to the sum of two other parameters – '**g**' and '**h**'. At each step it picks the node/cell having the lowest '**f**', and process that node/cell. We define '**g**' and '**h**' as simply as possible below **g** = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there. **h** = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate this 'h' which are discussed in the later sections.

Algorithm

We create two lists – Open List and Closed List (just like Dijkstra Algorithm)

```
// A* Search Algorithm
1.  Initialize the open list
2.  Initialize the closed list
    put the starting node on the open
    list (you can leave its f at zero)

3.  while the open list is not empty
    a) find the node with the least f on
       the open list, call it "q"

    b) pop q off the open list

    c) generate q's 8 successors and set their
       parents to q
```

d) for each successor
i) if successor is the goal, stop search

ii) else, compute both **g** and **h** for successor
successor.**g** = q.**g** + distance between
successor and q
successor.**h** = distance from goal to
successor (This can be done using many
ways, we will discuss three heuristics-
Manhattan, Diagonal and Euclidean
Heuristics)

successor.**f** = successor.**g** + successor.**h**

iii) if a node with the same position as
successor is in the OPEN list which has a
lower **f** than successor, skip this successor

iV) if a node with the same position as
successor is in the CLOSED list which has
a lower **f** than successor, skip this successor
otherwise, add the node to the open list
end (for loop)

e) push q on the closed list
end (while loop)

So suppose as in the below figure if we want to reach the target cell from the source cell, then the A* Search algorithm would follow path as shown below. Note that the below figure is made by considering Euclidean Distance as a heuristics.

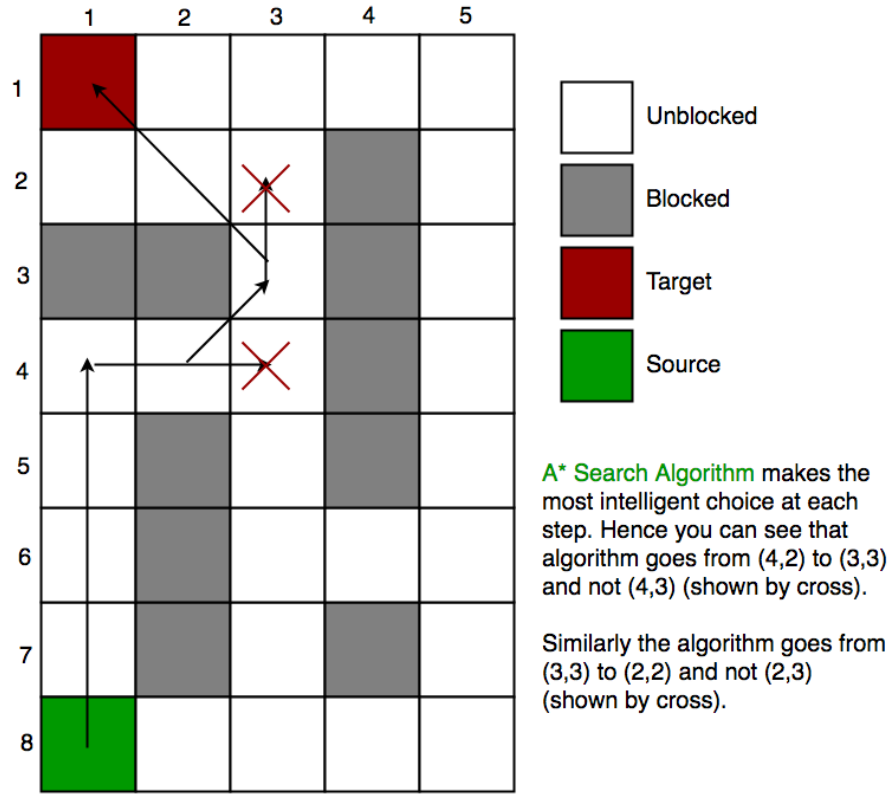


Figure 23 - Distance

Heuristics

There are generally three approximation heuristics to calculate h –

1) Manhattan Distance –

- It is nothing but the sum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

h = abs (current_cell.x – goal.x) +
abs (current_cell.y – goal.y)

- When to use this heuristic? – When we are allowed to move only in four directions only

(right, left, top, bottom)

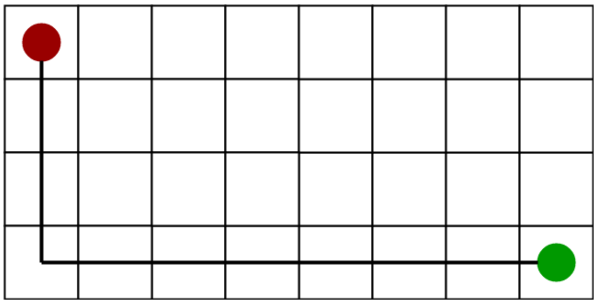


Figure 24 - Distance

The Manhattan Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).

2) Diagonal Distance-

- It is nothing but the maximum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

dx = abs(current_cell.x – goal.x)
dy = abs(current_cell.y – goal.y)

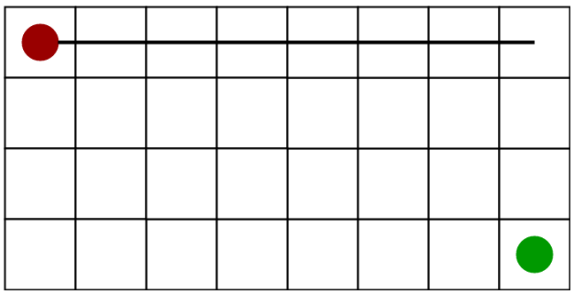


Figure 25 - Distance

h = D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)

where D is length of each node(usually = 1) and D2 is diagonal distance between each node (usually = sqrt(2)).

- When to use this heuristic? – When we are allowed to move in eight directions only (similar to a move of a King in Chess)

The Diagonal Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).

3) Euclidean Distance-

- As it is clear from its name, it is nothing but the distance between the current cell and the goal cell using the distance formula

h = sqrt ((current_cell.x – goal.x)2 + (current_cell.y – goal.y)2)

- When to use this heuristic? – When we are allowed to move in any directions.

The Euclidean Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).

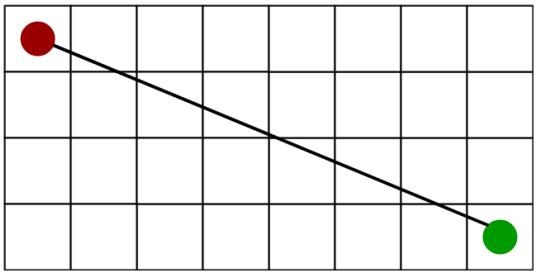


Figure 26 - Distance

Relation (Similarity and Differences) with other algorithms- Dijkstra is a special case of A* Search Algorithm, where h = 0 for all nodes.

2) Solved Lab Activities:

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*45*	*High*	*CLO-6*

Activity 1:

Consider a maze as shown below. Each empty tile represents a separate node in the graph, while the walls are represented by blue tiles. Your starting node is A and the goal is to reach Y. Implement an A* search to find the resulting path.

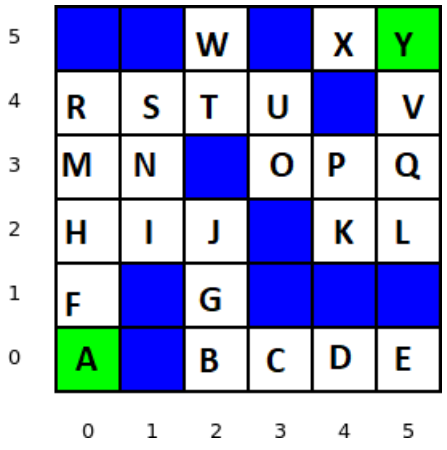


Figure 27 - Maze

Solution:

Since A* search needs a heuristic function that specifies the distance of each node with the goal, you can use Euclidean distance between a particular node and the goal node as your heuristic function of that particular node.

But first we need to modify the structure of Node class that will also include the heuristic distance of that node.

```
class Node:
    def __init__(self, state, parent, actions, totalCost, heuristic):
        self.state = state
        self.parent = parent
        self.actions = actions
        self.totalCost = totalCost
        self.heuristic = heuristic
```

Next we modify the structure of dictionary that will save our graph. Here at each node, we also specify the coordinate location of each node. For implementation we will follow uniform cost implementation except for the fact that the total cost will include both the previous cost of reaching that node and the distance of the goal with that node.

```
import math

class Node:
    def __init__(self, state, parent, actions, totalCost, heuristic):
        self.state = state
        self.parent = parent
        self.actions = actions
        self.totalCost = totalCost
        self.heuristic = heuristic

def findMin(frontier):
    minV = math.inf
    node = ''
    for i in frontier:
        if minV > frontier[i][1]:
            minV = frontier [i][1]
            node = i
    return node

def actionSequence(graph, initialState, goalState):
    solution = [goalState]
    currentParent = graph[goalState].parent
    while currentParent!= None:
        solution.append(currentParent)
        currentParent = graph[currentParent].parent
    solution.reverse()
    return solution
```

```
def Astar():
    initialState = 'A'
    goalState = 'Y'


    graph = {'A': Node('A', None, [('F',1)], (0,0), 0),
             'B': Node('B', None, [('G',1), ('C',1)], (2,0), 0),
             'C': Node('C', None, [('B',1), ('D',1)], (3,0), 0),
             'D': Node('D', None, [('C',1), ('E',1)], (4,0), 0),
             'E': Node('E', None, [('D',1)], (5,0), 0),
             'F': Node('F', None, [('A',1), ('H',1)], (0,1), 0),
             'G': Node('G', None, [('B',1), ('J',1)], (2,1), 0),
             'H': Node('H', None, [('F',1), ('I',1), ('M',1)], (0,2), 0),
             'I': Node('I', None, [('H',1), ('J',1), ('N',1)], (1,2), 0),
             'J': Node('J', None, [('G',1), ('I',1)], (2,2), 0),
             'K': Node('K', None, [('L',1), ('P',1)], (4,2), 0),
             'L': Node('L', None, [('K',1), ('Q',1)], (5,2), 0),
             'M': Node('M', None, [('H',1), ('N',1), ('R',1)], (0,3), 0),
             'N': Node('N', None, [('I',1), ('M',1), ('S',1)], (1,3), 0),
             'O': Node('O', None, [('P',1), ('U',1)], (3,3), 0),
             'P': Node('P', None, [('O',1), ('Q',1)], (4,3), 0),
             'Q': Node('Q', None, [('L',1), ('P',1), ('V',1)], (5,3), 0),
             'R': Node('R', None, [('M',1), ('S',1)], (0,4), 0),
             'S': Node('S', None, [('N',1), ('R',1), ('T',1)], (1,4), 0),
             'T': Node('T', None, [('S',1), ('U',1), ('W',1)], (2,4), 0),
             'U': Node('U', None, [('O',1), ('T',1)], (3,4), 0),
             'V': Node('V', None, [('Q',1), ('Y',1)], (5,4), 0),
             'W': Node('W', None, [('T',1)], (2,5), 0),
             'X': Node('X', None, [('Y',1)], (4,5), 0),
             'Y': Node('Y', None, [('V',1), ('X',1)], (5,5), 0)
             }
```

```
frontier = dict()
heuristicCost= math.sqrt(((graph[goalState].heuristic[0]-graph[initialState].heuristic[0])\
               **2)+((graph[goalState].heuristic[1]-graph[initialState].heuristic[1])**2))
frontier[initialState]=(None, heuristicCost)
explored=dict()
while len(frontier)!=0:
    currentNode =findMin(frontier)
    print(currentNode)
    del frontier[currentNode]
    if graph[currentNode].state==goalState:
        return actionSequence(graph, initialState, goalState)

    heuristicCost= math.sqrt(((graph[goalState].heuristic[0]-graph[currentNode].heuristic[0]
               **2)+((graph[goalState].heuristic[1]-graph[currentNode].heuristic[1])**2))
    currentCost=graph[currentNode].totalCost
    explored[currentNode]=(graph[currentNode].parent, heuristicCost+currentCost)
    for child in graph[currentNode].actions:
        currentCost=child[1] + graph[currentNode].totalCost
        heuristicCost=math.sqrt(((graph[goalState].heuristic[0]-graph[child[0]].heuristic[0]
               **2)+((graph[goalState].heuristic[1]-graph[child[0]].heuristic[1])**2))
        if child[0] in explored:
            if graph[child[0]].parent==currentNode or child[0]==initialState or \
                explored[child[0]][1] <= currentCost + heuristicCost:
                continue
        if child[0] not in frontier:
            graph[child[0]].parent=currentNode
            graph[child[0]].totalCost=currentCost
            frontier[child[0]]=(graph[child[0]].parent, currentCost + heuristicCost)
        else:
            if frontier[child[0]][1] < currentCost + heuristicCost:
                graph[child[0]].parent=frontier[child[0]][0]
                graph[child[0]].totalCost=frontier[child[0]][1] - heuristicCost
                  else:
                       frontier[child[0]]=(currentNode, currentCost + heuristicCost)
                       graph[child[0]].parent=frontier[child[0]][0]
                       graph[child[0]].totalCost=currentCost


solution = Astar()
print(solution)
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Your goal is to navigate a robot out of a maze. The robot starts in the corner of the maze marked with red color. You can turn the robot to face north, east, south, or west. You can direct the robot to move forward a certain distance, although it will stop before hitting a wall. The goal is to reach the final state marked with green color.

Write a program that implements A algorithms to solve this maze. Write the path followed (in the form of coordinates) and the cost of the path.*

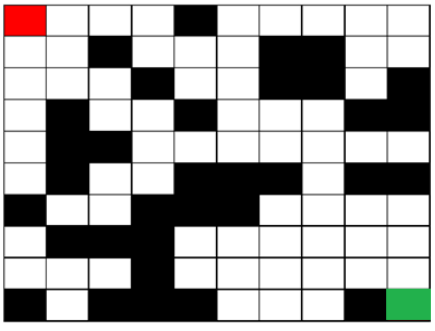


Figure 28 - Maze

Lab 08
Hill Climbing Search

Objective:

This lab will introduce students to search problems. We will first start by representing problems in terms of state space graph. Given a state space graph, starting state and a goal state, students will then perform a basic **hill climbing** solution within that graph. The output will be a set of actions or a path that will begin from initial state/node and end in the goal node. In computer science, hill climbing is a mathematical optimization technique which belongs to the family of local search. It is relatively simple to implement, making it a popular first choice. Although more advanced algorithms may give better results, in some situations hill climbing works just as well.

Activity Outcomes:

This lab teaches you the following topics:

- How to represent problems in terms of state space graph
- How to find a solution of those problems using hill climbing Search.

Instructor Note:

As pre-lab activity, read Chapter 3 from the book (Artificial Intelligence, A Modern Approach by Peter Norvig, 3rd edition) to know the basics of search algorithms.

1) Useful concepts:

Hill climbing can be used to solve problems that have many solutions, some of which are better than others. It starts with a random (potentially poor) solution, and iteratively makes small changes to the solution, each time improving it a little. When the algorithm cannot see any improvement anymore, it terminates. Ideally, at that point the current solution is close to optimal, but it is not guaranteed that hill climbing will ever come close to the optimal solution.
For example, hill climbing can be applied to the traveling salesman problem. It is easy to find a solution that visits all the cities but will be very poor compared to the optimal solution. The algorithm starts with such a solution and makes small improvements to it, such as switching the order in which two cities are visited. Eventually, a much better route is obtained. Hill climbing is used widely in artificial intelligence, for reaching a goal state from a starting node. Choice of next node and starting node can be varied to give a list of related algorithms.

Hill Climbing is a heuristic search used for mathematical optimization problems in the field of Artificial Intelligence. Given a large set of inputs and a good heuristic function, it tries to find a sufficiently good solution to the problem. This solution may not be the global optimal maximum.

- In the above definition, **mathematical optimization problems** imply that hill-climbing solves the problems where we need to maximize or minimize a given real function by choosing values from the given inputs. Example-Travelling salesman problem where we need to minimize the distance traveled by the salesman.
- 'Heuristic search' means that this search algorithm may not find the optimal solution to the problem. However, it will give a good solution in a **reasonable time.**
- A **heuristic function** is a function that will rank all the possible alternatives at any branching step in the search algorithm based on the available information. It helps the algorithm to select the best route out of possible routes.

Features of Hill Climbing

1. Variant of generate and test algorithm: It is a variant of generating and test algorithm. The generate and test algorithm is as follows :
1. Generate possible solutions.
2. Test to see if this is the expected solution.
3. If the solution has been found quit else go to step 1.
Hence we call Hill climbing a variant of generating and test algorithm as it takes the feedback from the test procedure. Then this feedback is utilized by the generator in deciding the next move in search space.

2. Uses the Greedy approach: At any point in state space, the search moves in that direction only which optimizes the cost of function with the hope of finding the optimal solution at the end.

Types of Hill Climbing

1. **Simple Hill climbing:** It examines the neighboring nodes one by one and selects the first neighboring node which optimizes the current cost as the next node.

Algorithm:

Step 1 : *Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make initial state as current state.*

Step 2 : *Loop until the solution state is found or there are no new operators present which can be applied to the current state.*

a) Select a state that has not been yet applied to the current state and apply it to produce a new state.

b) Perform these to evaluate new state

i. If the current state is a goal state, then stop and return success.

ii. If it is better than the current state, then make it current state and proceed further.

iii. If it is not better than the current state, then continue in the loop until a solution is found.

Step 3 : *Exit.*

2. Steepest-Ascent Hill climbing: It first examines all the neighboring nodes and then selects the node closest to the solution state as of the next node.

Algorithm:

Step 1 : *Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make initial state as current state.*

Step 2 : *Repeat these steps until a solution is found or current state does not change*

a) Select a state that has not been yet applied to the current state.

b) Initialize a new 'best state' equal to current state and apply it to produce a new state.

c) Perform these to evaluate new state

i. If the current state is a goal state, then stop and return success. ii. If it is better then best state, then make it best state else continue loop with another new state.

d) Make best state as current state and go to Step 2: b) part.

Step 3 : *Exit*

3. **Stochastic hill climbing:** It does not examine all the neighboring nodes before deciding which node to select. It just selects a neighboring node at random and decides (based on the amount of improvement in that neighbor) whether to move to that neighbor or to examine another.

Algorithm:

Step 1: *Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make* the *initial state* the *current state.*

Step 2: *Repeat these steps until a solution is found or* the *current state does not change.*

a) Select a state that has not been yet applied to the current state.

b) Apply successor function to the current state and generate all the neighbor *states.*

c) Among the generated neighbor *states which are better than* the *current state choose a state randomly (or based on some probability function).*

d) If the chosen *state is* the *goal state, then return success, else make it* the *current state and repeat step 2: b) part.*

Step 3: *Exit.*

2) Solved Lab Activities

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*45*	*High*	*CLO-6*

Activity 1:

For the given graph , imagine node A as starting node and your goal is to reach Y. Apply hill climbing and see how closer you can get to your destination.

Solution:

Instead of maintaining a fringe or a frontier to save the nodes that are to be explored, hill climbing just explores the best child of a given node, then explores the best grandchild of a particular child and so on and so forth.

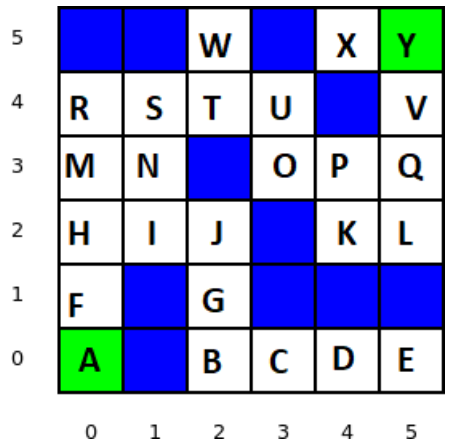


Figure 29 - Maze

```
import math

class Node:
    def __init__(self, state, parent, actions, totalCost, heuristic):
        self.state = state
        self.parent = parent
        self.actions = actions
        self.totalCost = totalCost
        self.heuristic = heuristic
def hillClimbing():

    graph = {
            'A' : Node('A', None, [('F',1)], 0, (0,0)),
            'B' : Node('B', None, [('C',1), ('G',1)], 0,(2,0)),
            'C' : Node('C', None, [('B',1), ('D',1)], 0,(3,0)),
            'D' : Node('D', None, [('C',1), ('E',1)], 0,(4,0)),
            'E' : Node('E', None, [('D',1)], 0,(5,0)),
            'F' : Node('F', None, [('A',1), ('H',1)], 0,(0,1)),
            'G' : Node('G', None, [('B',1), ('J', 1)], 0,(2,1)),
            'H' : Node('H', None, [('F',1), ('I',1), ('M',1)], 0,(0,2)),
            'I' : Node('I', None, [('H',1), ('J',1), ('N',1)], 0,(1,2)),
            'J' : Node('J', None, [('G',1), ('I',1)], 0,(2,2)),
            'K' : Node('K', None, [('L',1), ('P',1)], 0,(4,2)),
            'L' : Node('L', None, [('K',1), ('Q',1)], 0,(5,2)),
            'M' : Node('M', None, [('H',1), ('N',1), ('R',1)], 0,(0,3)),
            'N' : Node('N', None, [('I',1), ('M', 1), ('S', 1)], 0,(1,3)),
            'O' : Node('O', None, [('P',1), ('U',1)], 0,(3,3)),
            'P' : Node('P', None, [('K',1), ('O',1), ('Q',1)], 0,(4,3)),
            'Q' : Node('Q', None, [('L',1), ('P',1), ('V',1)], 0,(5,3)),
            'R' : Node('R', None, [('M',1), ('S',1)], 0,(0,4)),
            'S' : Node('S', None, [('N',1), ('R',1), ('T',1)], 0,(1,4)),
            'T' : Node('T', None, [('S',1), ('W',1), ('U',1)], 0,(2,4)),
            'U' : Node('U', None, [('O',1), ('T', 1)], 0,(3,4)),
            'V' : Node('V', None, [('Q',1), ('Y',1)], 0,(5,4)),
            'W' : Node('W', None, [('T',1)], 0,(2,5)),
            'X' : Node('X', None, [('Y',1)], 0,(4,5)),
            'Y' : Node('Y', None, [('X',1), ('Y',1)], 0,(5,5))}

    initialState = 'A'
    goalState = 'Y'
```

```
parentNode=initialState
parentCost = math.sqrt((graph[goalState].heuristic[0] - \
                        graph[initialState].heuristic[0])**2+\
                       (graph[goalState].heuristic[1] - \
                        graph[initialState].heuristic[1])**2)
explored=[]
solution=[]
minChildCost = parentCost - 1
while parentNode!=goalState:
    bestNode=parentNode
    minChildCost=parentCost
    explored.append(parentNode)
    for child in graph[parentNode].actions:
        if child[0] not in explored:
            childCost = math.sqrt((graph[goalState].heuristic[0]\
                                   - graph[child[0]].heuristic[0])**2\
                  +(graph[goalState].heuristic[1] \
                    - graph[child[0]].heuristic[1])**2)
            if childCost<minChildCost:
                bestNode=child[0]
                minChildCost=childCost
    if bestNode==parentNode:
        break
    else:
        parentNode=bestNode
        parentCost=minChildCost
        solution.append(parentNode)
return solution

solution = hillClimbing()
print(solution)
```

This will give ['F', 'H', 'I', 'J'] as the solution, which means that the algorithm gets stuck at J and doesn't go further towards Y since the distance of G is greater than J. Remember that the path is not important in local search. The solution should be your last node (currently that is J which happens to be local maxima).

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Write a program that implements Hill Climbing algorithms to solve this maze. Write the path followed (in the form of coordinates) and the cost of the path.

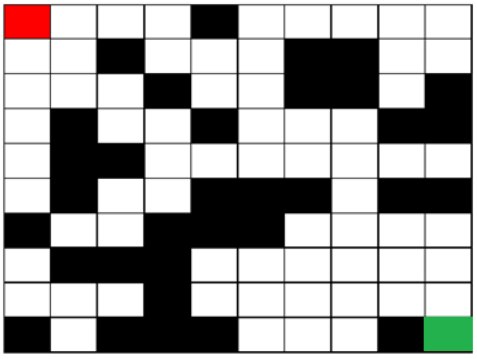


Figure 30 - Maze

Lab 10
Genetic Algorithm

Objective:

This lab will introduce students to genetic algorithms. Students will get the opportunity to get into details of genetic concepts of computation including crossover, mutation and survivor selection. This lab will also introduce students into different schemes of chromosome encoding.

Activity Outcomes:

This lab teaches you the following topics:

- How to encode chromosomes into alphabets
- How to select survivors based upon fitness function How to find global maxima using genetic algorithms

.

Instructor Note:

As pre-lab activity, read Chapters 4 from the book (Artificial Intelligence, A Modern Approach by Peter Norvig, 4th edition) to know the basics of genetic algorithms.

1) Useful concepts:

In previous lab we implemented hill climbing and saw that it can stuck at local maxima. A possible solution to avoid local maxima is to use genetic algorithms.

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

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

Foundation of Genetic Algorithms

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

1. Individual in population compete for resources and mate
2. Those individuals who are successful (fittest) then mate to create more offspring than others
3. Genes from "fittest" parent propagate throughout the generation, that is sometimes parents create offspring which is better than either parent.
4. Thus each successive generation is more suited for their environment.

Search space

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

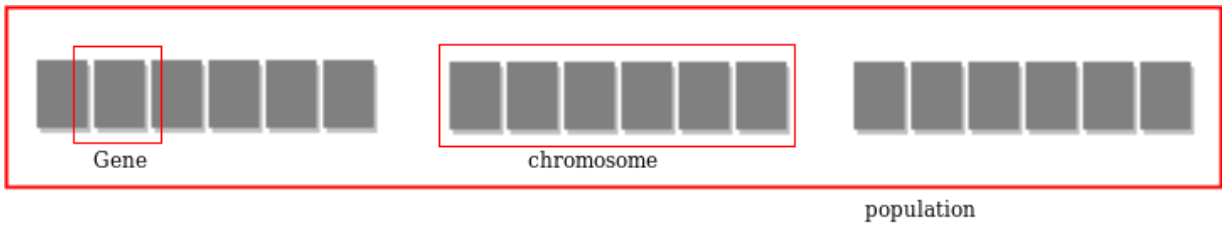


Figure 31 - Gene, Chromososme

Fitness Score

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

The GAs maintains the population of n individuals (chromosome/solutions) along with their fitness scores.The individuals having better fitness scores are given more chance to reproduce than others. The individuals with better fitness scores are selected who mate and produce **better offspring** by combining chromosomes of parents. The population size is static so the room has to be created for

new arrivals. So, some individuals die and get replaced by new arrivals eventually creating new generation when all the mating opportunity of the old population is exhausted. It is hoped that over successive generations better solutions will arrive while least fit die.
Each new generation has on average more "better genes" than the individual (solution) of previous generations. Thus each new generations have better **"partial solutions"** than previous generations. Once the offspring produced having no significant difference from offspring produced by previous populations, the population is converged. The algorithm is said to be converged to a set of solutions for the problem.

Operators of Genetic Algorithms
Once the initial generation is created, the algorithm evolves the generation using following operators –

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

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

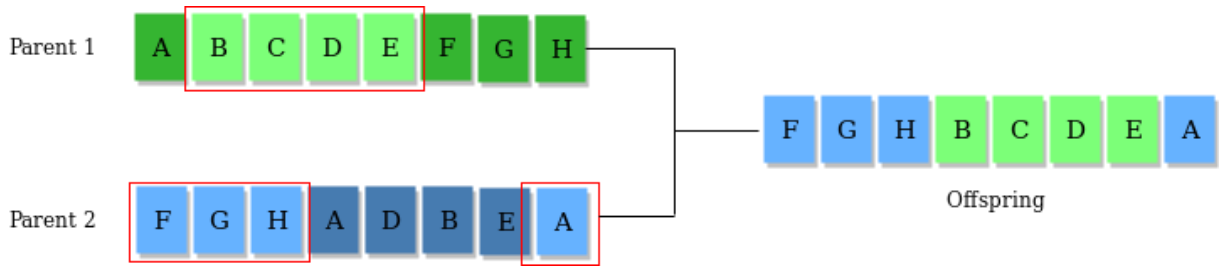


Figure 32 - crossOver

4) **Mutation Operator:** The key idea is to insert random genes in offspring to maintain the diversity in the population to avoid premature convergence. For example –

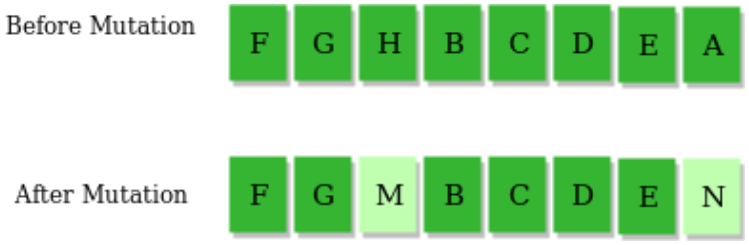


Figure 33 - Mutation

The whole algorithm can be summarized as –
1) Randomly initialize populations p
2) Determine fitness of population
3) Until convergence repeat:
   a) Select parents from population
   b) Crossover and generate new population

c) Perform mutation on new population
d) Calculate fitness for new population

2) Solved Lab Activities

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*45*	*High*	*CLO-6*

Activity 1:

Consider maxOne problem where the goal is to arrange a string of L bits into all ones. At first the solution may seem trivial i.e., for L=8 the solution is [1, 1, 1, 1, 1, 1, 1, 1]. Despite this we shall see how many iterations it will take for an instance of genetic algorithm to find the solution.

Solution:

We start by looking at the GA class which is instantiated by providing individualSize (length of chromosome or L in our case) and populationSize (how many survivors will we retain in each generation). The instance variable population is a dictionary of items where each item contains a bit array of an individual and its heuristic distance from the goal. The variable totalFitness maintains the total heuristic value of each generation/population.

```
import random
import math

class GA:
    def __init__(self, individualSize, populationSize):
        self.population=dict()
        self.individualSize = individualSize
        self.populationSize = populationSize
        self.totalFitness=0
        i=0
        while i < populationSize:
            listOfBits = [0] * individualSize
            listOfLocations = list(range(0,individualSize))
            numberOfOnes = random.randint(0, individualSize-1)
            onesLocations = random.sample(listOfLocations,numberOfOnes)
            for j in onesLocations:
                listOfBits[j]=1
                self.population[i]=[listOfBits, numberOfOnes]
                self.totalFitness = self.totalFitness + numberOfOnes
                i=i+1
```

Whenever we update current population, we also update the heuristic value of each individual/state/chromosome. This is done using method updatePopulationFitness(). The individual fitness/heuristic value is calculated as simply the number of ones in an array of an individual. The goal is to maximize this value generation after generation.

```
def updatePopulationFitness(self):
    self.totalFitness = 0
    for individual in self.population:
        individualFitness=sum(self.population[individual][0])
        self.population[individual][1] = individualFitness
        self.totalFitness = self.totalFitness + individualFitness
```

We now focus on how to select parents for reproduction. This is done using roulette wheel implementation. We first determine the size of the roulette wheel i.e., how many values will it store. This is simply set as 5 times the size of population, e.g., if we consider that we will retain 10 survivors at each generation then the size of the wheel is 50. We now have to fill these 50 values based upon the fitness value of each of those 10 individuals. This is done by calculating the probability of occurrence (h/sum(h)) of each individual. The variable individualLength determines how many values out of those 50 belong to that specific individual. Finally we generate random locations from roulette wheel populationSize times and select individuals based upon that. Those individuals now replace the original population.

```
def selectParents(self):
    rouletteWheel=[]
    wheelSize=self.populationSize*5
    h_n=[]
    for individual in self.population:
        h_n.append(self.population[individual][1])
    j=0
    for individual in self.population:
        individualLength=round(wheelSize*(h_n[j]/sum(h_n)))
        j=j+1
        if individualLength>0:
            i=0
            while i < individualLength:
                rouletteWheel.append(individual)
                i=i+1
    random.shuffle(rouletteWheel)
    parentIndices=[]
    i=0
    while i< self.populationSize:
        parentIndices.append(rouletteWheel[\
            random.randint(0, len(rouletteWheel)-1)])
        i=i+1
    newGeneration=dict()
    i=0
    while i < self.populationSize:
        newGeneration[i]=self.population[parentIndices[i]].copy()
        i=i+1
    del self.population
    self.population = newGeneration.copy()
    self.updatePopulationFitness()
```

We now come to the method generateChildren() which gerates children based upon crossover. A crossover probability is provided as input. For population size of 8 and crossover probability of 0.8 for instance, we need to do the crossover only between 80% of the 4 pairs i.e., we will do crossover only with 3 pairs and will take a pair as it is to next step (which is mutation). After doing crossover, we also

update the fitness value of that child (although this shouldn't be required since we are already calling updatePopulationFitness method at the end).

```
def generateChildren(self, crossoverProbability):
    numberOfPairs = round(crossoverProbability*self.populationSize/2)
    individualIndices = list(range(0,self.populationSize))
    random.shuffle(individualIndices)
    i=0
    j=0
    while i<numberOfPairs:
        crossoverPoint=random.randint(0, self.individualSize-1)
        child1=self.population[j][0][0:crossoverPoint]\
               +self.population[j+1][0][crossoverPoint:]
        child2=self.population[j+1][0][0:crossoverPoint]\
               +self.population[j][0][crossoverPoint:]
        self.population[j] = [child1, sum(child1)]
        self.population[j+1] = [child2, sum(child2)]
        i=i+1
        j=j+2
    self.updatePopulationFitness()
```

The next step is to mutate the population (not individual child) based upon a certain mutation probability provided as an input. For example, for individualSize of 5 bits and populationSize of 8 and a probability of 0.05, we need to swap 5% of those 8*5=40 bits, i.e., round(0.05*40) bits will be swapped.

```
def mutateChildren(self, mutationProbability):
    numberOfBits = round(mutationProbability*\
                         self.populationSize*self.individualSize)
    totalIndices = list(range(0,\
                        self.populationSize*self.individualSize))
    random.shuffle(totalIndices)
    swapLocations = random.sample(totalIndices,numberOfBits)

    for loc in swapLocations:
        individualIndex=math.floor(loc/self.individualSize)
        bitIndex=math.floor(loc%self.individualSize)

        if self.population[individualIndex][0][bitIndex]==0:
            self.population[individualIndex][0][bitIndex]=1
        else:
            self.population[individualIndex][0][bitIndex]=0
    self.updatePopulationFitness()
```

We now focus on our main function that will initialize the class. Whenever we find an individual in a generation which has ideal heuristic value, we terminate the algorithm.

```
individualSize, populationSize = 8, 10
i=0
instance = GA(individualSize,populationSize)
while True:
    instance.selectParents()
    instance.generateChildren(0.8)
    instance.mutateChildren(0.03)
    print(instance.population)
    print(instance.totalFitness)
    print(i)
    i=i+1
    found=False
    for individual in instance.population:
        if instance.population[individual][1]==individualSize:
            found=True
            break
    if found:
        break
```

3) Graded Lab Task

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Imagine an 8 queen problem, where the goal is to place 8 queens on an 8 X 8 board such that no two queens are on the same row or column or diagonal. (Before proceeding, kindly refer to lectures). A sample state is shown below.

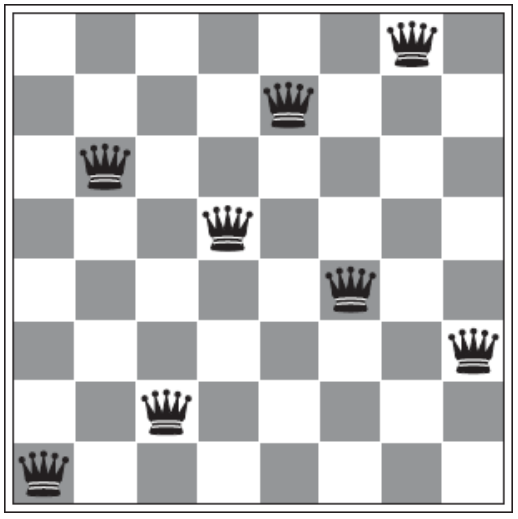


Figure 34 - 8 Queen Problem

Lab 11
Constraint Satisfaction Problem

Objective:

This lab will introduce students to solving constraint satisfaction problems.

Activity Outcomes:

This lab teaches you the following topics:

- How to solve constraint satisfaction problem.

.

Instructor Note:

As pre-lab activity, read Chapters 5 from the book (Artificial Intelligence, A Modern Approach by Peter Norvig, 4th edition) to know the basics of constraint satisfaction problems.

1) Useful concepts:

Constraint satisfaction is a technique where a problem is solved when its values satisfy certain constraints or rules of the problem. Such type of technique leads to a deeper understanding of the problem structure as well as its complexity.
Constraint satisfaction depends on three components, namely:

- **X:** It is a set of variables.
- **D:** It is a set of domains where the variables reside. There is a specific domain for each variable.
- **C:** It is a set of constraints which are followed by the set of variables.

In constraint satisfaction, domains are the spaces where the variables reside, following the problem specific constraints. These are the three main elements of a constraint satisfaction technique. The constraint value consists of a pair of **{scope, rel}**. The **scope** is a tuple of variables which participate in the constraint and **rel** is a relation which includes a list of values which the variables can take to satisfy the constraints of the problem.

Solving Constraint Satisfaction Problems
The requirements to solve a constraint satisfaction problem (CSP) is:

- A state-space
- The notion of the solution.

A state in state-space is defined by assigning values to some or all variables such as
{$X_1=v_1$, $X_2=v_2$, and so on…}.

An assignment of values to a variable can be done in three ways:

- **Consistent or Legal Assignment:** An assignment which does not violate any constraint or rule is called Consistent or legal assignment.
- **Complete Assignment:** An assignment where every variable is assigned with a value, and the solution to the CSP remains consistent. Such assignment is known as Complete assignment.
- **Partial Assignment:** An assignment which assigns values to some of the variables only. Such type of assignments are called Partial assignments.

Types of Domains in CSP
There are following two types of domains which are used by the variables :

- **Discrete Domain:** It is an infinite domain which can have one state for multiple variables. **For example,** a start state can be allocated infinite times for each variable.
- **Finite Domain:** It is a finite domain which can have continuous states describing one domain for one specific variable. It is also called a continuous domain.

Constraint Types in CSP
With respect to the variables, basically there are following types of constraints:

- **Unary Constraints:** It is the simplest type of constraints that restricts the value of a single variable.
- **Binary Constraints:** It is the constraint type which relates two variables. A value $\mathbf{x_2}$ will contain a value which lies between **x1** and **x3**.
- **Global Constraints:** It is the constraint type which involves an arbitrary number of variables.

Some special types of solution algorithms are used to solve the following types of constraints:

- **Linear Constraints:** These type of constraints are commonly used in linear programming where each variable containing an integer value exists in linear form only.

- **Non-linear Constraints:** These type of constraints are used in non-linear programming where each variable (an integer value) exists in a non-linear form.

Note: A special constraint which works in real-world is known as **Preference constraint.**

Constraint Propagation

In local state-spaces, the choice is only one, i.e., to search for a solution. But in CSP, we have two choices either:

- We can search for a solution or
- We can perform a special type of inference called **constraint propagation**.

Constraint propagation is a special type of inference which helps in reducing the legal number of values for the variables. The idea behind constraint propagation is **local consistency.**

In local consistency, variables are treated as **nodes**, and each binary constraint is treated as an **arc** in the given problem. **There are following local consistencies which are discussed below:**

- **Node Consistency:** A single variable is said to be node consistent if all the values in the variable's domain satisfy the unary constraints on the variables.
- **Arc Consistency:** A variable is arc consistent if every value in its domain satisfies the binary constraints of the variables.
- **Path Consistency:** When the evaluation of a set of two variable with respect to a third variable can be extended over another variable, satisfying all the binary constraints. It is similar to arc consistency.
- **k-consistency:** This type of consistency is used to define the notion of stronger forms of propagation. Here, we examine the k-consistency of the variables.

2) Solved Lab Activies:

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*45*	*High*	*CLO-6*

Activity 1:

Imagine you have a map of Australia that you want to color by state/territory (which we'll collectively call "regions"). No two adjacent regions should share a color. Can you color the regions with only three different colors?

The answer is yes. Try it out on your own (the easiest way is to print out a map of Australia with a white background). As human beings, we can quickly figure out the solution by inspection and a little trial and error. It's a trivial problem and a great first problem for our backtracking constraint-satisfaction solver. The problem is illustrated in figure:

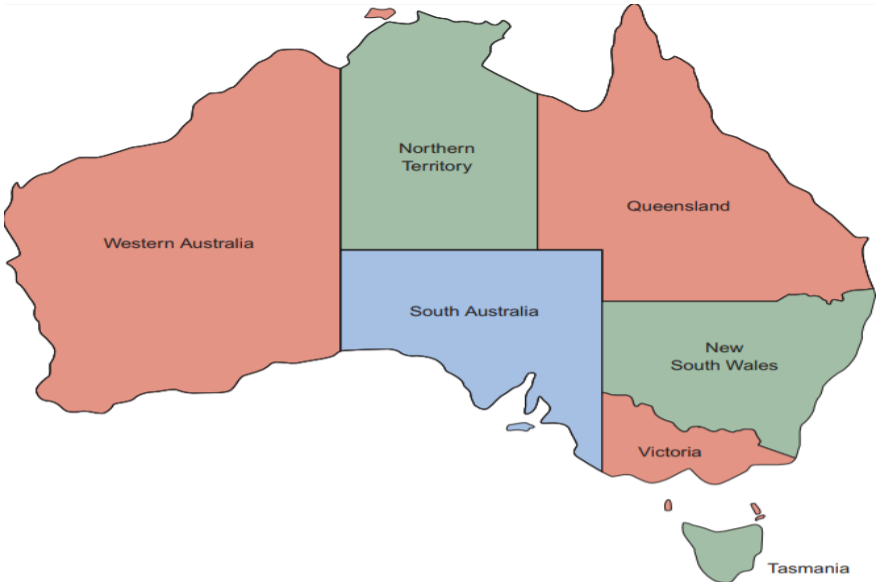


Figure 35 - CSP

To model the problem as a CSP, we need to define the variables, domains, and constraints. The variables are the seven regions of Australia (at least the seven that we'll restrict ourselves to): Western Australia; Northern Territory; South Australia; Queensland; New South Wales; Victoria; and Tasmania. In our CSP, they can be modeled with strings. The domain of each variable is the three different colors that can possibly be assigned (we'll use red, green, and blue). The constraints are the tricky part. No two adjacent regions can be colored with the same color, and our constraints are dependent on which regions border one another. We can use binary constraints (constraints between two variables). Every two regions that share a border also share a binary constraint indicating they can't be assigned the same color.

To implement these binary constraints in code, we need to subclass the `Constraint` class. The `MapColoringConstraint` subclass takes two variables in its constructor (therefore being a binary constraint): the two regions that share a border. Its overridden `satisfied()` method check whether the two regions both have a domain value (color) assigned to them—if either doesn't, the constraint's trivially satisfied until they do (there can't be a conflict when one doesn't yet have a color). Then it checks whether the two regions are assigned the same color (obviously there's a conflict, meaning the constraint isn't satisfied, when they're the same).

The class is presented here in its entirety. `MapColoringConstraint` isn't generic in terms of type hinting, but it subclasses a parameterized version of the generic class `Constraint` that indicates both variables and domains are of type `str`.

```
from csp import Constraint, CSP
from typing import Dict, List, Optional


class MapColoringConstraint(Constraint[str, str]):
    def __init__(self, place1: str, place2: str) -> None:
        super().__init__([place1, place2])
        self.place1: str = place1
        self.place2: str = place2

    def satisfied(self, assignment: Dict[str, str]) -> bool:
        # If either place is not in the assignment then it is not
        # yet possible for their colors to be conflicting
        if self.place1 not in assignment or self.place2 not in assignment:
            return True
        # check the color assigned to place1 is not the same as the
        # color assigned to place2
        return assignment[self.place1] != assignment[self.place2]
```

Now that we have a way of implementing the constraints between regions, fleshing out the Australian map-coloring problem with our CSP solver is a matter of filling in domains and variables, and then adding constraints.

```
if __name__ == "__main__":
    variables: List[str] = ["Western Australia", "Northern Territory", "South Australia",
                            "Queensland", "New South Wales", "Victoria", "Tasmania"]
    domains: Dict[str, List[str]] = {}
    for variable in variables:
        domains[variable] = ["red", "green", "blue"]
    csp: CSP[str, str] = CSP(variables, domains)
    csp.add_constraint(MapColoringConstraint("Western Australia", "Northern Territory"))
    csp.add_constraint(MapColoringConstraint("Western Australia", "South Australia"))
    csp.add_constraint(MapColoringConstraint("South Australia", "Northern Territory"))
    csp.add_constraint(MapColoringConstraint("Queensland", "Northern Territory"))
    csp.add_constraint(MapColoringConstraint("Queensland", "South Australia"))
    csp.add_constraint(MapColoringConstraint("Queensland", "New South Wales"))
    csp.add_constraint(MapColoringConstraint("New South Wales", "South Australia"))
    csp.add_constraint(MapColoringConstraint("Victoria", "South Australia"))
    csp.add_constraint(MapColoringConstraint("Victoria", "New South Wales"))
    csp.add_constraint(MapColoringConstraint("Victoria", "Tasmania"))
```

Finally, backtracking_search() is called to find a solution.

```
solution: Optional[Dict[str, str]] = csp.backtracking_search()
if solution is None:
    print("No solution found!")
else:

    print(solution)
```

A correct solution includes an assigned color for every region.

```
{'Western Australia': 'red', 'Northern Territory': 'green',
'South Australia': 'blue', 'Queensland': 'red', 'New South Wales':
'green', 'Victoria': 'red', 'Tasmania': 'green'}
```

3) Graded Lab Tasks:

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

The four queens problem

In a four queens problem, a board is a four-by-four grid of squares. A queen is a chess piece that can move on the chessboard any number of squares along any row, column, or diagonal. A queen is attacking another piece if, in a single move, it can move to the square the piece is on without jumping over any other piece. If the other piece is in the line of sight of the queen, then it's attacked by it). The four queens problem poses the question of how four queens can be placed on a chessboard without any queen attacking another queen. The problem is illustrated in figure:

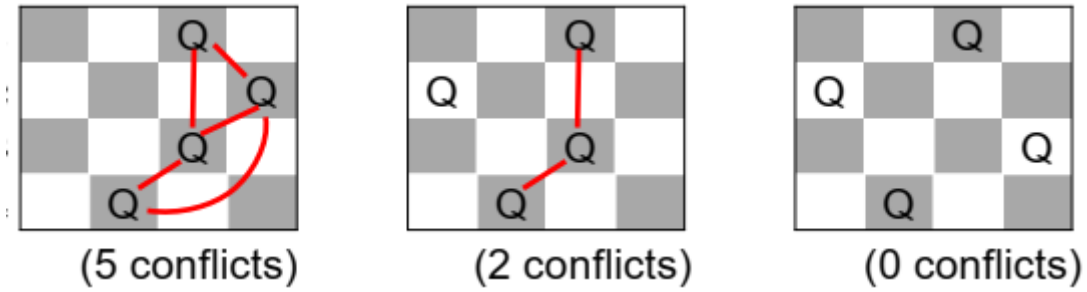


Figure 36 - 4 Queens problem

Solve the given problem modeling it as a CSP

Lab 12
Introduction to Prolog

Objective:

This lab will introduce students to Prolog. Prolog is a programming language, but a rather unusual one. ``Prolog" is short for ``Programming with Logic", and the link with logic gives Prolog its special character. At the heart of Prolog lies a surprising idea: don't tell the computer what to do. Instead, describe situations of interest, and compute by asking questions. Prolog will logically deduce new facts about the situations and give its deductions back to us as answers..

Activity Outcomes:

This lab teaches you the following topics:

- How to install and use the interface of Prolog How to create a simple Knowledge Base How to use simple queries

.

Instructor Note:

As pre-lab activity, read Chapter 1 from the book (Learn Prolog Now, Vol 7, by BlackBurn et. al.,) to know the basics of prolog programming

1) Useful concepts:

Prolog or **PRO**gramming in **LOG**ics is a logical and declarative programming language. It is one major example of the fourth generation language that supports the declarative programming paradigm. This is particularly suitable for programs that involve **symbolic** or **non-numeric computation**. This is the main reason to use Prolog as the programming language in **Artificial Intelligence**, where **symbol manipulation** and **inference manipulation** are the fundamental tasks.

In Prolog, we need not mention the way how one problem can be solved, we just need to mention what the problem is, so that Prolog automatically solves it. However, in Prolog we are supposed to give clues as the **solution method**.

Key Features :

1. Unification : The basic idea is, can the given terms be made to represent the same structure.
2. Backtracking : When a task fails, prolog traces backwards and tries to satisfy previous task.
3. Recursion : Recursion is the basis for any search in program.

Prolog language basically has three different elements –

Facts

We can define fact as an explicit relationship between objects, and properties these objects might have. So facts are unconditionally true in nature. Suppose we have some facts as given below −

- Tom is a cat
- Kunal loves to eat Pasta
- Hair is black
- Nawaz loves to play games
- Pratyusha is lazy.

So these are some facts, that are unconditionally true. These are actually statements, that we have to consider as true.

Following are some guidelines to write facts −

- Names of properties/relationships begin with lower case letters.
- The relationship name appears as the first term.
- Objects appear as comma-separated arguments within parentheses.
- A period "." must end a fact.
- Objects also begin with lower case letters. They also can begin with digits (like 1234), and can be strings of characters enclosed in quotes e.g. color(penink, ‘red’).
- phoneno(agnibha, 1122334455). is also called a predicate or clause.

Syntax

The syntax for facts is as follows −

```
relation(object1,object2...).
```

Example

Following is an example of the above concept −

```
cat(tom).
loves_to_eat(kunal,pasta).
of_color(hair,black).
loves_to_play_games(nawaz).
lazy(pratyusha).
```

Rules

We can define rule as an implicit relationship between objects. So facts are conditionally true. So when one associated condition is true, then the predicate is also true. Suppose we have some rules as given below −

- Lili is happy if she dances.
- Tom is hungry if he is searching for food.
- Jack and Bili are friends if both of them love to play cricket.
- will go to play if school is closed, and he is free.

So these are some rules that are **conditionally** true, so when the right hand side is true, then the left hand side is also true.

Here the symbol (:-) will be pronounced as “If”, or “is implied by”. This is also known as neck symbol, the LHS of this symbol is called the Head, and right hand side is called Body. Here we can use comma (,) which is known as conjunction, and we can also use semicolon, that is known as disjunction.

Syntax

```
rule_name(object1, object2, ...) :- fact/rule(object1,
 object2, ...)
Suppose a clause is like :
P :- Q;R.
This can also be written as
P :- Q.
P :- R.

If one clause is like :
P :- Q,R;S,T,U.

Is understood as
P :- (Q,R);(S,T,U).
Or can also be written as:
P :- Q,R.
P :- S,T,U.
```

Example

```
happy(lili) :- dances(lili).
hungry(tom) :- search_for_food(tom).
friends(jack, bili) :- lovesCricket(jack), lovesCricket(bili).
goToPlay(ryan) :- isClosed(school), free(ryan).
```

Queries

Queries are some questions on the relationships between objects and object properties. So question can be anything, as given below −

- Is tom a cat?
- Does Kunal love to eat pasta?
- Is Lili happy?
- Will Ryan go to play?

So according to these queries, Logic programming language can find the answer and return them.

2) Solved Lab Activies

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*10*	*Low*	*CLO-6*
2	*10*	*Medium*	*CLO-6*
3	*10*	*Medium*	*CLO-6*
4	*15*	*Medium*	*CLO-6*

Activity 1:

How to add facts in a Knowledge Base?

Solution:

Knowledge Base 1 (KB1) is simply a collection of facts. Facts are used to state things that are unconditionally true of the domain of interest. For example, we can state that Mia, Jody, and Yolanda are women, and that Jody plays air guitar, using the following four facts:

```
woman(mia). woman(jody). woman(yolanda). playsAirGuitar(jody).
```

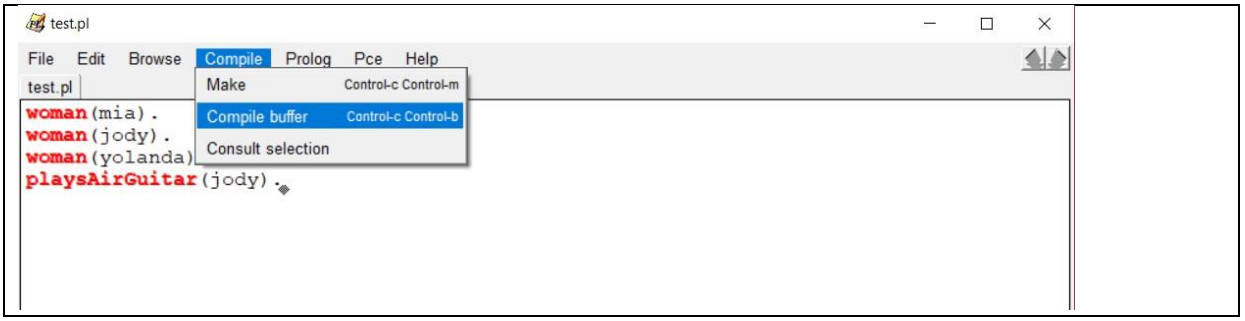
Do not forget to enter a sentence with a period (.).

Activity 2:

How to run a query within a Knowledge Base?

Solution:

How can we use KB1? By posing queries. That is, by asking questions about the information KB1 contains. But first compile the program using compile option as shown in the following figure,



Now we can ask Prolog whether Mia is a woman by posing the query in the command prompt.

?- woman(mia).

Prolog will answer true for the obvious reason that this is one of the facts explicitly recorded in KB1. Incidentally, *we* don't type in the ?-. This symbol (or something like it, depending on the implementation of Prolog you are using) is the prompt symbol that the Prolog interpreter displays when

it is waiting to evaluate a query. We just type in the actual query (for example woman(mia)) followed by .(a full stop).

Activity 3:

How to add rules alongside facts in a Knowledge Base?

Solution:
Here is KB2, our second knowledge base:

```
listensToMusic(mia). happy(yolanda).
playsAirGuitar(mia) :- listensToMusic(mia). playsAirGuitar(yolanda) :- listensToMusic(yolanda).
listensToMusic(yolanda):- happy(yolanda).
```

KB2 contains two facts, listensToMusic(mia) and happy(yolanda). The last three items are rules.

Rules state information that is *conditionally* true of the domain of interest. For example, the first rule says that Mia plays air guitar *if* she listens to music, and the last rule says that Yolanda listens to music *if* she if happy. More generally, the :- should be read as "if", or "is implied by". The part on the left hand side of the :- is called the head of the rule, the part on the right hand side is called the body. So in general rules say: *if* the body of the rule is true, *then* the head of the rule is true too. And now for the key point: *if a knowledge base contains a rule* head :- body, *and Prolog knows that* body *follows from the information in the knowledge base, then Prolog can infer* head.

This fundamental deduction step is what logicians call modus ponens.

Let's consider an example. We will ask Prolog whether Mia plays air guitar:
?- playsAirGuitar(mia).

Prolog will respond "yes". Why? Well, although playsAirGuitar(mia) is not a fact explicitly recorded in KB2, KB2 does contain the rule playsAirGuitar(mia) :- listensToMusic(mia). Moreover, KB2 also contains the fact listensToMusic(mia). Hence Prolog can use modus ponens to deduce that playsAirGuitar(mia).

Our next example shows that Prolog can chain together uses of modus ponens. Suppose we ask:
?- playsAirGuitar(yolanda). Prolog would respond "yes". Why? Well, using the fact happy(yolanda) and the rule listensToMusic(yolanda):- happy(yolanda), Prolog can deduce the new fact listensToMusic(yolanda). This new fact is not explicitly recorded in the knowledge base — it is only *implicitly* present (it is *inferred* knowledge). Nonetheless, Prolog can then use it just like an explicitly recorded fact. Thus, together with the rule playsAirGuitar(yolanda) :- listensToMusic(yolanda) it can deduce that playsAirGuitar(yolanda), which is what we asked it. Summing up: any fact produced by an application of modus ponens can be used as input to further rules. By chaining together applications of modus ponens in this way, Prolog is able to retrieve information that logically follows from the rules and facts recorded in the knowledge base.

The facts and rules contained in a knowledge base are called clauses. Thus KB2 contains five clauses, namely three rules and two facts. Another way of looking at KB2 is to say that it consists of three predicates (or procedures). The three predicates are:
listensToMusic happy playsAirGuitar

The happy predicate is defined using a single clause (a fact). The listensToMusic and playsAirGuitar predicates are each defined using two clauses (in both cases, two rules). It is a good idea to think about Prolog programs in terms of the predicates they contain. In essence, the predicates are the concepts we find important, and the various clauses we write down concerning them are our attempts to pin down what they mean and how they are inter-related.

One final remark. We can view a fact as a rule with an empty body. That is, we can think of facts as "conditionals that do not have any antecedent conditions", or "degenerate rules"

Activity 4:

Use conjunction based rules in a Knowledge Base?

Solution:
KB3, our third knowledge base, consists of five clauses:

```
happy(vincent). listensToMusic(butch). playsAirGuitar(vincent):- listensToMusic(vincent), happy(vincent). playsAirGuitar(butch):- happy(butch). playsAirGuitar(butch):- listensToMusic(butch).
```

There are two facts, namely happy(vincent) and listensToMusic(butch), and three rules. KB3 defines the same three predicates as KB2 (namely happy, listensToMusic, and playsAirGuitar) but it defines them differently. In particular, the three rules that define the playsAirGuitar predicate introduce some new ideas. First, note that the rule,

playsAirGuitar(vincent):- listensToMusic(vincent), happy(vincent).

has *two* items in its body, or (to use the standard terminology) two goals. What does this rule mean? The important thing to note is the comma , that separates the goal listensToMusic(vincent) and the goal happy(vincent) in the rule's body. This is the way logical conjunction is expressed in Prolog (that is, the comma means *and*). So this rule says: "Vincent plays air guitar if he listens to music and he is happy".

Thus, if we posed the query
?- playsAirGuitar(vincent).

Prolog would answer "no". This is because while KB3 contains happy(vincent), it does *not* explicitly contain the information listensToMusic(vincent), and this fact cannot be deduced either. So KB3 only fulfils one of the two preconditions needed to establish playsAirGuitar(vincent), and our query fails. Incidentally, the spacing used in this rule is irrelevant. For example, we could have written it as
playsAirGuitar(vincent):- happy(vincent),listensToMusic(vincent). and it would have meant exactly the same thing. Prolog offers us a lot of freedom in the way we set out knowledge bases, and we can take

advantage of this to keep our code readable. Next, note that KB3 contains two rules with *exactly* the same head, namely:
playsAirGuitar(butch):-
happy(butch). playsAirGuitar(butch):- listensToMusic(butch).

This is a way of stating that Butch plays air guitar if *either* he listens to music, *or* if he is happy. That is, listing multiple rules with the same head is a way of expressing logical disjunction (that is, it is a way of saying *or*). So if we posed the query ?- playsAirGuitar(butch). Prolog would answer "yes". For although the first of these rules will not help (KB3 does not allow
Prolog to conclude that happy(butch)), KB3 *does* contain listensToMusic(butch) and this means Prolog can apply modus ponens using the rule playsAirGuitar(butch):- listensToMusic(butch). to conclude that playsAirGuitar(butch). There is another way of expressing disjunction in Prolog. We could replace the pair of rules given above by the single rule
playsAirGuitar(butch):-
happy(butch); listensToMusic(butch).

That is, the semicolon ; is the Prolog symbol for *or*, so this single rule means exactly the same thing as the previous pair of rules. But Prolog programmers usually write multiple rules, as extensive use of semicolon can make Prolog code hard to read. It should now be clear that Prolog has something do with logic: after all, the :- means implication, the , means conjunction, and the ; means disjunction. (What about negation? That is a whole other story. We'll be discussing it later in the course.) Moreover, we have seen that a standard logical proof rule (modus ponens) plays an important role in Prolog programming. And in fact "Prolog" is short for "Programming in logic".

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Create a knowledge base which defines your family tree and make a query that uses application of modus ponens to derive a fact which is not explicitly elaborated in the knowledge base.

Lab 13
Complex Knowledge-bases

Objective:

This lab will introduce students to complex knowledge bases in Prolog. Students will also get in depth of prolog syntax and the process of deducing facts from knowledge base.

Activity Outcomes:

This lab teaches you the following topics:

- How to use variables in Prolog
- How to create complex queries
- How to create complex knowledge base and use it for deduction

.

Instructor Note:

As pre-lab activity, read Chapter 1 from the book (Learn Prolog Now, Vol 7, by BlackBurn et. al.,) to know the basics of prolog programming

1) Useful concepts:

Creating complex knowledge bases poses a challenge in Prolog. It requires user to get familiar with notation and program flow which is quite different in prolog compared to other programming languages.

Knowledge Base in Logic Programming

There are three main components in logic programming − **Facts, Rules** and **Queries**. Among these three if we collect the facts and rules as a whole then that forms a **Knowledge Base**. So we can say that the **knowledge base** is a **collection of facts and rules**.

Now, we will see how to write some knowledge bases. Suppose we have our very first knowledge base called KB1. Here in the KB1, we have some facts. The facts are used to state things, that are unconditionally true of the domain of interest.

Knowledge Base 1

Suppose we have some knowledge, that Priya, Tiyasha, and Jaya are three girls, among them, Priya can cook. Let's try to write these facts in a more generic way as shown below −

```
girl(priya).
girl(tiyasha).
girl(jaya).
can_cook(priya).
```

Here we have written the name in lowercase letters, because in Prolog, a string starting with uppercase letter indicates a **variable**.

Now we can use this knowledge base by posing some queries. "Is priya a girl?", it will reply "yes", "is jamini a girl?" then it will answer "No", because it does not know who jamini is. Our next question is "Can Priya cook?", it will say "yes", but if we ask the same question for Jaya, it will say "No".

Output

```
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
| ?- change_directory('D:/TP Prolog/Sample_Codes').

yes
| ?- [kb1]
.
compiling D:/TP Prolog/Sample_Codes/kb1.pl for byte code...
D:/TP Prolog/Sample_Codes/kb1.pl compiled, 3 lines read - 489 bytes written, 10 ms

yes
| ?- girl(priya)
.

yes
| ?- girl(jamini).
```

```
no
| ?- can_cook(priya).

yes
| ?- can_cook(jaya).

no
| ?-
```

Knowledge Base 2

Let us see another knowledge base, where we have some rules. Rules contain some information that are conditionally true about the domain of interest. Suppose our knowledge base is as follows –

```
sing_a_song(ananya).
listens_to_music(rohit).

listens_to_music(ananya) :- sing_a_song(ananya).
happy(ananya) :- sing_a_song(ananya).
happy(rohit) :- listens_to_music(rohit).
playes_guitar(rohit) :- listens_to_music(rohit).
```

So there are some facts and rules given above. The first two are facts, but the rest are rules. As we know that Ananya sings a song, this implies she also listens to music. So if we ask “Does Ananya listen to music?”, the answer will be true. Similarly, “is Rohit happy?”, this will also be true because he listens to music. But if our question is “does Ananya play guitar?”, then according to the knowledge base, it will say “No”. So these are some examples of queries based on this Knowledge base.

Output

```
| ?- [kb2].
compiling D:/TP Prolog/Sample_Codes/kb2.pl for byte code...
D:/TP Prolog/Sample_Codes/kb2.pl compiled, 6 lines read - 1066 bytes written, 15 ms

yes
| ?- happy(rohit).

yes
| ?- sing_a_song(rohit).

no
| ?- sing_a_song(ananya).

yes
| ?- playes_guitar(rohit).

yes
| ?- playes_guitar(ananya).
```

```
no
| ?- listens_to_music(ananya).

yes
| ?-
```

Knowledge Base 3

The facts and rules of Knowledge Base 3 are as follows –

```
can_cook(priya).
can_cook(jaya).
can_cook(tiyasha).

likes(priya,jaya) :- can_cook(jaya).
likes(priya,tiyasha) :- can_cook(tiyasha).
```

Suppose we want to see the members who can cook, we can use one **variable** in our query. The variables should start with uppercase letters. In the result, it will show one by one. If we press enter, then it will come out, otherwise if we press semicolon (;), then it will show the next result.

Let us see one practical demonstration output to understand how it works.

Output

```
| ?- [kb3].
compiling D:/TP Prolog/Sample_Codes/kb3.pl for byte code...
D:/TP Prolog/Sample_Codes/kb3.pl compiled, 5 lines read - 737 bytes written, 22 ms
warning: D:/TP Prolog/Sample_Codes/kb3.pl:1: redefining procedure can_cook/1
        D:/TP Prolog/Sample_Codes/kb1.pl:4: previous definition

yes
| ?- can_cook(X).

X = priya ? ;

X = jaya ? ;

X = tiyasha

yes
| ?- likes(priya,X).

X = jaya ? ;

X = tiyasha

yes
| ?-
```

2) Solved Lab Activies:

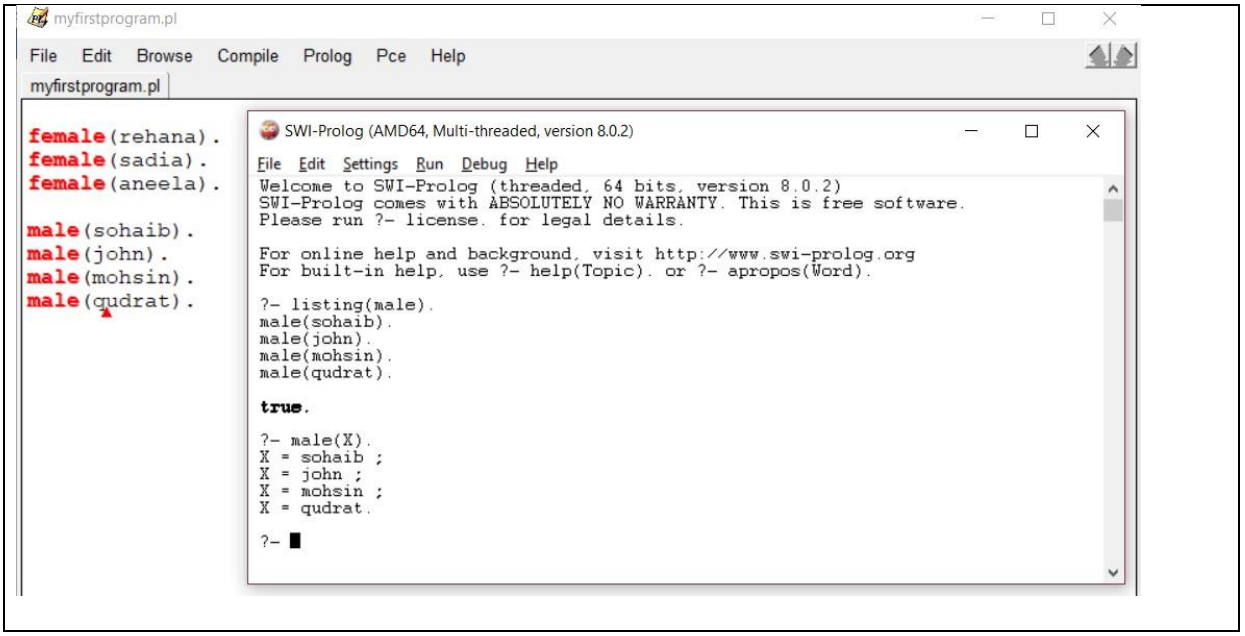
Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*5*	*Medium*	*CLO-6*
2	*5*	*Low*	*CLO-6*
3	*5*	*Medium*	*CLO-6*
4	*5*	*Low*	*CLO-6*
5	*5*	*Low*	*CLO-6*
6	*5*	*Medium*	*CLO-6*
7	*5*	*Medium*	*CLO-6*
8	*10*	*Medium*	*CLO-6*

Activity 1:

Using variables and listing functors in prolog

Solution:

A functor is simply the function name and the number of arguments of that functor is called arity. For instance, woman(mia) is a complex term with functor woman and arity 1, while loves(vincent,mia) is a complex term with functor loves and arity 2.



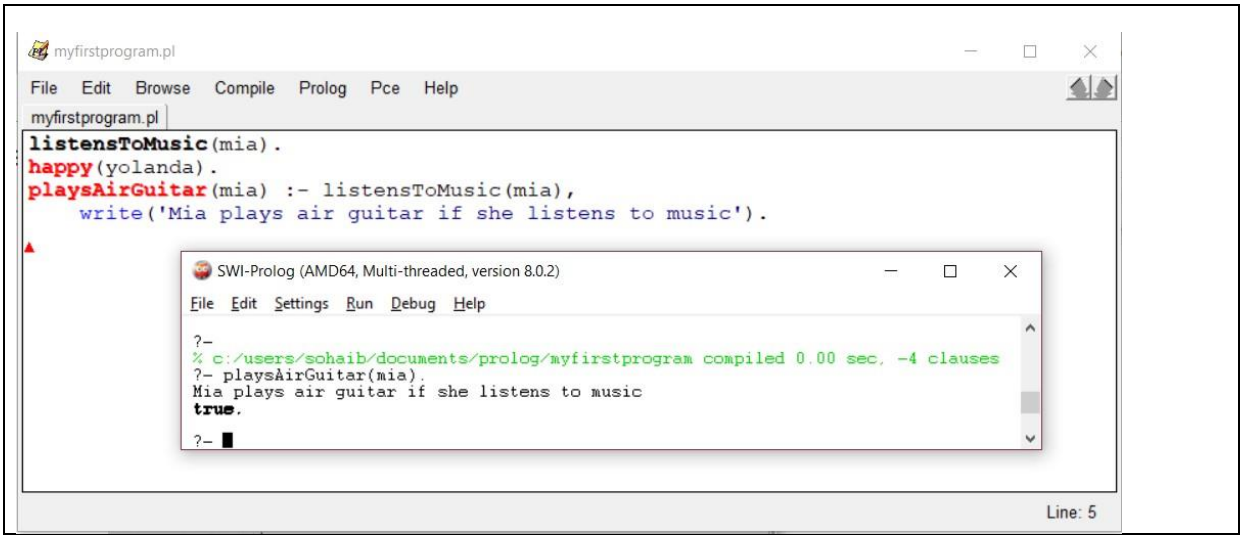
The listing method provides a way to get all the facts related to a particular functor. A variable is started by a capital letter or an underscore, so typing male(X) provides a list of males which you can get one by one by entering semicolon (;) in console.

Likewise typing "male(X), female(Y)." in console will list all possible combinations of males and females (instead of typing semicolon; you can press enter to exist the search).

Activity 2:

How to print text in prolog?

Solution:



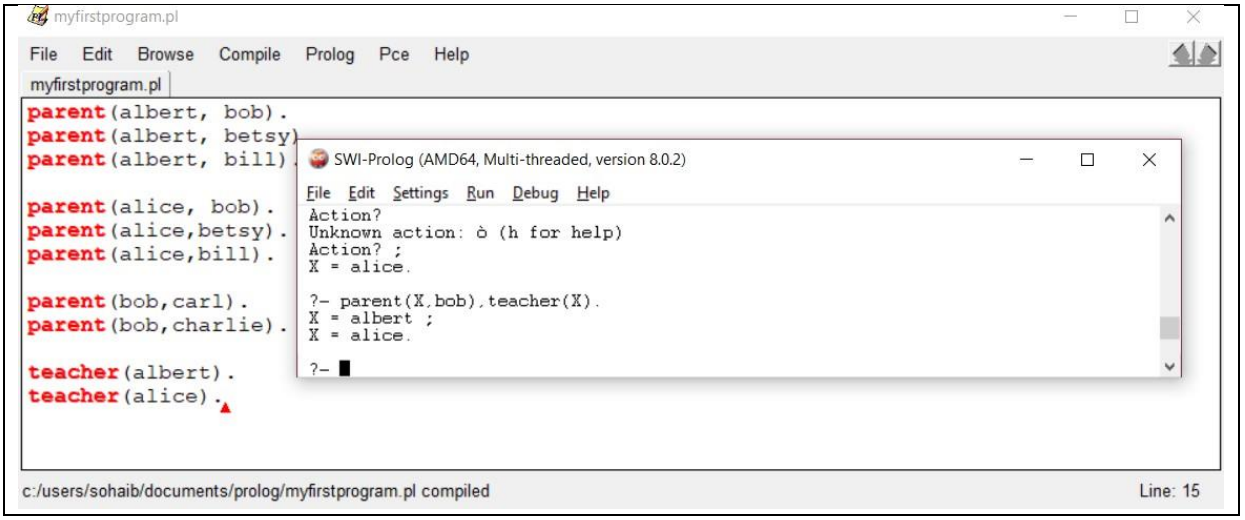
We can print custom text even within conditionals as shown above.

Activity 3:

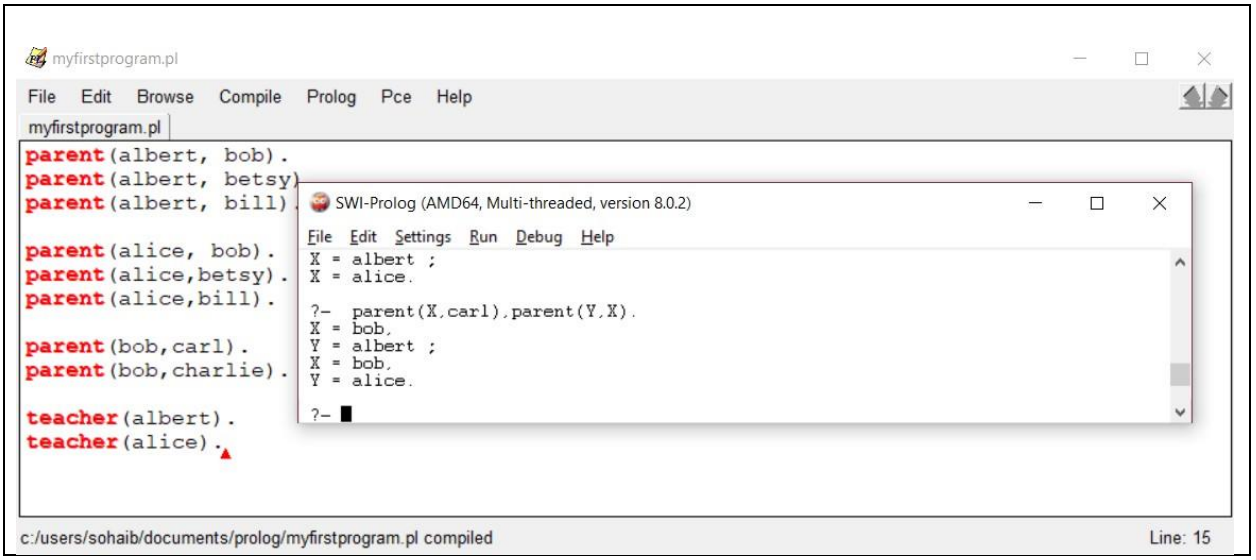
How to ask complex queries?

Solution:

Suppose that we define a knowledgebase containing parenthood relationship alongside the teaching relationship. WE then ask the question that "give me that person X which is a parent of bob and also a teacher". WE would do that as follows,



Suppose that we want to ask if Carl has a grandparent. We would do that as follows,

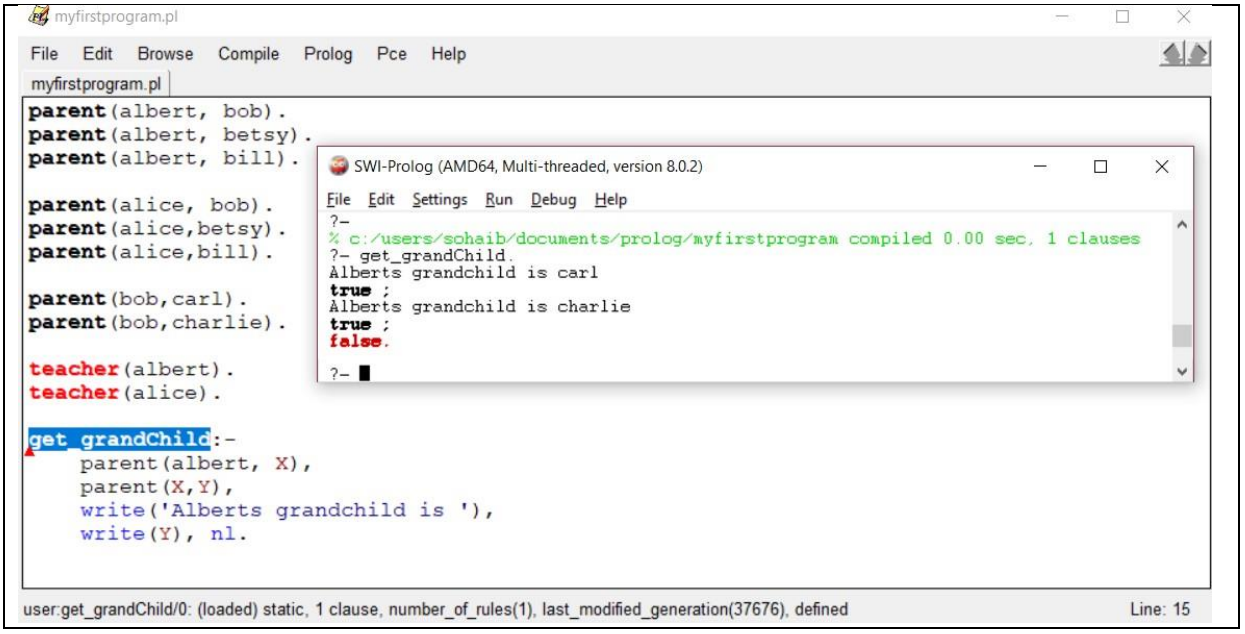


Activity 4:

Write a query in a similar fashion to determine grandchildren of Albert in above knowledgebase.

Solution:

We can also define get_GrandChild as a rule in the knowledgebase,



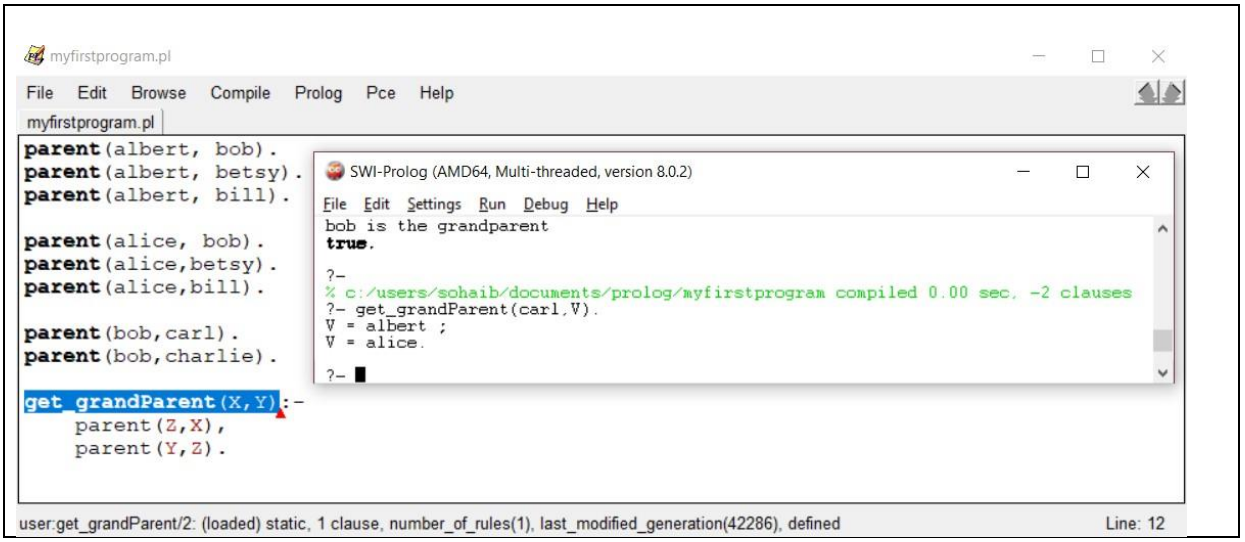
Let's see if Carl and Charlie share a parent,

?- parent(X,carl),parent(X,charlie). X = bob.

Activity 5:

We can also define variables within a consequent of a predicate which helps us to find grandparent of any X.

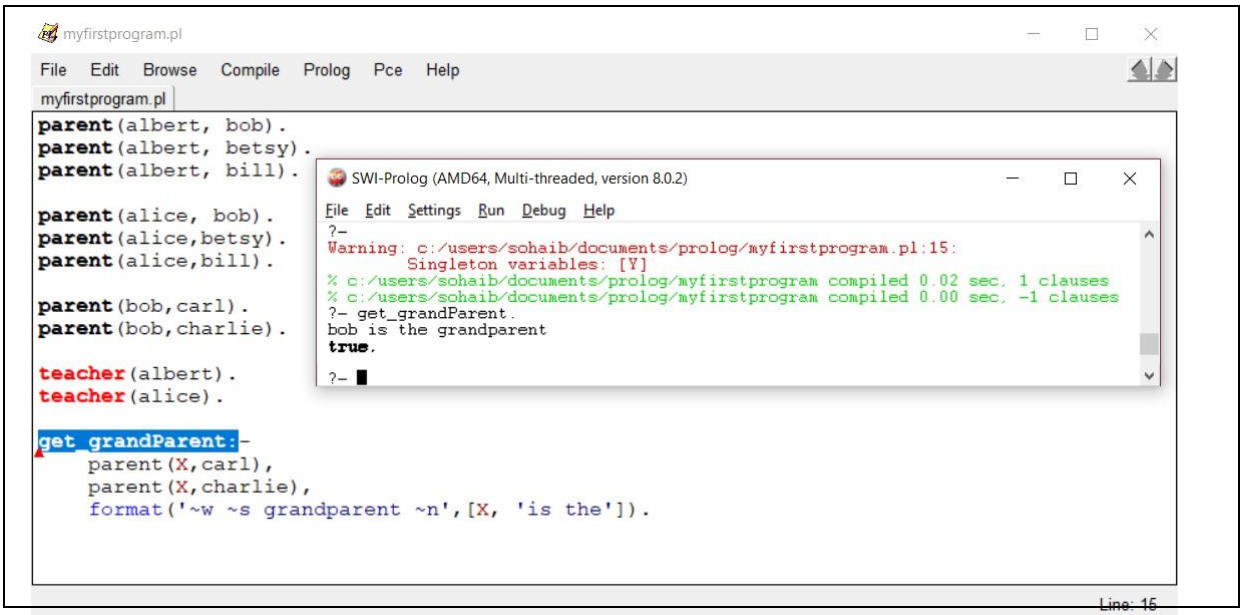
Solution:



Activity 6:

How to use format command to print inside a Knowledge Base?

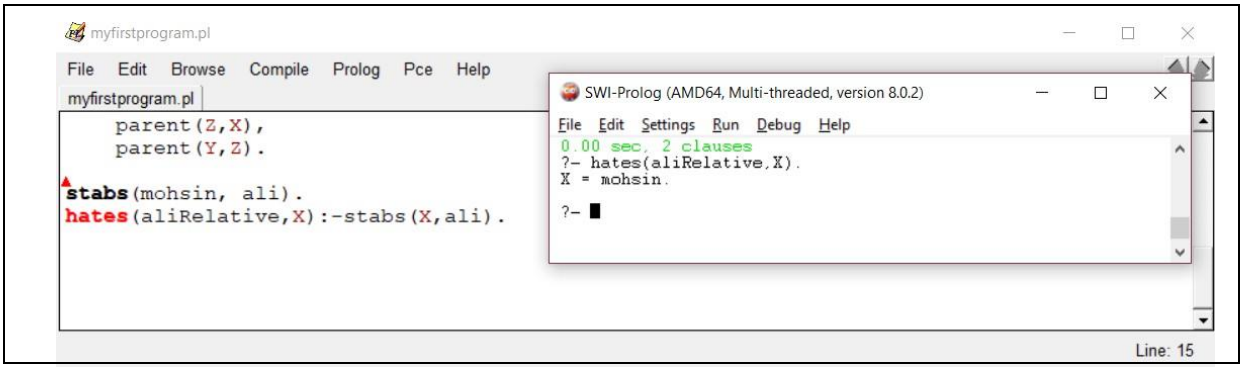
Solution:



Activity 7:

Deriving facts using variables?

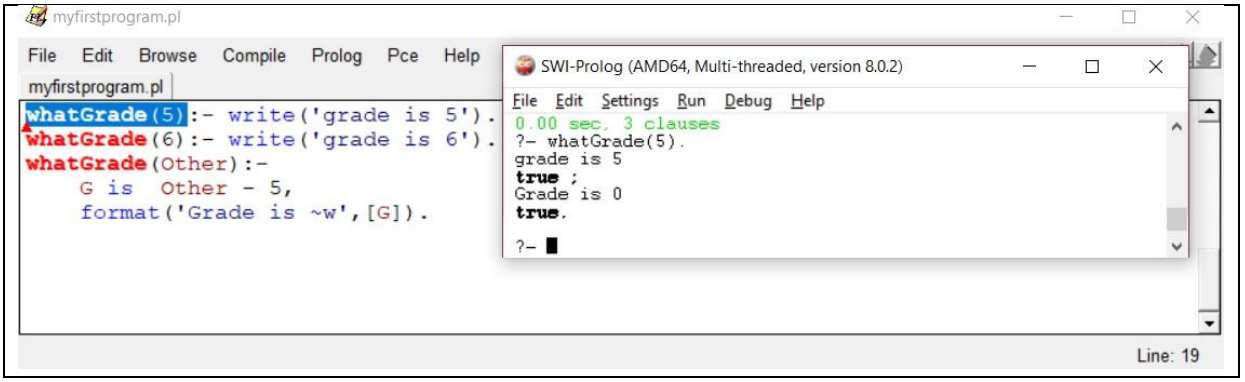
Solution:



Activity 8:

Using a single predicate with constants and variables as arguments?

Solution:



3) Graded Lab Tasks:

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Consider the knowledgebase in activity 3. Define a brotherhood relationship and write a query that gives every uncle of a person X in the knowledgebase. An uncle is defined as brother of one's parent.

Lab Task 2

Consider the knowledge base of activity 4. Write a query to find all pairs that share grand-parenthood relationship.

Lab Task 3

Resistance and Resistive Circuits - write a prolog program that will help us find the equivalent resistance of a resistive circuit.

Let us consider the following circuit to understand this concept −

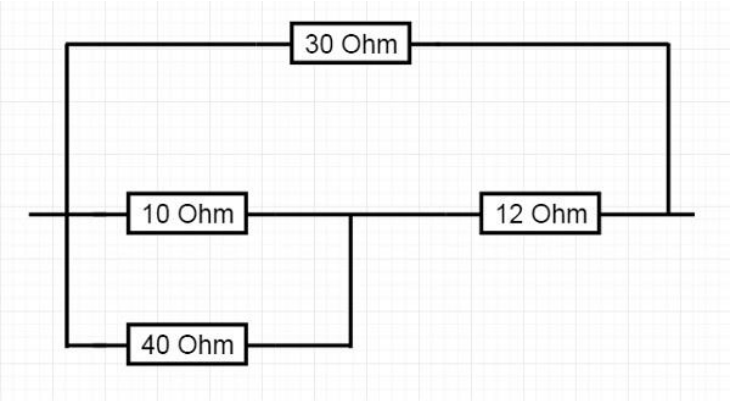


Figure 37 - Circuit

We have to find the equivalent resistance of this network. At first, we will try to get the result by hand, then try to see whether the result is matching with the prolog output or not.

We know that there are two rules −

- *If R1 and R2 are in Series, then equivalent resistor Re = R1 + R2.*
- *If R1 and R2 are in Parallel, then equivalent resistor Re = (R1 * R2)/(R1 + R2).*

Here 10 Ohm and 40 Ohm resistors are in parallel, then that is in series with 12 Ohm, and the equivalent resistor of the lower half is parallel with 30 Ohm. So let's try to calculate the equivalent resistance.

- *R3 = (10 * 40)/(10 + 40) = 400/50 = 8 Ohm*
- *R4 = R3 + 12 = 8 + 12 = 20 Ohm*
- *R5 = (20 * 30)/(20 + 30) = 12 Ohm*

Lab Task 4

*Given figure below write relevant facts and rule "**encloses**". There will be two rules written to solve this problem…. Keep in mind the recursive case while writing these rules) that is true if its first argument encloses the second argument. For example **encloses(b1,b6)** will generate **true.***

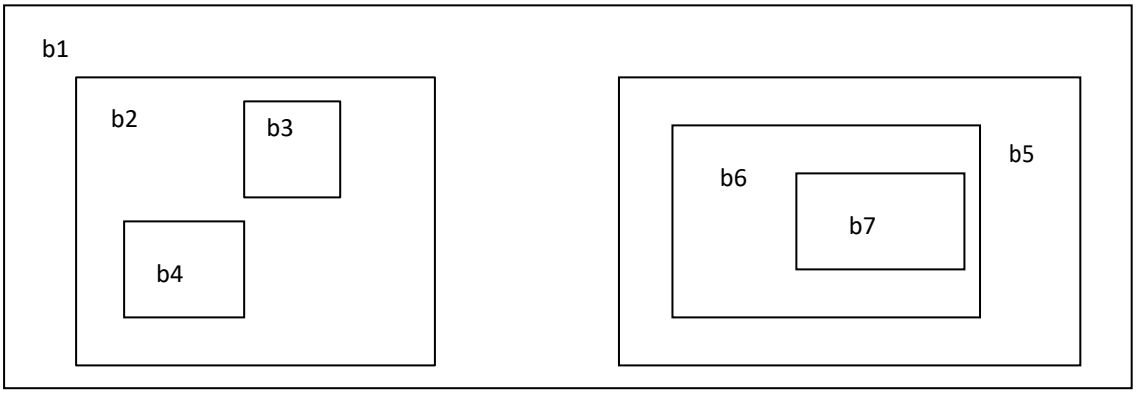


Figure 38 - Encloses

Lab 14
Expert System

Objective:

This lab will enable students to utilize their knowledge of Prolog and will enable students to write their own simple expert system in prolog

.

Activity Outcomes:

This lab teaches you the following topics:

- To enable students to use all their acquired knowledge of prolog to develop their first expert system in prolog (basic level)

.

Instructor Note:

As pre-lab activity, read Chapter 1 from the book (Learn Prolog Now, Vol 7, by BlackBurn et. al.,) to know the basics of prolog programming. For detailed examples please refer to Visual Prolog Version 5.0, Language Tutorial, Prolog Development Center A/S, H.J. Holst Vej 3A-5A, Copenhagen DK-2605 Broendby, Denmark

1) Useful concepts:

Creating complex knowledge bases poses a challenge in Prolog. It requires user to get familiar with notation and program flow which is quite different in prolog compared to other programming languages.

Artificial Intelligence is a piece of software that simulates the behaviour and judgement of a human or an organization that has experts in a particular domain is known as an expert system. It does this by acquiring relevant knowledge from its knowledge base and interpreting it according to the user's problem. The data in the knowledge base is added by humans that are expert in a particular domain and this software is used by a non-expert user to acquire some information. It is widely used in many areas such as medical diagnosis, accounting, coding, games etc.
An expert system is AI software that uses knowledge stored in a knowledge base to solve problems that would usually require a human expert thus preserving a human expert's knowledge in its knowledge base. They can advise users as well as provide explanations to them about how they reached a particular conclusion or advice. **Knowledge Engineering** is the term used to define the process of building an Expert System and its practitioners are called **Knowledge Engineers**. The primary role of a knowledge engineer is to make sure that the computer possesses all the knowledge required to solve a problem. The knowledge engineer must choose one or more forms in which to represent the required knowledge as a symbolic pattern in the memory of the computer.

Components of an Expert System :

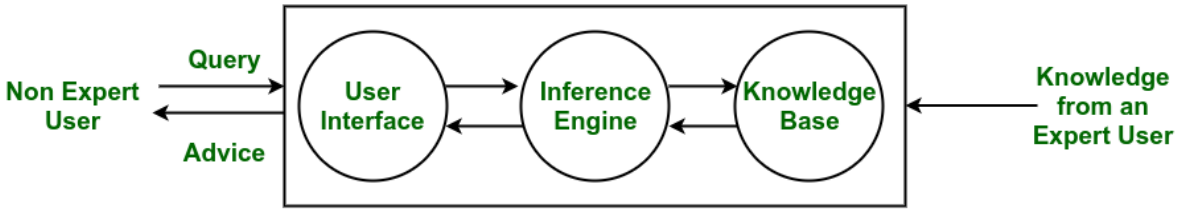


Figure 39 - Expert System

Architecture of an Expert System

1. **Knowledge Base –**

The knowledge base represents facts and rules. It consists of knowledge in a particular domain as well as rules to solve a problem, procedures and intrinsic data relevant to the domain.

2. **Inference Engine –**

The function of the inference engine is to fetch the relevant knowledge from the knowledge base, interpret it and to find a solution relevant to the user's problem. The inference engine acquires the rules from its knowledge base and applies them to the known facts to infer new facts. Inference engines can also include an explanation and debugging abilities.

3. **Knowledge Acquisition and Learning Module –**

The function of this component is to allow the expert system to acquire more and more knowledge from various sources and store it in the knowledge base.

4. **User Interface –**

This module makes it possible for a non-expert user to interact with the expert system and find a solution to the problem.

5. **Explanation Module –**

This module helps the expert system to give the user an explanation about how the expert system reached a particular conclusion.
The Inference Engine generally uses two strategies for acquiring knowledge from the Knowledge Base, namely –

- Forward Chaining
- Backward Chaining

6. Forward Chaining –

Forward Chaining is a strategic process used by the Expert System to answer the questions – What will happen next. This strategy is mostly used for managing tasks like creating a conclusion, result or effect. Example – prediction or share market movement status.

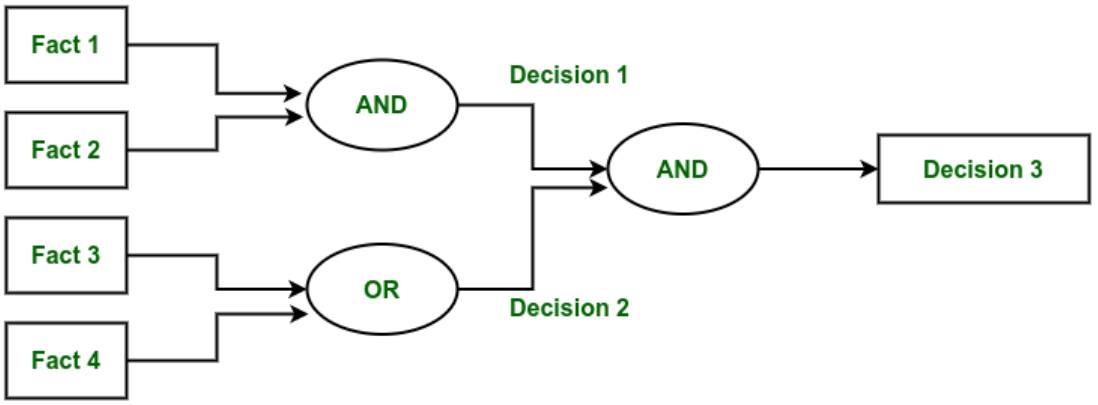


Figure 40 - Forward Chaining

7. Backward Chaining –

Backward Chaining is a storage used by the Expert System to answer the questions – Why this has happened. This strategy is mostly used to find out the root cause or reason behind it, considering what has already happened. Example – diagnosis of stomach pain, blood cancer or dengue, etc.

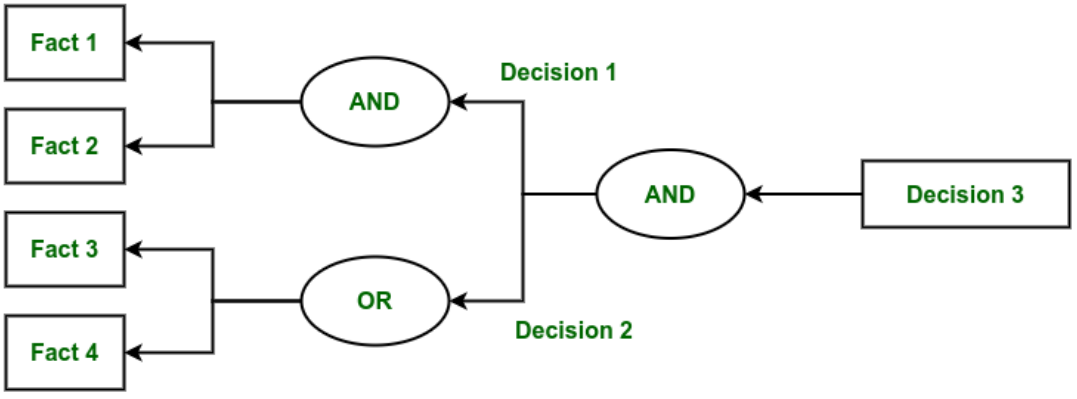


Figure 41 - Backward Chaining

2) Solved Lab Activites:

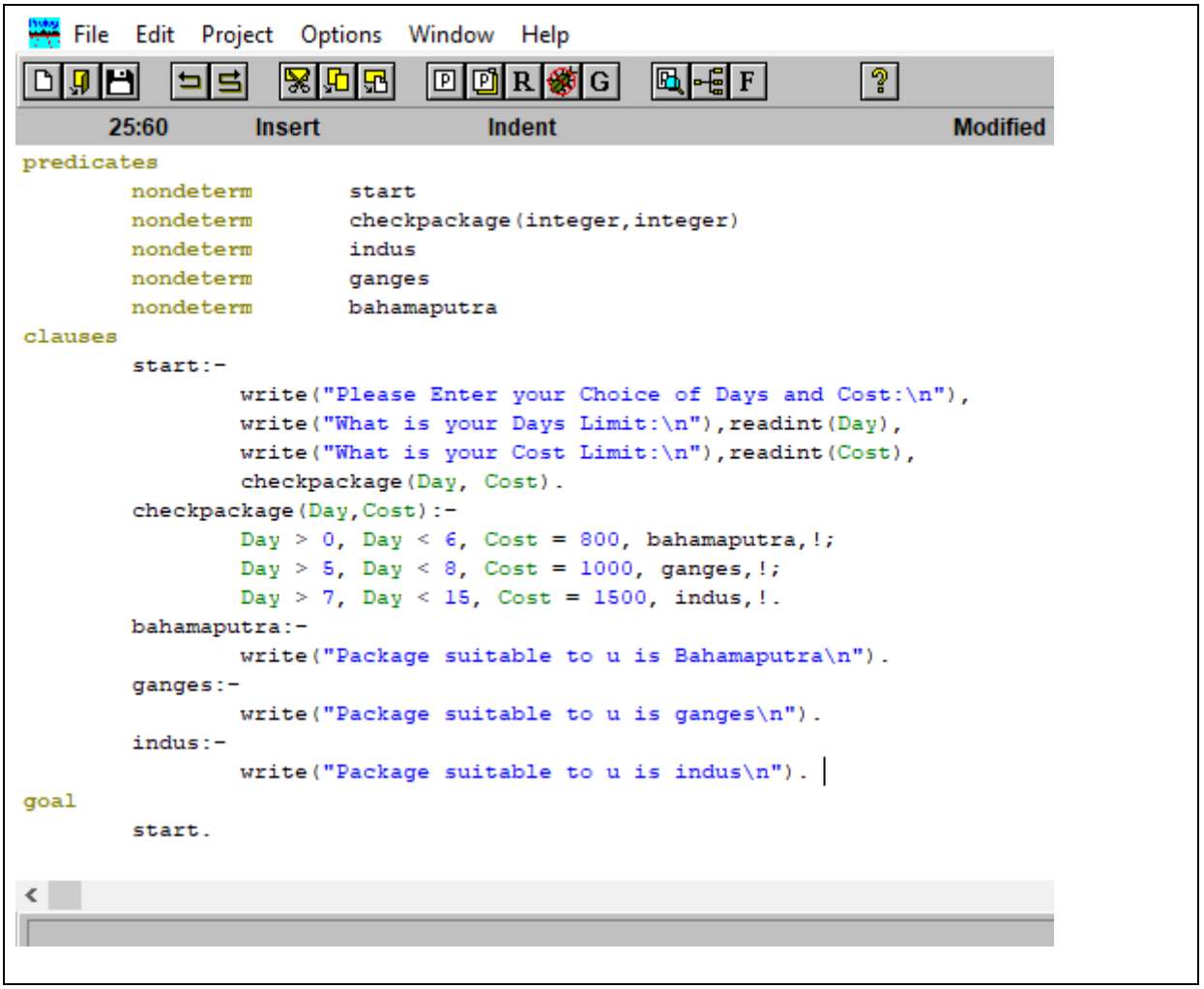
Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*45*	*High*	*CLO-6*

Activity 1:

Utilizing what have learned in the previous labs, develop a working expert system for tourists based on numbers of holidays and cost:

For the coming New Year holidays a company is offering three tourist packages for visiting Pakistan, India and BanglaDesh. They are called Indus, Ganges and Brahamaputra. The Indus package is for 7 days and costs $1000, the Ganges package is for 14 days and costs $ 1500 and the Brahamaputra package is for 5 days and costs $800.
Your implementation should have the following goal: start.
This goal, when run should ask for the budget and the time at the disposal of the client and then suggest a package based on the time and financial considerations.
Use the predicates write() and readint()

Solution:



The write("<string>") predicate displays the text on screen. Using the variables Day and Cost, the package is suggested.

3) Graded Lab Tasks:

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1

Utilizing what have learned in the previous labs, develop a working expert system for determining the horoscope based on date and month of birth:
Based on the date of birth and month of birth determine the zodiac sign, giving its basic information.
Your implementation should have the following goal:
start.
This goal, when run should ask for the date of month and the month of the year for example if your birthday is 1st February 1900, then as Date of month you should enter "1" and as month of the year value should be "2". Based on this information inform the zodiac sign, that is on of the Aries, Taurus, Gemini, etc out of the 12 zodiac signs.
Use the predicates write() and readint()