CS351L Introduction to Artificial Intelligence Lab

Faculty of Computer Science and Engineering

Lab manual

CS351L – Introduction to Artificial Intelligence Lab Manual

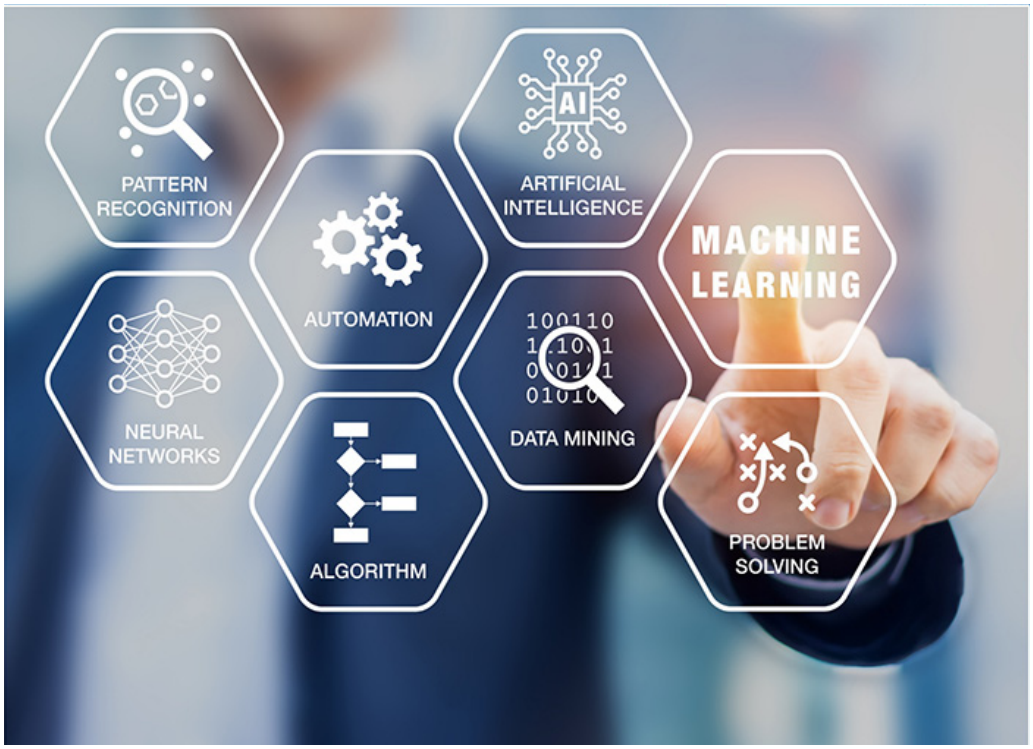
Faculty of Computer Science & Engineering (FCSE)

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology (GIKI)

PREPARED BY DR. SAJID, DR. HASHIM & MR. ARSALAN

CS351-L Introduction to Artificial Intelligence Lab

Lab Outline



Dr. Sajid Anwar, Dr. Hashim Ali & Mr. Arsalan
FACULTY OF COMPUTER SCIENCE AND ENGINEERING (FCSE)
GHULAM ISHAQ KHAN INSTITUTE OF ENGINEERING SCIENCES AND TECHNOLOGY (GIKI)

CS351L Artificial Intelligence Lab (1 CH)	CS

Pre-Requisite: CS221

Instructor: **Mr. Usman Haider**
Office # F-11 FES, GIK Institute,
Email: *usman.haider@giki.edu.pk*
Office Hours: 09:00am ~ 12:30 am (Monday-Tuesday) and 02:00 pm ~ 04:30 pm (Thursday-Friday)

Course Introduction

This course introduces strategies, methods and algorithms for solving problems that requires decision making on part of the computer. Further, different implementation techniques in automated reasoning, data and knowledge representation, search techniques, planning and learning techniques are practiced.

Course Contents

Broadly, this lab will cover following contents:

1- Introduction to Matlab and python from artificial intelligence perspective.
2- Introduction to Prolog – matching and proof searching, and recursive definition in Prolog.
3- Introduction to unsupervised learning – implementation of depth-first, breadth-first and other related approaches for specific problems.
4- Introduction to adversarial search problems – Formulation, application and implementation of minimax algorithm and alpha-beta pruning.
5- Introduction to supervised learning – implementation of neural networks in Matlab with and without NNT, and convolutional neural networks.
6- Introduction to optimization algorithms – understanding and applying local hill climbing, simulated annealing and genetic algorithms for optimization problems.
7- Introduction to unsupervised learning – Hierarchical clustering, K-means algorithm and other clustering techniques.
8- Handling uncertainty in systems – Fuzzy logic and Fuzzy systems.
9- Invited sessions on Tensor Flow

Mapping of CLOs and PLOs

Sr. No	Lab Learning Outcomes	PLOs	Blooms Taxonomy (Psychomotor domain)
CLO 1	Apply the fundamentals and basic concepts of reasoning and knowledge representation (e.g., in Prolog)	PLO 1	P3 (Guided Response)
CLO 2	Apply concepts of learning and implement it using Matlab/Python	PLO 3	P3 (Guided Response)
	[+]Please add the prefix "Upon successful completion of this lab, the student will be able to" [*]PLOs are for BS (CE) only		

CLO Assessment Mechanism

Assessment tools	CLO_1	CLO_2
Labs	100%	50%
Midterm Exam	-	-
Final Exam	-	50%

Overall Grading Policy

Assessment Items	Percentage
Lab Tasks	30%
Midterm Exam	25%
Final Exam	45%

Text and Reference Books

- Lab Manual
- Online prolog, python and Matlab documentation
- Course handouts and textbook.

Administrative Instruction

- According to institute policy, 80% attendance is mandatory to appear in the final examination.
- Attendance is mandatory for students in all the labs. If a student is absent from a lab due to any reason, he/she will have to get written permission of the Dean to perform that lab. The Dean will allow students to perform lab if he feels that the student has a genuine excuse.
- Students should bring their text books to the lab, so that they can refer to theory and diagrams whenever required. `
- Labs will be graded in double entry fashion; one entry in the assessment sheet given at the end of every lab and another entry in the instructor's record. This system of keeping records will keep students aware of their performance throughout the lab.
- For queries, kindly follow the office hours in order to avoid any inconvenience.

Computer Usage/Software Tools

- Matlab, Python, and Prolog

Lecture Breakdown

Lab 1	• Introduction to MATLAB. • Introduction to Python and selected libraries in Python.
Lab 2	• Introduction to Prolog – Basic concepts, Structures of Prolog programs, syntax, facts, rules and queries, Recursive definition, and Clause ordering & Goal ordering in prolog.
Lab 3	• Uninformed search techniques – Formulation, identification and solution implementation using Breadth-First, Depth-First & other similar search approaches.
Lab 4	• Informed search techniques – Formulation, identification and solution implementation using Greedy and A* search approach.
Lab 5	• Adversarial search strategies – Basics, Applicability, Implementation of algorithms & techniques (minimax algorithm & alpha-beta pruning).
Lab 6	• Optimization problems and local search algorithms – local hill climbing and simulated annealing.
Lab 7	• Optimization problems and bio-inspired search algorithms – Genetic algorithms.
Lab 8	• Supervised learning I – Neural Networks implementation in Matlab with and without neural network toolbox (NNT).
Lab 9	• Supervised Learning II – Introduction to Convolutional Neural Networks (CNN).
Lab 10	• Unsupervised Learning I – Hierarchical clustering libraries and implementation in MATLAB/Python.
Lab 11	• Unsupervised Learning II – Other clustering algorithms like K-means.
Lab 12	• Handling Uncertainty – Formulation and implementation of solutions for Fuzzy systems using Fuzzy logic.
Lab 13	• Open Ended Lab

"Artificial Intelligence is the science of making machines do things that would require intelligence if done by men"

Marvin Minsky

CS351-L Artificial Intelligence Lab

Lab Benchmark Report

Dr. Sajid Anwar, Dr. Hashim Ali & Mr. Arsalan

FACULTY OF COMPUTER SCIENCE AND ENGINEERING (FCSE)GHULAM ISHAQ KHAN INSTITUTEO F ENGINEERING SCIENCES AND TECHNOLOGY (GIKI)

The following were consulted to benchmark CS351-L:

Higher Education Commission (HEC) curriculum for Computer Science (Revised 2017)

In the curriculum defined by HEC, Artificial Intelligence is a Computer Science core course. However, there is no information about separate lab courses in the curriculum.
At FCSE, the contents of CS351L cover the entire hands-on aspects of the course contents as laid out in the curriculum manual (summarized in the table below).

Association of Computing Machinery (ACM) curriculum for CS (2013)

As per ACM Curriculum for CS (2013), Compiler Construction is categorized under the knowledge area of Intelligent Systems, as an elective course. There is no separate information for lab courses.
At FCSE, the contents of CS351-L caters to the bulk of the body of knowledge coverage as laid out in the ACM curriculum (summarized in the table below).

Curriculum Benchmarking for CS351-L Artificial Intelligence Lab				
Topic#	**Topic**	**FCSE, GIKI**	**HEC Curriculum for CS (Revised 2017)**	**ACM curriculum for CS (2013)**
1	Introduction to Artificial Intelligence	✓	✓	✓
2	Reasoning and knowledge representation	✓	✓	✓
3	Problem solving by searching	✓	✓	✓
4	Informed search	✓	✓	✓
5	Uninformed search	✓	✓	✓
6	Constraint satisfaction problems	✓	✓	✓
7	Adversarial search	✓	✓	✓
8	Learning	✓	✓	✓
9	Supervised learning	✓	✓	✓
10	Unsupervised learning	✓	✓	✓
11	Reinforcement learning	✗	✓	✓
12	Handling uncertainty in AI	✓	✓	✓
13	Recent trends in AI & Applications of AI algorithms	✓	✓	✓
14	Advanced knowledge representation	✓	✗	✓
15	Decision trees	✗	✗	✓
16	Local searching	✓	✗	✓
17	Naive Bayes	✗	✗	✓
18	Precision and accuracy	✗	✗	✓
19	Cross-fold validation	✗	✗	✓

CS351-L Intro to Artificial Intelligence Lab

LAB EVALUATION RUBRICS

Dr. Sajid Anwar, Dr. Hashim Ali, & Mr. Arsalan
FACULTY OF COMPUTER SCIENCE AND ENGINEERING (FCSE)
GHULAM ISHAQ KHAN INSTITUTE OF ENGINEERING SCIENCES & TECHNOLOGY (GIKI)

Faculty of Computer Science & Engineering

CS351L – Introduction to Artificial Intelligence Lab

Rubrics for Assessment of Lab

Rubrics for Assessment of CS351-L Introduction to Artificial Intelligence Lab				
Criteria	**Total Marks**	**Excellent (2)**	**Average (1)**	**Poor (0)**
Accuracy	2	The solution does exactly what has been asked in the task including for tricky cases and gives the accurate/expected results for all parts.	The solution does exactly as asked in the task but does not handle exceptions/tricky input.	The solution is poor, does not perform at all according to the task or gives absolutely wrong results even for standard input.
Clarity	2	The solution is clear, and the student is able to formulate and explain the solution coherently.	The solution is partially clear and the student can explain parts of the program	The student has not formulated the solution correctly or can not explain his/her solution at all.
Code	2	The code is coherently written and documented to the point that the reader can broadly understand the working of each major part.	The code is somewhat coherent, tabbing has not been used but has useful chunks written with broad documentation.	The code lacks readability, can not be explained without the coder and has little/no documentation.
Performance	2	The code is extremely fast and for standard medium sized input performs its processing quickly.	The code is somewhat fast, runs quickly for small inputs but may take long time for medium sized/large inputs.	The code is extremely slow and gets stuck even for small-sized inputs.
Completion Time	2	The final complete solution for all tasks was submitted before half of the class	The final complete solution for all tasks was submitted later than half of the class but before 80% of the size of class.	The final complete solution was submitted along with the last 20% of the class
Total	10	10	5	0

Topics & Content summary and weekly allocation for CS351-L

CS351-L – Introduction to Artificial Intelligence Lab

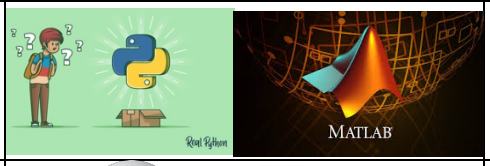
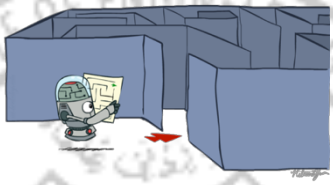
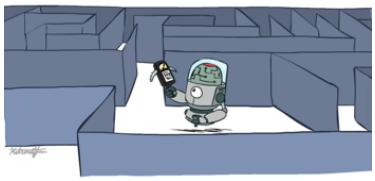
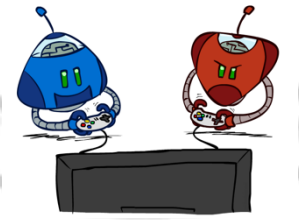
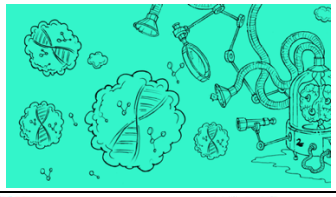
Dr. Sajid, Dr. Hashim & Mr. Arsalan

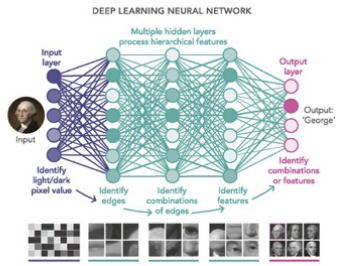
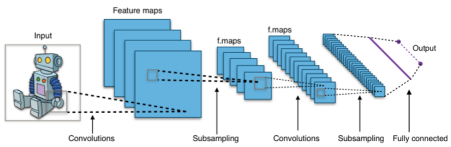
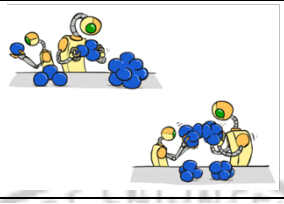
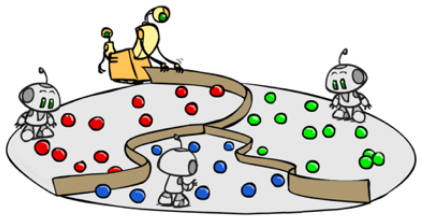
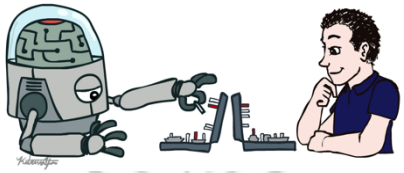
Ghulam Ishaq Khan Institute of Engineering Sciences

Faculty of Computer Science & Engineering

CS351L – Introduction to Artificial Intelligence Lab

Topics Summary and Weekly Allocation for Lab Manual

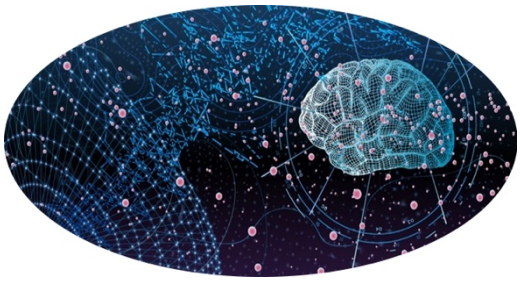
Week #	Topic	Pictorial Content	Contents	Page #
1	Introduction to programs and languages used in Artificial Intelligence		Introduction to MATLAB and Python	1
2	Reasoning and Knowledge Representation		Predicate Logic search and analysis in PROLOG	17
3	Problem-solving by searching (Uninformed search)		DFS and BFS implementation with applications and case studies in Python	37
4	Problem-solving by searching (Informed search)		A* algorithm implementation	43
5	Problem-solving by searching (Adversarial search)		Implementation of Minimax and Alpha-beta pruning	53
6	Optimization techniques and local search algorithms - I		Formulation and implementation of local hill climbing and simulated annealing	61
7	Optimization techniques and local search algorithms - II		Formulation and implementation of Genetic Algorithms	68
8	Mid-term exams		-	-

9	Supervised learning - I		Introduction to deep neural networks and implementation with and without using toolbox	77
10	Supervised learning - II		Introduction to Convolutional Neural Networks	88
11	Unsupervised learning - I		Hierarchical clustering in MATLAB/Python	
12	Unsupervised learning – II		Implementation of K-Means algorithm and other clustering methods	
13	Handling uncertainty		Formulation and implementation of solutions for Fuzzy system using Fuzzy logic	
14	Open ended lab		-	
15	Lab finals		-	

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology (GIKI)

Faculty of Computer Science & Engineering (FCSE)

CS351L – Introduction to Artificial Intelligence Lab Manual



"Predicting the future isn't magic, It's artificial intelligence!"

Dave Waters

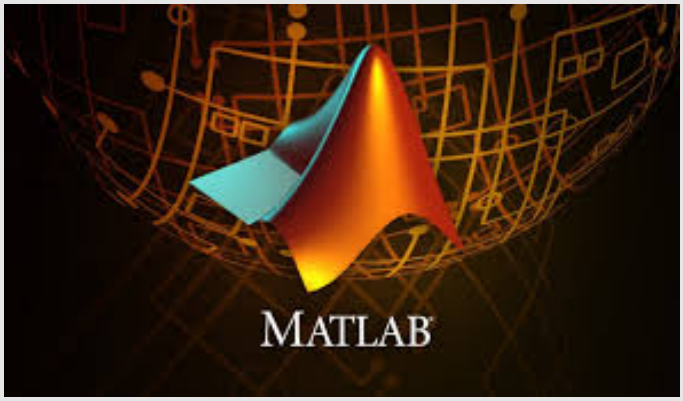
"By far the greatest danger of Artificial Intelligence is that people conclude too early that they understand it"

Eliezer Yudkowsky

PREPARED BY DR. SAJID, DR. HASHIM & MR. ARSALAN

CS351-L – Introduction to Artificial Intelligence Lab 1

INTRODUCTION TO MATLAB & PYTHON



Contents: Dr. Hashim Ali

Ghulam Ishaq Khan Institute of Engineering Sciences

Faculty of Computer Science & Engineering

CS351L – Introduction to Artificial Intelligence Lab

Lab 1 – Introduction to Important Tools in AI

Intro to Important Programming Languages & Tools in AI

Objective (Part I)

The objective of first part of this session is to get exposure to MATLAB and use some simple operations to access data and plot it using some key MATLAB commands.

Learning outcomes (Part I)

I. Perform basic mathematical operations on simple variables, vectors, matrices and complex numbers.
II. Generate 2-D plots.
III. Use and write script files (MATLAB programs). MATLAB script files have a file extension name .m and are, therefore, usually referred as M-files.
IV. Use the 'help' and 'lookfor' commands to debug codes.

Instructions

- Read through the handout sitting in front of a computer that has a PROLOG software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering problems, indicate the commands you entered and the output displayed.
- You must open a word document. After each command, copy & paste commands outputs.

Table of Contents

1.1. MATLAB

The name MATLAB stands for MATrix LABoratory. MATLAB was written originally to provide easy access to matrix software developed by the LINPACK (linear system package) and EISPACK (Eigen system package) projects. MATLAB is a high-performance language for technical computing. It integrates computation, visualization, and programming environment. Furthermore, MATLAB is a modern programming language environment: it has sophisticated data structures, contains built-in editing and debugging tools, and supports object-oriented programming. These factors make MATLAB an excellent tool for teaching and research. MATLAB has many advantages compared to conventional computer languages (e.g., C, FORTRAN) for solving technical problems. MATLAB is an interactive system whose basic data element is an array that does not require dimensioning. The software package has been commercially available since 1984 and is now considered as a standard tool at most universities and industries worldwide. It has powerful built-in routines that enable a very wide variety of computations. It also has easy to use graphics commands that make the visualization of results immediately available. Specific applications are collected in packages referred to as toolbox. There are toolboxes for signal processing, symbolic computation, control theory, simulation, optimization, and several other fields of applied science and engineering.

1.1.1. Starting MATLAB

Start MATLAB by clicking on it in the Start menu.
Once the program is running, you will see a screen similar to Figure 1.1.

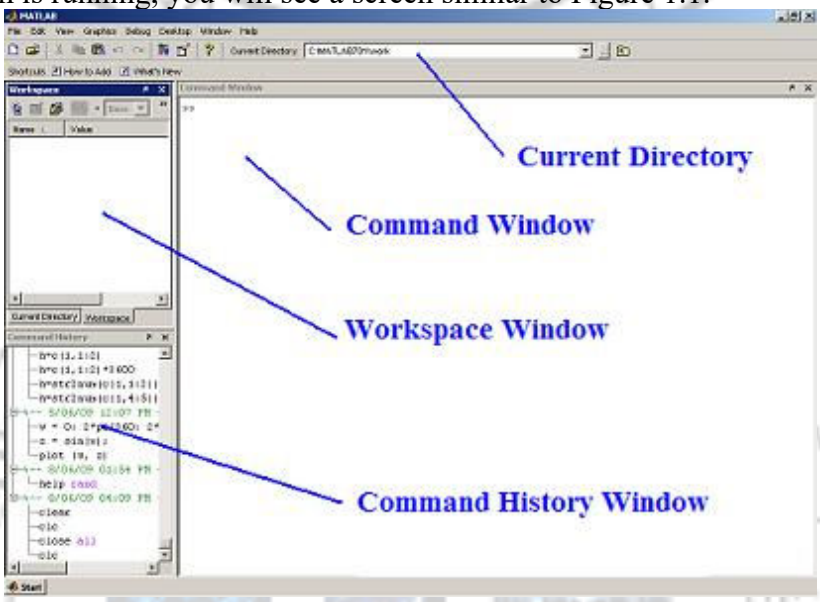


Figure 1.1: The MATLAB Graphical User Interface (GUI) (Image taken from http://www.matrixlab-examples.com/using-MATLAB.html)

The Command Window: is where you type commands. Hit Enter to run the command you just typed.

The Current Directory: shows the directory that you are working in. This directory is where MATLAB expects to find your files (M-files, audio, images, etc). If you encounter a 'file not found' error, it means the file is not in your Current Directory. You can change the working directory by typing into it or clicking through the file browser.

The Workspace Window: displays information about all the variables that are currently active. In particular, it will tell you the type (int, double, etc) of variable, as well as the dimensions of the data structure (such as a 2x2 or 8000x1 matrix). This information can be extremely useful for debugging!

The Command History Window: keeps track of the operations that you've performed recently. This is handy for keeping track of what you have or haven't already tried.

1.1.2. MATLAB Commands

help

MATLAB has two important help commands to find the right syntax and a description of a command with its all options.
Typing help by itself on the command line gives you a list of help topics.
If you want to find more about the syntax of a certain command, you type in

```
>> help function_name
```

where the word function_name is in the above entry is substituted by the actual name of a function for which description is sought. For example, we can type in

```
>> help plot
```

and MATLAB will display the description, syntax and options of the plot function.

lookfor

The other helpful command is lookfor. If we are not sure the exact name of the function, we can make a search for a string which is related to the function, and MATLAB displays all m-files (commands) that have a matching string. MATLAB searches for the string on the first comment line of the m-file. For example:

```
>>lookfor inverse
```

will display all m-files which contain the string 'inverse' in the comment line.

whos and clear

Two other useful commands are

```
>>whos
```

which lists all your active variables (this info also appears in your Workspace Window), and

```
>> clear
```

which clears and deletes all variables (for when you want to start over).
MATLAB has tab-completion for when you're typing long function names repeatedly. This means that you can type part of a command and then press <Tab> and it will fill in the rest of the command automatically. Moreover, MATLAB stores command history, and previous commands can be searched by pressing the up or down arrow keys.

1.1.3. Matrix Operations

MATLAB is designed to operate efficiently on matrices (hence the name MATLAB = "Matrix Laboratory). Consequently, MATLAB treats all variables as matrices, even scalars! Like many other programming languages, variables are defined in MATLAB by typing:

```
<VariableName> = <Assignment>
```

MATLAB will then acknowledge the assignment by repeating it back to you. The following are what you would see if you defined a scalar x, a vector y, and a matrix z:

```
>> x = 3
x =
        3
>> y = [1, 2, 3]
y =
        1       2       3
>> z = [1, 2, 3; 4, 5, 6]
z =
        1       2       3
        4       5       6
```

You can see from the above examples that scalar variables require no special syntax; you just type the value after the equals sign. Matrices are denoted by square brackets [ ]. Commas separate the elements within a row, and semicolons separate different rows. A row array, such as y, is just a special case of a matrix that has only one row.

The size command

The size command is extremely useful. This command tells you the dimensions of the matrix that MATLAB is using to represent the variable. To determine the dimension of a matrix x, for example, you type in

```
>> size(x)
```

You can also just use whitespace to separate elements within a row. The following two ways to define the variable are equivalent:

```
>> y = [1, 2, 3]
y =
        1       2       3
>> y2 = [1 2 3]
y2 =
        1       2       3
```

You now know how to define matrices. The ( ) operator allows you to access the contents of a matrix. MATLAB is 1-indexed, meaning that the first element of each array has index 1 (indexing starts from 1 not from 0).

```
>> y = [1, 2, 3];
>> y(1)
ans =
        1
```

To access a single element in a multidimensional matrix, use (i,j). The syntax is matrix(row,column):

```
>> y= [1, 2; 3, 4];
>> y(2, 1)
ans =
        3
```

There is a quick way to define arrays that are a range of numbers, by using the **:** operator.

The plot command

The MATLAB command plot allows you to graphically display vector data in the form of (surprise!) a plot. Most of the time, you'll want to graph two signals and compare them. If you had a variable t for time, and x for a signal, then typing the command

```
>> plot(t,x)
```

will display a plot of the signal x against time. See help plot if you haven't done so already. You MUST label your plots in order to communicate clearly. Your graphs must be able to tell a story without you being present! Here are a few useful annotation commands:

```
>> title('Here is a title'); %– Adds the text "Here is a title" to the top
of the plot.
>> xlabel('Distance traveled (m)'); %– Adds text to the X axis.
>> ylabel('Distance from Origin (m)'); % – Adds text to the Y axis.
>> grid on; %– Adds a grid to the plot.
>> grid off; %– Removes the grid (sometimes the plot is too cluttered with
it on).
>> hold on; %– Draws the next plot on top of the current one (on the same
axes). Useful for comparing plots.
>> hold off; %– Erases the current plot before drawing the next (this is the
default).
```

In order to display multiple plots in one window, you must use the subplot command.
This command takes three arguments as follows: subplot(m,n,p). The first two arguments break the window into an m by n grid of smaller graphs, and the third argument p selects which of those smaller graphs is being drawn right now.
For example, if you had three signals x, y, and z, and you wanted to plot each against time t, then we could use the subplot command to produce a single window with three graphs stacked vertically:

```
>> subplot(3,1,1);
>> plot(t,x);
>> subplot(3,1,2);
>> plot(t,y);
>> subplot(3,1,3);
>> plot(t,z);
```

See help subplot or look online for more examples.
MATLAB only handles discrete representations of signals. As such, you need a way to represent time for continuous signals in MATLAB. All functions of time in MATLAB must be defined over a particular range of time (and with a particular granularity or increment). Note that the granularity of your time vector will impact the resolution of your plots.

1.2. Python

Objective (Part II)

The objective of this session is to get exposure to Python programming within PyScripter environment, and write some simple code to access data and plot it using some key Python libraries.

Learning outcomes (Part II)

1) Write simple Python code inside PyScripter to create random numbers and frequency histogram using the numpy library.
2) Create graphs in Python using the matplotlib library.
3) Understand and work with numeric data in the library pandas.

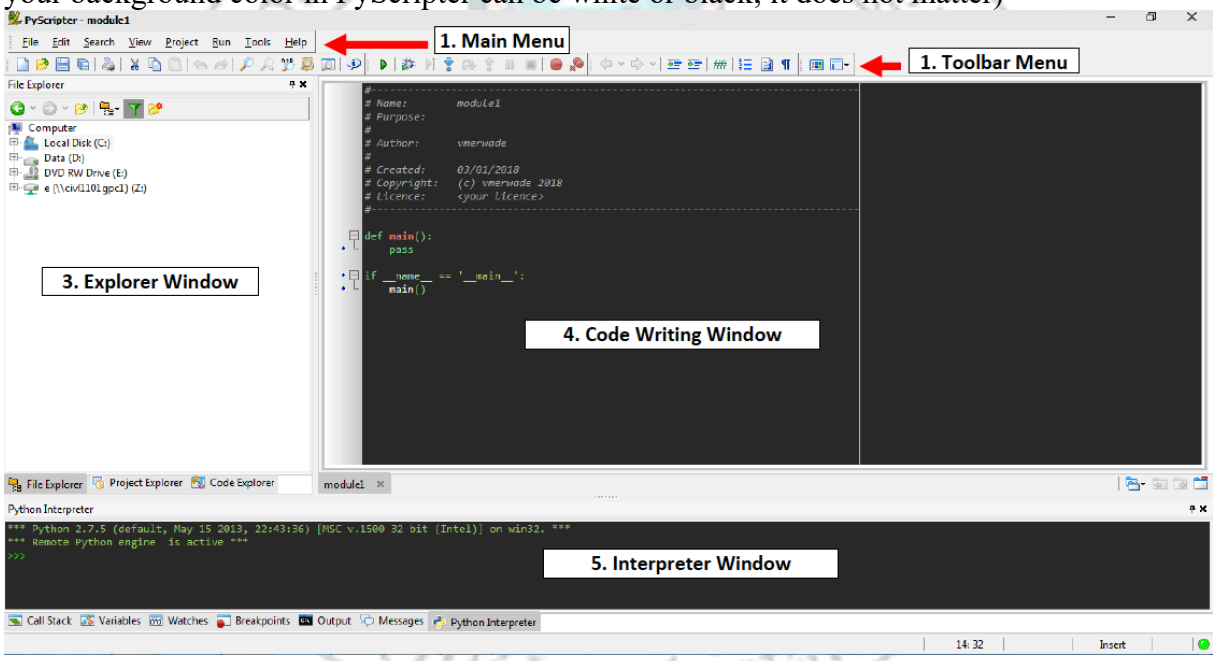
1.2.1. Introduction

When it comes to programming, adopting a language depends on what you want to accomplish. For example, if you want to write a code to solve complex numerical equations, you may use C++ or Fortran. Similarly, if you are interested in statistics, R may be a good choice. Accordingly, when it comes to implementation of pattern recognition or quick scripting for artificial intelligence problems, python is widely used as it is a higher level language that is open source, cross-platform, and is easy to learn and code. Additionally, standard libraries for clustering, optimization, and classification are available for direct use in Python. You can find more about python at http://www.python.org/doc/ and http://www.diveintopython.org.

Now, you need an "environment" to write your Python code. There are many other independent softwares, such as IDLE, Spider, Sublime, and PyScripter, to write and run a Python code. In this class, we are going to use PyScripter as this is one of the most famous python IDE available.

Getting PyScripter (Purdue students can skip the download step and just open the program from the Start menu). **Download** PyScripter from https://sourceforge.net/projects/pyscripter and install it on your computer.

Start/open PyScripter by double clicking the icon or from the start menu. The PyScripter window, shown on the next page, allows you to write the Python code, run the program, and look at the results. The PyScripter window has four sections as shown in the Figure 1.2. (Note: your background color in PyScripter can be white or black; it does not matter)



Pyscripter has five main sections:

1) Main Menu allows you to create a new file, print, zoom in or out and search for help.

2) Toolbar Menu has several Common tools to run the code, search, cut/paste, etc.
3) The Explorer allows you to explore files similar to windows explorer, explore your current python project, or explore your code to see the name of the libraries, subroutines, etc.
4) Code writing window is where you will write your code to create your .py file, modify it, save it and run it.
5) Python interpreter is where you will see the results (more often errors!) after running the code.

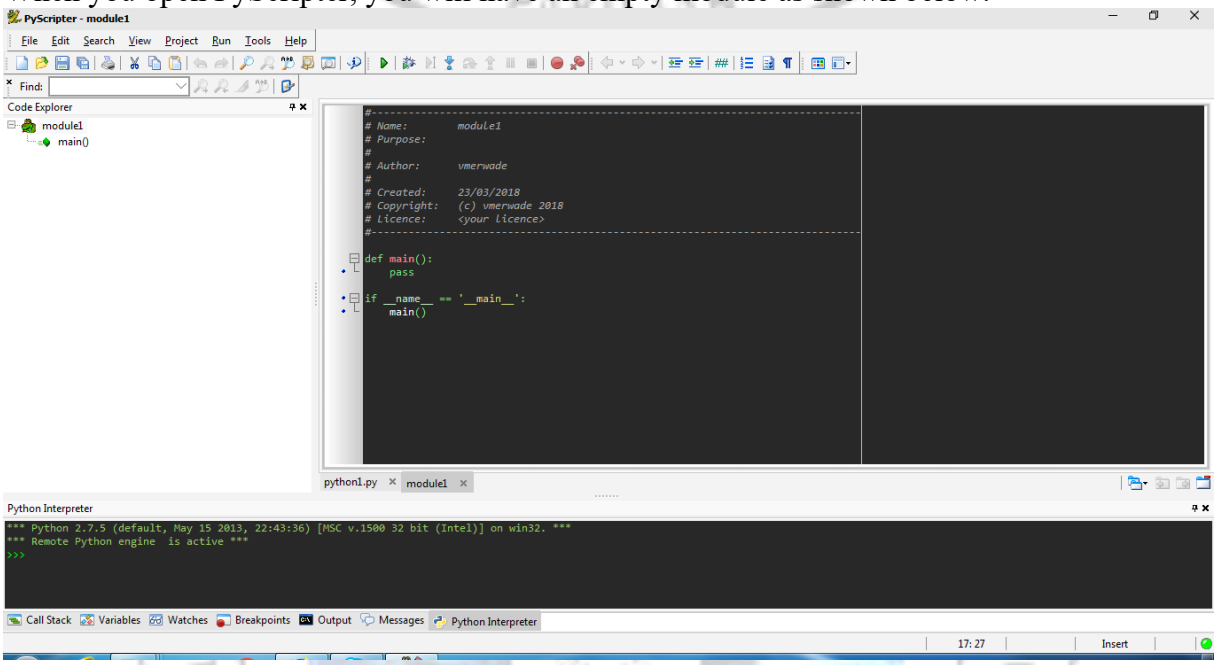
1.2.2 Python Libraries

A library is basically a collection of standard code or sub-routines that are frequently used in programming. A library helps a coder (you) to accomplish a programming task in a few lines of code by borrowing all the lines stored in a function within a library. One of the greatest strengths of Python is its large library that provide access to tools for numerical computations as well as GIS applications. Some common libraries that we will be using in this class are briefly introduced below, along with a sample code that you can try in your PyScripter.

1.2.2.1. NumPy

If you are familiar with Matlab, numpy provides Matlab like functionality within Python. Using NumPy you can solve multi-dimensional arrays and matrices, and perform simple to complex mathematical operations including statistics. Now, lets see how to use numpy in PyScripter to generate 1000 random numbers from a normal distribution with mean = 100 and standard deviation = 15.

When you open PyScripter, you will have an empty module as shown below.



If you do not see an empty module, you can create a new file by using the Main menu. Click on File → New →New Python Module. Save this module as "python1" inside a folder called M1.

The initial lines that start with # are just comments, and are not considered as code. You can write any information about your script by using # before any text/comment. This is our first code so for now, we will just leave the default text unchanged. Delete everything after the last line with #. Specifically, the line that starts with def main(), and everything after that.

Write the first line of code as below. (Note: All code in this tutorial will be highlighted to distinguish it from the regular text)

```
import numpy as np
```

Using this single statement, we have imported or borrowed the *numpy* library and referenced it as np. This is how we will import any library. You can use any name to reference a library. In this case we used np as it is an abbreviation of numpy. You can use nmy, npy or anything. You get the point! Use something logical!

Now we are going to define two variable for the mean and standard deviation, and assign them the value of 100 and 15, respectively. Again, you can pick any name for defining a variable, but it is a good practice to use something that is self-explanatory. We use greek letters and for mean and standard deviation, respectively. Lets write our second line of code to define these two variables as below.

```
mu, sigma = 100, 15
```

Here we defined both variables in a single statement. We could also do mu = 100 on one line and sigma = 15 on another line. Once we have the mean and standard deviation, we can then generate values using the following statement.

```
x = mu + sigma*np.random.randn(1000)
```

The randn function above generates random numbers from a standard normal distribution (mean = 0 and variance = 1), which is then scaled to our mu and sigma, and stored in variable *x*. Basically, *x* is an array to store 1000 random values from the specified normal distribution.

Q. What is an array?

Now run the code by clicking on the green triangle button in the toolbar (ctrl + F9). In the lower left corner, you will see “Running” and then "Ready" or “Script run OK”. This means the code ran successfully, and you can now see the results.

We defined an array named x to store our random values. You can see the results by calling this x array in Python Interpreter window. Simply type x in the Python interpreter window, and press *Enter*. You should then see all the random values as shown below.

```
>>> x
array([  99.43481397,  119.01639987,   91.37194674, ...,   94.69074833,
         86.805851  ,   92.4282399 ])
>>>
```

Congratulations! You just finished your first code in Python! Lets experiment with another library in Python.

1.2.2.2. Matplotlib

matplotlib is a plotting library for the Python programming language and it can be used to make plots in PyScripter. Lets use this library to create a histogram from the x array we just created. Import the pyplot sub-library from matplotlib and name it as plt. Write the following code in your pyscripter window as shown below. (Note: you can either write this statement after your last statement, but it is a good practice to have all imports on the top so you know what libraries are available in the code.)

```
import matplotlib.pyplot as plt
```

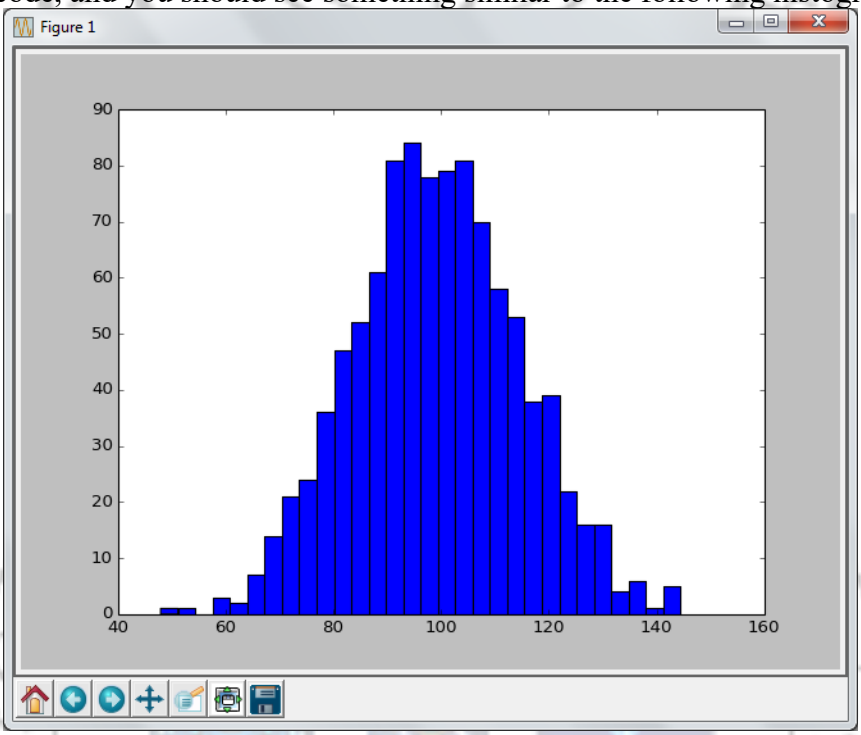
Now lets create the histogram of x by using plt. Write the following code in your pyscripter window. This will generate the histogram by dividing the array into 30 bins. Bins size is a parameter so you can choose some other value if you wish.

```
plt.hist(x,bins=30)
```

After creating the histogram, lets plot the histogram by using the following code.

```
plt.show()
```

Now run the code, and you should see something similar to the following histogram.



If you want, you can play around with the bin size to see how the plot changes.

Lets use pyplot to create scatter plot between two random variables. Define two random variables *a* and *b* as shown below. Remember we used *randn* earlier to create random numbers from a standard normal distribution. The random function will generate purely random numbers. The number inside the parenthesis defines the size.

```
a = np.random.random(100)
b = np.random.random(100)
```

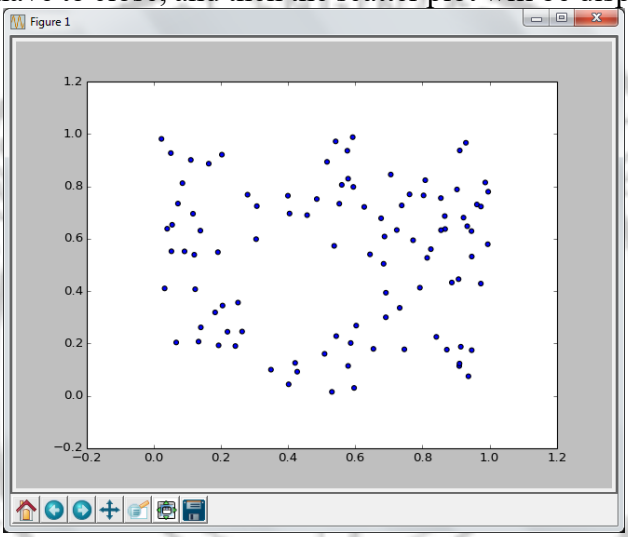
Now plot a and b on a scatter plot by using the code below. After you type plt, you will see a list of associated functions so pick scatter and provide a and b as two variables as shown below.

```
plt.scatter(a,b)
```

Then show the plot.

```
plt.show()
```

You are ready to run the code at this point. After you run the code, you will first see the histogram, which you have to close, and then the scatter plot will be displayed as shown below.



How can you avoid the histogram to show and let the program only show the scatter plot? Think of #

1.2.2.3. Pandas

Python Data Analysis Library, or Pandas, is a Python package providing fast, flexible, and expressive data structures designed to make working with is "relational" or "labeled" data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python.

As usual, lets first import the *pandas* library as *pd* <all import statements will be at the beginning of your code>

```
import pandas as pd
```

Now read the csv file from your working directory and store it in data as shown below (note the forward slash in the directory name). *usecols* will store the data from each column in the respective array.

```
data = pd.read_csv('C:/temp/wabash.csv', usecols=['datetime', 'discharge', 'stage'])
```

The above statement reads the CSV file and stores the data in three arrays: time stamp in 'datetime', flow values in 'discharge', and stage in 'stage'. Because the datetime has both date and time in hh:ss format, it is stored as a string (or text) instead of date. Lets convert this array from string to date by using the statement below. This statement will convert the string to a datetime format array and stored it in timestamp series.

```
timestamp = pd.to_datetime(data.datetime)
```

Now we can plot information from data by using the statement below.

```
plt.plot(data.stage,data.discharge)
```

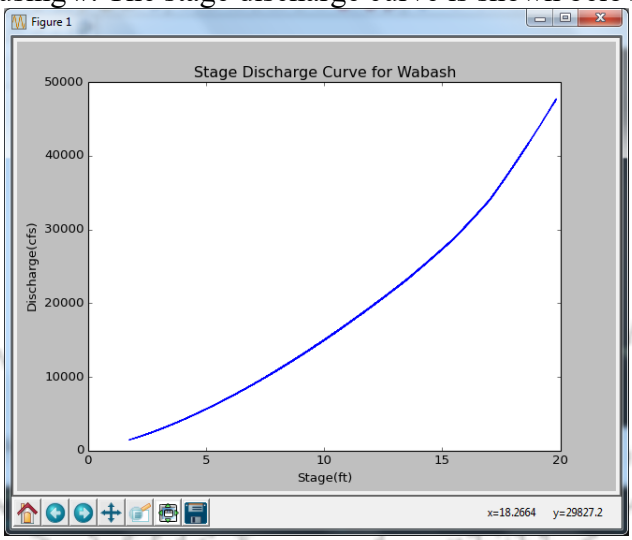
The scatter function we used earlier gave us points. The plot function will give us a line. Lets also define our x and y axes labels, and provide a title for the plot as shown below. The functions we are using here are self-explanatory.

```
plt.xlabel("Stage(ft)")
plt.ylabel("Discharge(cfs)")
plt.title("Stage Discharge Curve for Wabash")
```

Finally, show the plot by using the statement below.

```
plt.show()
```

Run your code. If you want you can tell the program to skip the earlier plots by commenting the related statements using #. The stage discharge curve is shown below.



Now write some python code on your own to create a discharge time series as shown below. Make sure it has axes and plot titles.

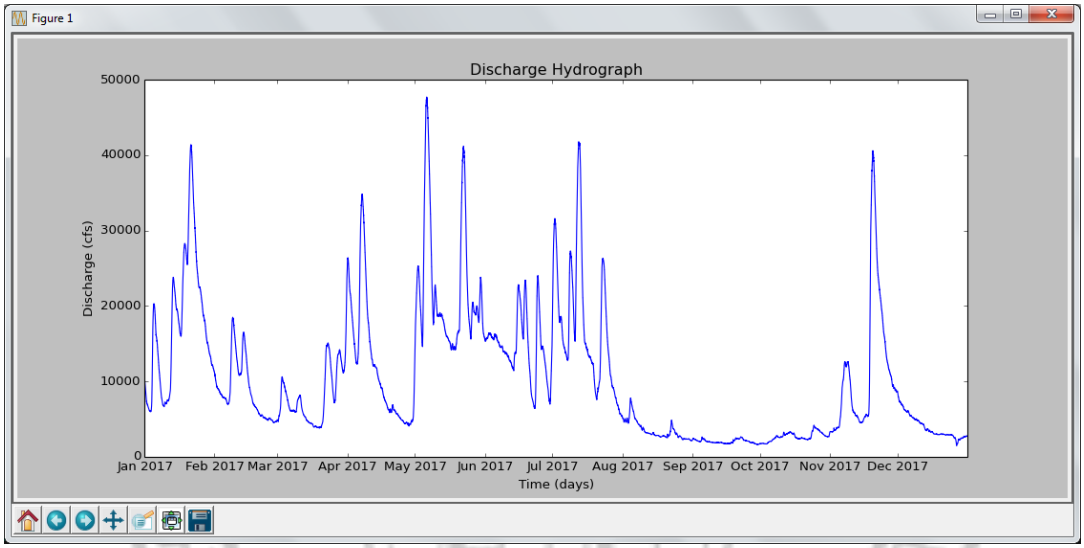


Figure 1
Discharge Hydrograph
Discharge (cfs)
50000
40000
30000
20000
10000
0
Jan 2017
Feb 2017
Mar 2017
Apr 2017
May 2017
Jun 2017
Jul 2017
Aug 2017
Sep 2017
Oct 2017
Nov 2017
Dec 2017
Time (days)

1.3. Exercises

Exercise 1.1.

Define the following column arrays in MATLAB:

$$a = \begin{pmatrix} 2 \\ 4 \end{pmatrix} \text{ and } b = \begin{pmatrix} 3 \\ 1 \end{pmatrix}$$

Then issue the following commands:

```
a'
a * b'
a .* b
3 .* b
```

What do each of these three operators do?

```
'       *       .*
```

Exercise 1.2.

Perform the following operation:

```
a * b
```

What is the error message? What does the message mean?
What is the correct fix?

Exercise 1.3.

Perform the following:

```
c = a + b
d = a + b;
```

What does the ; do?

Exercise 1.4.

Define `e = 17` in MATLAB. Use size to find the dimensions of `a`, `b'` and `e`?
There is an alternate syntax for size to determine length of a vector. Can you guess? Search for the command by using the help commands of MATLAB.
Use the alternate syntax for size to determine the height of `b`.

Exercise 1.5.

Define the following:

```
g = 0:25;
h = -10:10;
```

How big are `g` and `h`? Write the command to find their sizes.
What are the first, second, third, and last elements of each?
What exactly do `g` and `h` contain?
Create the following vector `k` as follows:

```
k = -10:0.1:10;
```

How big is `k`?
What are the first, second, third, and last elements?
What exactly does `k` contain?

Exercise 1.6.

Create and plot a signal $x_0(t) = te^{-|t|}$ using the following commands:

```
>> t = -10:0.1:10
>> x0 = t .* exp(-abs(t));
>> plot(t,x0)
```

Create the related signals $x_e(t) = |t|e^{-|t|}$ and $x(t) = 0.5 * [x_0(t) + x_e(t)]$.
What are $x_0(t)$ and $x_e(t)$, relative to $x(t)$?
Plot all three signals together in one window.

Exercise 1.7

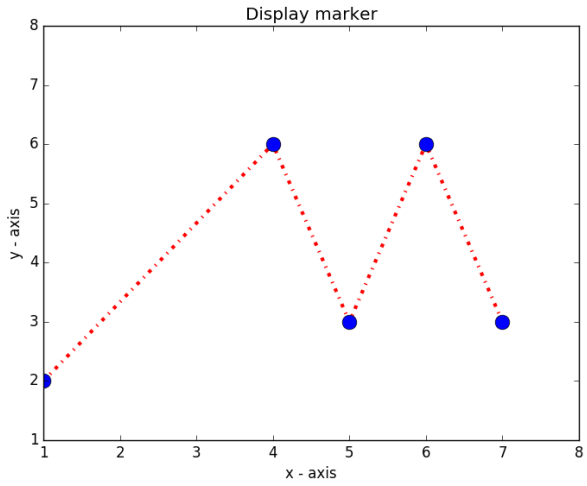
Write a NumPy program to create an array of all the even integers from 200 to 240.

Exercise 1.8

Write a NumPy program to generate an array of 10 random numbers from a standard normal distribution.

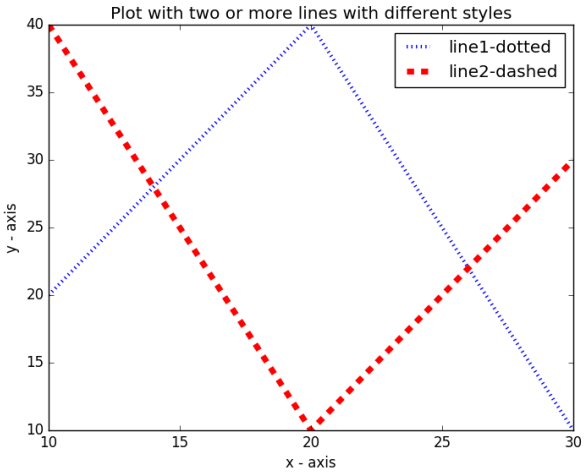
Exercise 1.9

Write a Python program to plot two or more lines and set the line markers. The code snippet gives the output shown in the following screenshot:



Exercise 1.10

Write a Python program to plot two or more lines with different styles. The code snippet gives the output shown in the following screenshot:



Exercise 1.11

Write a Python program to change the datatype of a given column or a Series

```
Sample Series:
Original Data Series:
0 100
1 200
2 python
3 300.12
4 400
dtype: object
```

Change the said datatype to numeric

```
0 100.00
1 200.00
2 NaN
3 300.12
4 400.00
dtype: float64
```

Exercise 1.12

Write a pandas program to delete DataFrame row(s) based on given column value.

```
Sample data:
Original DataFrame
col1 col2 col3
0 1 4 7
1 4 5 8
2 3 6 9
3 4 7 0
4 5 8 1
New DataFrame
col1 col2 col3
0 1 4 7
2 3 6 9
3 4 7 0
4 5 8 1
```

CS351-L – Introduction to Artificial Intelligence Lab 2

REASONING AND KNOWLEDGE REPRESENTATION

Contents: Dr. Hashim Ali

Ghulam Ishaq Khan Institute of Engineering Sciences

Faculty of Computer Science & Engineering

CS351L – Introduction to Artificial Intelligence Lab

Lab 2 – Reasoning and Knowledge Representation

Reasoning and Knowledge Representation

Objective

The objective of this session is to get exposure to PROLOG – a language for reasoning & knowledge representation, and use predicate logic and look up tables to infer information from simple statements.

Learning outcomes

1. To give some simple examples of Prolog programs that introduces you to three basic constructs in Prolog: facts, rules, and queries as well as to a number of other themes, like role of logic in Prolog, and the idea of performing unification with aid of variables.
2. To begin the systematic study of Prolog by defining terms, atoms, and variables.
3. To introduce recursive definitions in Prolog.

Instructions

- Read through the handout sitting in front of a computer that has a PROLOG software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering problems, indicate the commands you entered and the output displayed.
- You must open a word document. After each command, copy & paste commands outputs.

Table of Contents

2.0. What is PROLOG?

- Prolog = Programmation en Logique (Programming in Logic).
- Prolog is a declarative programming language
  unlike most common programming languages.
- In a declarative language
  - the programmer specifies a goal to be achieved
  - the Prolog system works out how to achieve it
- relational databases owe something to Prolog
- traditional programming languages are said to be **procedural**
- procedural programmer must specify in detail how to solve a problem:

  mix ingredients;
  beat until smooth;
  bake for 20 minutes in a moderate oven;
  remove tin from oven;
  put on bench;
  close oven;
  turn off oven;
- in purely declarative languages, the programmer only states what the problem is and leaves the rest to the language system
- We'll see specific, simple examples of cases where Prolog fits really well shortly.

Applications of PROLOG

Some applications of Prolog are:

- intelligent data base retrieval
- natural language understanding
- expert systems
- specification language
- machine learning
- robot planning
- automated reasoning
- problem solving

Relations

- Prolog programs specify *relationships* among objects and properties of objects.
- When we say, "John owns the book", we are declaring the ownership relationship between two objects: John and the book.
- When we ask, "Does John own the book?" we are trying to find out about a relationship.
- Relationships can also rules such as:

  Two people are sisters **if**
  they are both female **and**
  they have the same parents.
- A rule allows us to find out about a relationship even if the relationship isn't explicitly stated as a fact.

A little more on being sisters

- As usual in programming, you need to be a bit careful how you phrase things:
- The following would be better:

  A and B are sisters **if**
  A and B are both female **and**
  they have the same father **and**
  they have the same mother **and**
  A is not the same as B

Programming in prolog

- declare facts describing explicit relationships between objects and properties objects might have (e.g. Jameel likes pizza, grass has_colour green, Tiger is_a_dog, Chutki taught Jamila Swedish)
- define rules defining implicit relationships between objects (e.g. the sister rule above) and/or rules defining implicit object properties (e.g. X is a parent if there is a Y such that Y is a child of X).

One then uses the system by:

- asking questions above relationships between objects, and/or about object properties (e.g. does Jameel like pizza? is Jan a parent?)

2.1. Some Simple Examples

There are only three basic constructs in Prolog: facts, rules, and queries. A collection of facts and rules is called a knowledge base (or a database) and Prolog programming is all about writing knowledge bases. That is, Prolog programs simply are knowledge bases, collections of facts and rules which describe some collection of relationships that we find interesting.

So how do we use a Prolog program? By posing queries. That is, by asking questions about the information stored in the knowledge base.

Now this probably sounds rather strange. It's certainly not obvious that it has much to do with programming at all. After all, isn't programming all about telling a computer what to do? But as we shall see, the Prolog way of programming makes a lot of sense, at least for certain tasks; for example, it is useful in computational linguistics and Artificial Intelligence (AI). But instead of saying more about Prolog in general terms, let's jump right in and start writing some simple knowledge bases; this is not just the best way of learning Prolog, it's the only way.

Knowledge Base 1

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

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

This collection of facts is KB1. It is our first example of a Prolog program. Note that names `mia`, `jody`, and `yolanda`, the properties `woman` and `playsAirGuitar`, and proposition `party` have been written so that the first letter is in lower-case. This is important; we will see why a little later on.

How can we use KB1? By posing queries. That is, by asking questions about the information KB1 contains. Here are some examples. We can ask Prolog whether Mia is a woman by posing the query:

```
?-  woman(mia).
```

Prolog will answer

```
yes
```

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). The full stop is important. If you don't type it, Prolog won't start working on the query.

Similarly, we can ask whether Jody plays air guitar by posing the following query:

```
?-  playsAirGuitar(jody).
```

Prolog will again answer yes, because this is one of the facts in KB1. However, suppose we ask whether Mia plays air guitar:

```
?-  playsAirGuitar(mia).
```

We will get the answer

```
no
```

Why? Well, first of all, this is not a fact in KB1. Moreover, KB1 is extremely simple, and contains no other information (such as the rules we will learn about shortly) which might help Prolog try to infer (that is, deduce) whether Mia plays air guitar. So Prolog correctly concludes that `playsAirGuitar(mia)` does not follow from KB1.

Here are two important examples. First, suppose we pose the query:

```
?-  playsAirGuitar(vincent).
```

Again Prolog answers `no`. Why? Well, this query is about a person (`Vincent`) that it has no information about, so it (correctly) concludes that `playsAirGuitar(vincent)` cannot be deduced from the information in KB1.

Similarly, suppose we pose the query:

```
?-  tatooed(jody).
```

Again Prolog will answer no. Why? Well, this query is about a property (being tatooed) that it has no information about, so once again it (correctly) concludes that the query cannot be deduced from the information in KB1. (Actually, some Prolog implementations will respond to this query with an error message, telling you that the predicate or procedure `tatooed` is not defined; we will soon introduce the notion of predicates.)

Needless to say, we can also make queries concerning propositions. For example, if we pose the query

```
?-  party.
```

then Prolog will respond

```
yes
```

and if we pose the query

```
?-  rockConcert.
```

then Prolog will respond

```
no
```

exactly as we would expect.

Knowledge Base 2

Here is KB2, our second knowledge base:

```
happy(yolanda).
listens2Music(mia).
listens2Music(yolanda):-  happy(yolanda).
playsAirGuitar(mia):-  listens2Music(mia).
playsAirGuitar(yolanda):-  listens2Music(yolanda).
```

There are two facts in KB2, `listens2Music(mia)` and `happy(yolanda)`. The last three items it contains are rules.

Rules state information that is conditionally true of the situation of interest. For example, the first rule says that Yolanda listens to music if she is happy, and the last rule says that Yolanda plays air guitar if she listens to music. 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 called modus ponens.

Let's consider an example. Suppose we ask whether Mia plays air guitar:

```
?-  playsAirGuitar(mia).
```

Prolog will respond yes. Why? Well, although it can't find `playsAirGuitar(mia)` as a fact explicitly recorded in KB2, it can find the rule

```
playsAirGuitar(mia):-  listens2Music(mia).
```

Moreover, KB2 also contains the fact `listens2Music(mia)`. Hence Prolog can use the rule of 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, first of all, by using the fact `happy(yolanda)` and the rule

```
listens2Music(yolanda):-  happy(yolanda).
```

Prolog can deduce the new fact `listens2Music(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. In particular, from this inferred fact and the rule

```
playsAirGuitar(yolanda):-  listens2Music(yolanda).
```

it can deduce `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:

```
listens2Music
happy
playsAirGuitar
```

The `happy` predicate is defined using a single clause (a fact). The `listens2Music` and `playsAirGuitar` predicates are each defined using two clauses (in one case, two rules, and in the other case, one rule and one fact). 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.

Knowledge Base 3

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

```
happy(vincent).
listens2Music(butch).
playsAirGuitar(vincent):- listens2Music(vincent), happy(vincent).
```

```
playsAirGuitar(butch):- happy(butch).
playsAirGuitar(butch):- listens2Music(butch).
```

There are two facts, `happy(vincent)` and `listens2Music(butch)`, and three rules.

KB3 defines the same three predicates as KB2 (namely `happy`, `listens2Music` , 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):- listens2Music(vincent), happy(vincent).
```

has two items in its body, or (to use the standard terminology) two goals. So, what exactly does this rule mean? The most important thing to note is the comma, that separates the goal `listens2Music(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 `listens2Music(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), listens2Music(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):- listens2Music(butch).
```

This is a way of stating that Butch plays air guitar either if 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 `listens2Music(butch)` and this means Prolog can apply modus ponens using the rule

```
playsAirGuitar(butch):- listens2Music(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); listens2Music(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. Is it better to use multiple rules or the semicolon? That depends. On the one hand, extensive use of semicolon can make Prolog code hard to read. On the other hand, the semicolon is more efficient as Prolog only has to deal with one rule.

It should now be clear that Prolog has something to do with logic: after all, the :- means implication, the , means conjunction, and the ; means disjunction. Moreover, we have seen that a standard logical proof rule (modus ponens) plays an important role in Prolog programming. So we are already beginning to understand why "Prolog" is short for "Programming with logic".

Knowledge Base 4

Here is KB4, our fourth knowledge base:

```
woman(mia).
woman(jody).
woman(yolanda).
friends(vincent,mia).
friends(marsellus,mia).
friends(pumpkin,honey_bunny).
friends(honey_bunny,pumpkin).
```

Now, this is a pretty boring knowledge base. There are no rules, only a collection of facts. Ok, we are seeing a relation that has two names as arguments for the first time (namely the `friends` relation), but, let's face it, that's a rather predictable idea.

No, the novelty this time lies not in the knowledge base, it lies in the queries we are going to pose. In particular, for the first time we're going to make use of variables. Here's an example:

```
?-  woman(X).
```

The `X` is a variable (in fact, any word beginning with an upper-case letter is a Prolog variable, which is why we had to be careful to use lower-case initial letters in our earlier examples). Now a variable isn't a name, rather it's a placeholder for information. That is, this query asks Prolog: tell me which of the individuals you know about is a woman.

Prolog answers this query by working its way through KB4, from top to bottom, trying to unify (or match) the expression `woman(X)` with the information KB4 contains. Now the first item in the knowledge base is `woman(mia)`. So, Prolog unifies `X` with `mia`, thus making the query agree perfectly with this first item. (Incidentally, there's a lot of different terminology for this process: we can also say that Prolog instantiates `X` to `mia`, or that it binds `X` to `mia`.) Prolog then reports back to us as follows:

```
X = mia
```

That is, it not only says that there is information about at least one woman in KB4, it actually tells us who she is. It didn't just say `yes`, it actually gave us the variable binding (or variable instantiation) that led to success.

But that's not the end of the story. The whole point of variables is that they can stand for, or unify with, different things. And there is information about other women in the knowledge base. We can access this information by typing a semicolon:

```
X = mia  ;
```

Remember that ; means or , so this query means: are there any alternatives ? So Prolog begins working through the knowledge base again (it remembers where it got up to last time and starts from there) and sees that if it unifies `X` with `jody`, then the query agrees perfectly with the second entry in the knowledge base. So it responds:

```
X = mia  ;
X = jody
```

It's telling us that there is information about a second woman in KB4, and (once again) it actually gives us the value that led to success. And of course, if we press ; a second time, Prolog returns the answer

```
X  =  mia  ;
X  =  jody  ;
X  =  yolanda
```

But what happens if we press ; a third time? Prolog responds `no`. No other unifications are possible. There are no other facts starting with the symbol woman. The last four entries in the knowledge base concern the `friend` relation, and there is no way that such entries can be unified with a query of the form `woman(X)`.

Let's try a more complicated query, namely

```
?-  friends(marsellus,X),  woman(X).
```

Now, remember that , means and, so this query says: is there any individual `X` such that Marsellus is friends with `X` and `X` is a woman? If you look at the knowledge base you'll see that there is: Mia is a woman (fact 1) and Marsellus is friends with Mia (fact 5). And in fact, Prolog is capable of working this out. That is, it can search through the knowledge base and work out that if it unifies `X` with Mia, then both conjuncts of the query are satisfied. So Prolog returns the answer

```
X  =  mia
```

The business of unifying variables with information in the knowledge base is the heart of Prolog. As we'll learn, there are many interesting ideas in Prolog — but when you get right down to it, it's Prolog's ability to perform unification and return the values of the variable bindings to us that is crucial.

Knowledge Base 5

Well, we've introduced variables, but so far, we've only used them in queries. But variables not only can be used in knowledge bases, it's only when we start to do so that we can write truly interesting programs. Here's a simple example, the knowledge base KB5:

```
friends(vincent,mia).
friends(marsellus,mia).
friends(pumpkin,honey_bunny).
friends(honey_bunny,pumpkin).
jealous(X,Y):-  friends(X,Z),  friends(Y,Z).
```

KB5 contains four facts about the `friends` relation and one rule. But this rule is by far the most interesting one we have seen so far: it contains three variables (note that `X`, `Y`, and `Z` are all upper-case letters). What does it say?

In effect, it is defining a concept of jealousy. It says that an individual X will be jealous of an individual Y if there is some individual Z that X is friends with, and Y is friends with that same individual Z too. (Ok, so jealousy isn't as straightforward as this in the real world.) The key thing to note is that this is a general statement: it is not stated in terms of mia, or pumpkin, or anyone in particular — it's a conditional statement about everybody in our little world.

Suppose we pose the query:

```
?-  jealous(marsellus,W).
```

This query asks: can you find an individual W such that Marsellus is jealous of W? Vincent is such an individual. If you check the definition of jealousy, you'll see that Marsellus must be jealous of Vincent, because they both are friends with the same woman, namely Mia. So Prolog will return the value

```
W  =  vincent
```

Now some questions for you . First, are there any other jealous people in KB5? Furthermore, suppose we wanted Prolog to tell us about all the jealous people: what query would we pose? Do any of the answers surprise you? Do any seem silly?

2.2. Recursive Definitions

Predicates can be defined recursively. Roughly speaking, a predicate is recursively defined if one or more rules in its definition refers to itself.

Example 1: Eating

Consider the following knowledge base:

```
is_digesting(X,Y) :- just_ate(X,Y).
is_digesting(X,Y) :- just_ate(X,Z), is_digesting(Z,Y).
just_ate(mosquito,blood(john)).
just_ate(frog,mosquito).
just_ate(stork,frog).
```

At first glance this seems pretty ordinary: it's just a knowledge base containing three facts and two rules. But the definition of the `is_digesting/2` predicate is recursive. Note that `is_digesting/2` is (at least partially) defined in terms of itself, for the `is_digesting/2` functor occurs in both the head and body of the second rule. Crucially, however, there is an 'escape' from this circularity. This is provided by the `just_ate/2` predicate, which occurs in the first rule. (Significantly, the body of the first rule makes no mention of `is_digesting/2`.) Let's now consider both the declarative and procedural meanings of this definition.

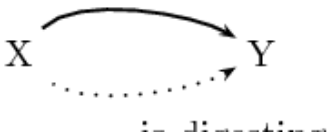
The word "declarative" is used to talk about the logical meaning of Prolog knowledge bases. That is, the declarative meaning of a Prolog knowledge base is simply "what it says", or "what it means, if we read it as a collection of logical statements". And the declarative meaning of this recursive definition is fairly straightforward. The first clause (the escape clause, the one that is not recursive, or as we shall usually call it, the base clause), simply says that: if X has just eaten Y, then X is now digesting Y. This is obviously a sensible definition.

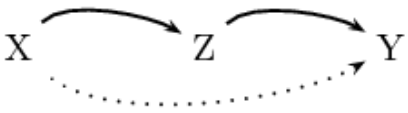
So what about the second clause, the recursive clause? This says that: if X has just eaten Z and Z is digesting Y, then X is digesting Y, too. Again, this is obviously a sensible definition.

So now we know what this recursive definition says, but what happens when we pose a query that actually needs to use this definition? That is, what does this definition actually do? To use the normal Prolog terminology, what is its procedural meaning?

This is also reasonably straightforward. The base rule is like all the earlier rules we've seen. That is, if we ask whether X is digesting Y, Prolog can use this rule to ask instead the question: has X just eaten Y?

What about the recursive clause? This gives Prolog another strategy for determining whether X is digesting Y: it can try to find some Z such that X has just eaten Z, and Z is digesting Y. That is, this rule lets Prolog break the task apart into two subtasks. Hopefully, doing so will eventually lead to simple problems which can be solved by simply looking up the answers in the knowledge base. The following picture sums up the situation:





Let's see how this works. If we pose the query:

```
?- is_digesting(stork,mosquito).
```

then Prolog goes to work as follows. First, it tries to make use of the first rule listed concerning is_digesting ; that is, the base rule. This tells it that X is digesting Y if X just ate Y , By unifying X with stork and Y with mosquito it obtains the following goal:

```
?- just_ate(stork,mosquito).
```

But the knowledge base doesn't contain the information that the stork just ate the mosquito, so this attempt fails. So Prolog next tries to make use of the second rule. By unifying X with stork and Y with mosquito it obtains the following goals:

```
?- just_ate(stork,Z), is_digesting(Z,mosquito).
```

That is, to show is_digesting(stork,mosquito) , Prolog needs to find a value for Z such that, firstly,

```
?- just_ate(stork,Z).
```

and secondly,

```
?- is_digesting(Z,mosquito).
```

And there is such a value for Z, namely frog . It is immediate that

```
?- just_ate(stork,frog).
```

will succeed, for this fact is listed in the knowledge base. And deducing

```
?- is_digesting(frog,mosquito).
```

is almost as simple, for the first clause of is_digesting/2 reduces this goal to deducing

```
?- just_ate(frog,mosquito).
```

and this is a fact listed in the knowledge base.

Well, that's our first example of a recursive rule definition. We're going to learn a lot more about them, but one very practical remark should be made right away. Hopefully it's clear that when you write a recursive predicate, it should always have at least two clauses: a base clause (the clause that stops the recursion at some point), and one that contains the recursion. If you don't do this, Prolog can spiral off into an unending sequence of useless computations. For example, here's an extremely simple example of a recursive rule definition:

```
p :- p.
```

That's it. Nothing else. It's beautiful in its simplicity. And from a declarative perspective it's an extremely sensible (if rather boring) definition: it says "if property p holds, then property p holds". You can't argue with that.

But from a procedural perspective, this is a wildly dangerous rule. In fact, we have here the ultimate in dangerous recursive rules: exactly the same thing on both sides, and no base clause to let us escape. For consider what happens when we pose the following query:

```
?- p.
```

Prolog asks itself: "How do I prove p?" and it realises, "Hey, I've got a rule for that! To prove p I just need to prove p!". So it asks itself (again): "How do I prove p?" and it realises, "Hey, I've got a rule for that! To prove p I just need to prove p!". So it asks itself (yet again): "How do I prove p?" and it realises, "Hey, I've got a rule for that! To prove p I just need to prove p!" and so on and so forth.

If you make this query, Prolog won't answer you: it will head off, looping desperately away in an unending search. That is, it won't terminate, and you'll have to interrupt it. Of course, if you use trace, you can step through one step at a time, until you get sick of watching Prolog loop.

Example 2: Descendant

Now that we know something about what recursion in Prolog involves, it is time to ask why it is so important. Actually, this is a question that can be answered on a number of levels, but for now, let's keep things fairly practical. So: when it comes to writing useful Prolog programs, are recursive definitions really so important? And if so, why?

Let's consider an example. Suppose we have a knowledge base recording facts about the child relation:

```
child(bridget,caroline).
```

```
child(caroline,donna).
```

That is, Caroline is a child of Bridget, and Donna is a child of Caroline. Now suppose we wished to define the descendant relation; that is, the relation of being a child of, or a child of a child of, or a child of a child of a child of, and so on. Here's a first attempt to do this. We could add the following two non -recursive rules to the knowledge base:

```
descend(X,Y) :- child(X,Y).
descend(X,Y) :- child(X,Z), child(Z,Y).
```

Now, fairly obviously these definitions work up to a point, but they are clearly limited: they only define the concept of descendant-of for two generations or less. That's ok for the above knowledge base, but suppose we get some more information about the child-of relation and we expand our list of child-of facts to this:

```
child(anne,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).
```

Now our two rules are inadequate. For example, if we pose the queries

```
?- descend(anne,donna).
```

or

```
?- descend(bridget,emily).
```

we get the answer no, which is not what we want. Sure, we could 'fix' this by adding the following two rules:

```
descend(X,Y) :- child(X,Z_1), child(Z_1,Z_2), child(Z_2,Y).
descend(X,Y) :- child(X,Z_1), child(Z_1,Z_2), child(Z_2,Z_3),
     child(Z_3,Y).
```

But, let's face it, this is clumsy and hard to read. Moreover, if we add further child-of facts, we could easily find ourselves having to add more and more rules as our list of child-of facts grow, rules like:

```
descend(X,Y) :-
child(X,Z_1),child(Z_1,Z_2),child(Z_2,Z_3),…,child(Z_17,Z_18),child(
     Z_18,Z_19),child(Z_19,Y).
```

This is not a particularly pleasant (or sensible) way to go!

But we don't need to do this at all. We can avoid having to use ever longer rules entirely. The following recursive predicate definition fixes everything exactly the way we want:

```
descend(X,Y) :- child(X,Y).
descend(X,Y) :- child(X,Z), descend(Z,Y).
```

What does this say? The declarative meaning of the base clause is: if Y is a child of X , then Y is a descendant of X . Obviously sensible. So what about the recursive clause? Its declarative meaning is: if Z is a child of X , and Y is a descendant ofZ , then Y is a descendant of X . Again, this is obviously true.

So let's now look at the procedural meaning of this recursive predicate, by stepping through an example. What happens when we pose the query:

```
?- descend(anne,donna).
```

Prolog first tries the first rule. The variable X in the head of the rule is unified with anne and Y with donna and the next goal Prolog tries to prove is

```
?- child(anne,donna).
```

This attempt fails, however, since the knowledge base neither contains the fact `child(anne,donna)` nor any rules that would allow to infer it. So Prolog backtracks and looks for an alternative way of proving `descend(anne,donna)` . It finds the second rule in the knowledge base and now has the following subgoals:

```
?- child(anne,_633), descend(_633,donna).
```

Prolog takes the first subgoal and tries to unify it with something in the knowledge base. It finds the fact `child(anne,bridget)` and the variable _633 gets instantiated to bridget . Now that the first subgoal is satisfied, Prolog moves to the second subgoal. It has to prove

```
?- descend(bridget,donna).
```

This is the first recursive call of the predicate `descend/2`. As before, Prolog starts with the first rule, but fails, because the goal

```
?- child(bridget,donna).
```

cannot be proved. Backtracking, Prolog finds that there is a second possibility to be checked for `descend(bridget,donna)`, namely the second rule, which again gives Prolog two new subgoals:

```
?- child(bridget,_1785), descend(_1785,donna).
```

The first one can be unified with the fact `child(bridget,caroline)` of the knowledge base, so that the variable _1785 is instantiated with caroline . Next Prolog tries to prove

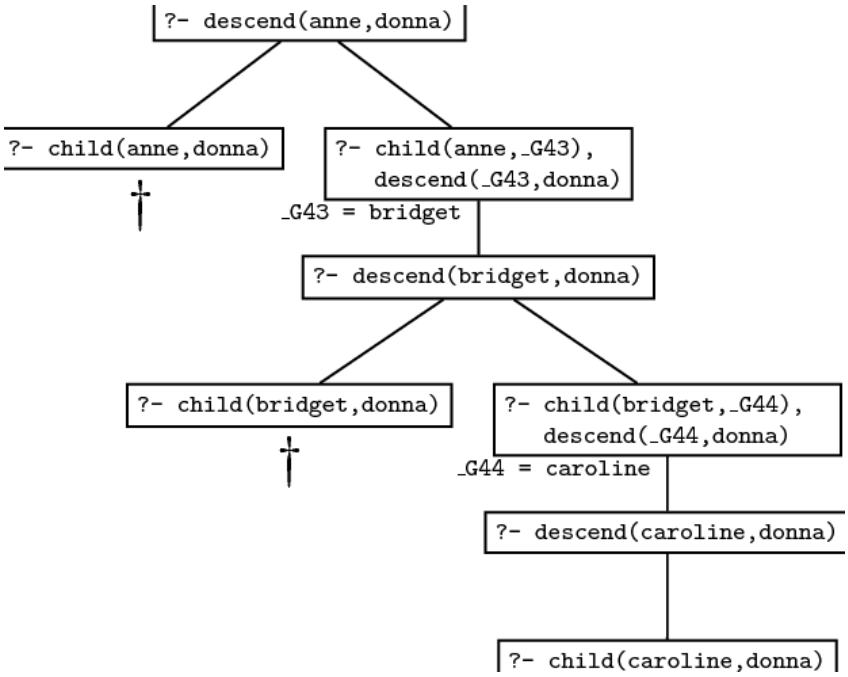
```
?- descend(caroline,donna).
```

This is the second recursive call of predicate `descend/2`. As before, it tries the first rule first, obtaining the following new goal:

```
?- child(caroline,donna).
```

This time Prolog succeeds, since `child(caroline,donna)` is a fact in the database. Prolog has found a proof for the goal `descend(caroline,donna)` (the second recursive call). But this means that `descend(bridget,donna)` (the first recursive call) is also true, which means that our original query `descend(anne,donna)` is true as well.

Here is the search tree for the query `descend(anne,donna)`. Make sure that you understand how it relates to the discussion in the text; that is, how Prolog traverses this search tree when trying to prove this query.



It should be obvious from this example that no matter how many generations of children we add, we will always be able to work out the descendant relation. That is, the recursive definition is both general and compact: it contains all the information in the non-recursive rules, and much more besides. The non-recursive rules only defined the descendant concept up to some fixed number of generations: we would need to write down infinitely many non-recursive rules if we wanted to capture this concept fully, and of course that's impossible. But, in effect, that's what the recursive rule does for us: it bundles up the information needed to cope with arbitrary numbers of generations into just three lines of code.

Recursive rules are really important. They enable to pack an enormous amount of information into a compact form and to define predicates in a natural way. Most of the work you will do as a Prolog programmer will involve writing recursive rules.

Example 3: Successor

Nowadays, when human beings write numerals, they usually use decimal notation (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, and so on) but as you probably know, there are many other notations. For example, because computer hardware is generally based on digital circuits, computers usually use binary notation to represent numerals (0, 1, 10, 11, 100, 101, 110, 111, 1000, and so on), for the 0 can be implemented as a switch being off, the 1 as a switch being on. Other cultures use different systems. For example, the ancient Babylonians used a base 60 system, while the ancient Romans used a rather ad-hoc system (I, II, III, IV, V, VI, VII, VIII, IX, X). This last example shows that notational issues can be important. If you don't believe this, try figuring out a systematic way of doing long-division in Roman notation. As you'll discover, it's a frustrating task. Apparently, the Romans had a group of professionals (analogs of modern accountants) who specialized in this.

Well, here's yet another way of writing numerals, which is sometimes used in mathematical logic. It makes use of just four symbols: 0, succ, and the left and right parentheses. This style of numeral is defined by the following inductive definition:

1. 0 is a numeral.
2. If X is a numeral, then so is succ(X) .

As is probably clear, succ can be read as short for successor. That is, `succ(X)` represents the number obtained by adding one to the number represented by X. So this is a very simple notation: it simply says that 0 is a numeral, and that all other numerals are built by stacking succ symbols in front. (In fact, it's used in mathematical logic because of this simplicity. Although it wouldn't be pleasant to do household accounts in this notation, it is a very easy notation to prove things about .)

Now, by this stage it should be clear that we can turn this definition into a Prolog program. The following knowledge base does this:

```
numeral(0).

numeral(succ(X)) :- numeral(X).
```

So if we pose queries like

```
?- numeral(succ(succ(succ(0)))).
```

we get the answer yes.

But we can do some more interesting things. Consider what happens when we pose the following query:

```
?- numeral(X).
```

That is, we're saying "Ok, show me some numerals". Then we can have the following dialogue with Prolog:

```
X = 0 ;

X = succ(0) ;

X = succ(succ(0)) ;
```

```
X = succ(succ(succ(0))) ;
X = succ(succ(succ(succ(0)))) ;
X = succ(succ(succ(succ(succ(0))))) ;
X = succ(succ(succ(succ(succ(succ(0)))))) ;
X = succ(succ(succ(succ(succ(succ(succ(0))))))) ;
X = succ(succ(succ(succ(succ(succ(succ(succ(0)))))))) ;
yes
```

Yes, Prolog is counting: but what's really important is how it's doing this. Quite simply, it's backtracking through the recursive definition, and actually building numerals using unification. This is an instructive example, and it is important that you understand it. The best way to do so is to sit down and try it out, with trace turned on.

Building and binding. Recursion, unification, and proof search. These are ideas that lie at the heart of Prolog programming. Whenever we have to generate or analyse recursively structured objects (such as these numerals) the interplay of these ideas makes Prolog a powerful tool. For example, in the next chapter we shall introduce lists, an extremely important recursive data structure, and we will see that Prolog is a natural list processing language. Many applications (computational linguistics is a prime example) make heavy use of recursively structured objects, such as trees and feature structures. So, it's not particularly surprising that Prolog has proved useful in such applications.

Example 4: Addition

As a final example, let's see whether we can use the representation of numerals that we introduced in the previous section for doing simple arithmetic. Let's try to define addition. That is, we want to define a predicate `add/3` which when given two numerals as the first and second argument returns the result of adding them up as its third argument. For example:

```
?- add(succ(succ(0)),succ(succ(0)), succ(succ(succ(succ(0))))).
yes
?- add(succ(succ(0)),succ(0),Y).
Y = succ(succ(succ(0)))
```

There are two things which are important to notice:

1. Whenever first argument is 0, the third argument has to be same as the second argument:

```
?- add(0,succ(succ(0)),Y).
Y = succ(succ(0))
?- add(0,0,Y).
Y = 0
```

This is the case that we want to use for the base clause.

2. Assume that we want to add the two numerals X and Y (for example `succ(succ(succ(0)))` and `succ(succ(0))`) and that X is not 0 . Now, if X1 is the numeral that has one succ functor less than X (that is, succ(succ(0)) in our example) and if we know the result – let's call it Z – of adding X1 and Y (namely succ(succ(succ(succ(0))))), then it is very easy to compute the result of adding X and Y : we just have to add one succ -functor to Z . This is what we want to express with the recursive clause.

Here is the predicate definition that expresses exactly what we just said:

```
add(0,Y,Y).
add(succ(X),Y,succ(Z)) :- add(X,Y,Z).
```

So what happens, if we give Prolog this predicate definition and then ask:

```
?- add(succ(succ(succ(0))), succ(succ(0)), R).
```

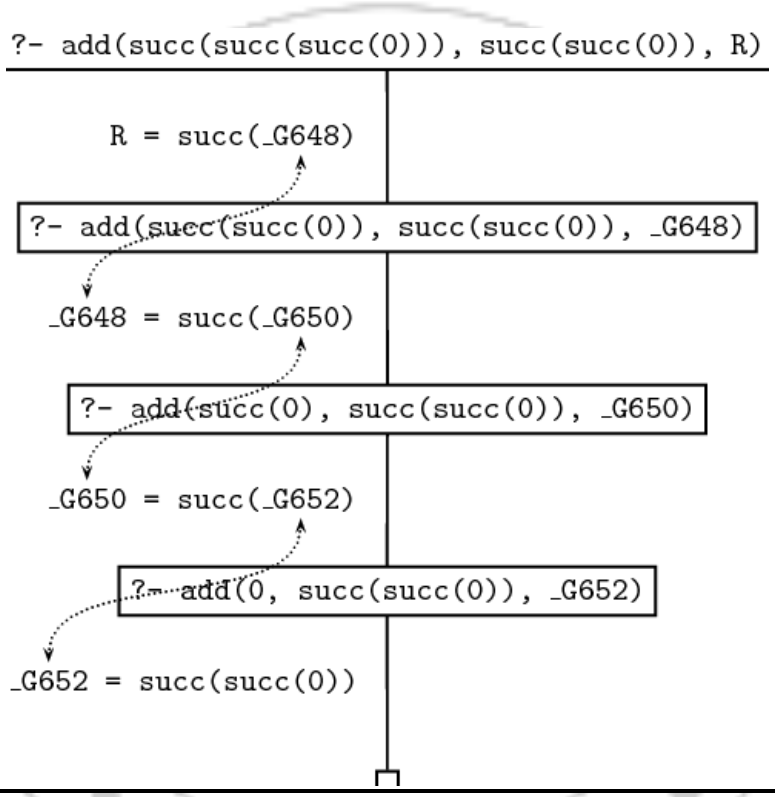
Let's go step by step through the way Prolog processes this query. The trace and search tree for the query are given below.

The first argument is not 0, which means that only the second clause for add/3 can be used. This leads to a recursive call of add/3. The outermost succ functor is stripped off the first argument of the original query, and the result becomes the first argument of the recursive query. The second argument is passed on unchanged to the recursive query, and the third argument of the recursive query is a variable, the internal variable _G648 in the trace given below. Note that _G648 is not instantiated yet. However, it shares values with R (the variable that we used as the third argument in the original query) because R was instantiated to succ(_G648) when the query was unified with the head of the second clause. But that means that R is not a completely uninstantiated variable anymore. It is now a complex term, that has a (uninstantiated) variable as its argument.

The next two steps are essentially the same. With every step the first argument becomes one layer of succ smaller; both the trace and the search tree given below show this nicely. At the same time, a succ functor is added to R at every step, but always leaving the innermost variable uninstantiated. After the first recursive call R is succ(_G648). After the second recursive call, _G648 is instantiated with succ(_G650), so that R is succ(succ(_G650). After the third recursive call, _G650 is instantiated with succ(_G652) and R therefore becomes succ(succ(succ(_G652))). The search tree shows this step by step instantiation.

At this stage all succ functors have been stripped off the first argument and we can apply the base clause. The third argument is equated with the second argument, so the 'hole' (the uninstantiated variable) in the complex term R is finally filled, and we are through.

Here's the search tree:



2.3. Rule Ordering, Goal Ordering, and Termination

Prolog was the first reasonably successful attempt to create a logic programming language. Underlying logic programming is a simple (and seductive) vision: the task of the programmer is simply to describe problems. The programmer should write down (in the language of logic) a declarative specification (that is: a knowledge base), which describes the situation of interest. The programmer shouldn't have to tell the computer what to do. To get information, he or she simply asks the questions. It's up to the logic programming system to figure out how to get the answer.

Well, that's the idea, and it should be clear that Prolog has taken some important steps in this direction. But Prolog is not , repeat not , a full logic programming language. If you only think about the declarative meaning of a Prolog program, you are in for a very tough time.

Recall our earlier descendant program (let's call it descend1.pl):

```
child(anne,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).
descend(X,Y) :- child(X,Y).
descend(X,Y) :- child(X,Z), descend(Z,Y).
```

We'll make one change to it, and call the result descend2.pl :

```
child(anne,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).
descend(X,Y) :- child(X,Z), descend(Z,Y).
descend(X,Y) :- child(X,Y).
```

All we have done is change the rule order. So if we read the program as a purely logical definition, nothing has changed. But does the change give rise to procedural differences? Yes, but nothing significant. For example, if you work through the examples you will see that the first solution that descend1.pl finds is

```
X = anne
Y = bridget
```

whereas the first solution that descend2.pl finds is

```
X = anne
Y = emily
```

But (as you should check) both programs generate exactly the same answers, they merely find them in a different order. And this is a general point. Roughly speaking changing the order of rules in a Prolog program does not change (up to the order in which solutions are found) the program's behaviour.

So let's move on. We'll make one small change to descend2.pl , and call the result descend3.pl :

```
child(anne,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).
descend(X,Y) :- descend(Z,Y), child(X,Z).
```

```
descend(X,Y) :- child(X,Y).
```

Note the difference. Here we've changed the goal order within a rule, not the rule order. Now, once again, if we read the program as a purely logical definition, nothing has changed; it means the same thing as the previous two versions. But this time the program's behaviour has changed dramatically. For example, if you pose the query

```
?- descend(anne,emily).
```

you will get an error message ("out of local stack", or something similar). Prolog is looping. Why? Well, in order to satisfy the query descend(anne,emily) Prolog uses the first rule. This means that its next goal will be to satisfy the query

```
?- descend(W1,emily).
```

for some new variable W1. But to satisfy this new goal, Prolog again has to use the first rule, and this means that its next goal is going to be

```
?- descend(W2,emily).
```

for some new variable W2. And of course, this in turn means that its next goal is going to be descend(W3,emily) and then descend(W4,emily) , and so on. That is, the (at first glance innocuous) change in the goal order has resulted in procedural disaster. To use the standard terminology, we have here a classic example of a left recursive rule, that is, a rule where the leftmost item of the body is identical (modulo the choice of variables) with the rule's head. As our example shows, such rules easily give rise to non-terminating computations. Goal order, and in particular left recursion, is the root of all evil when it comes to non-termination.

Still, as we said earlier, we need to make one small caveat about rule ordering. We said earlier that rule ordering only changes the order in which solutions are found. However, this may not be true if we are working with non-terminating programs. To see this, consider the fourth (and last) variant of our descendant program, namely descend4.pl:

```
child(anne,bridget).

child(bridget,caroline).

child(caroline,donna).

child(donna,emily).

descend(X,Y) :- child(X,Y).

descend(X,Y) :- descend(Z,Y), child(X,Z).
```

This program is descend3.pl with the rule ordering reversed. Now (once again) this program has the same declarative meaning as the other variants, but it is also procedurally different from its relatives. First, and most obviously, it is very different procedurally from both descend1.pl and descend2.pl. In particular, because it contains a left recursive rule, this new program does not terminate on some input. For example (just like descend3.pl) this new program does not terminate when we pose the query

```
?- descend(anne,emily).
```

But descend4.pl is not procedurally identical to descend3.pl. The rule ordering reversal does make a difference. For example, descend3.pl will not terminate if we pose the query

```
?- descend(anne,bridget).
```

However descend4.pl will terminate in this case, for the rule reversal enables it to apply the non-recursive rule and halt. So when it comes to non-terminating programs, rule ordering changes can lead to some extra solutions being found. Nonetheless, goal ordering, not rule ordering, is what is truly procedurally significant. To ensure termination, we need to pay attention to the order of goals within the bodies of rules. Tinkering with rule orderings does not get to grips with the roots of termination problems — at best it can yield some extra solutions.

Summing up, our four variant descendant programs are Prolog knowledge bases which describe exactly the same situations, but behave differently. The difference in behaviour between descend1.pl and descend2.pl (which differ only in the way rules are ordered) is relatively minor: they generate the same solutions, but in a different order. But descend3.pl and descend4.pl are procedurally very different from their two cousins, and this is because they differ from them in the way their goals are ordered. In particular, both these variants contain left recursive rules, and in both cases this leads to non-terminating behaviour. The change in rule ordering between descend3.pl and descend4.pl merely means that descend4.pl will terminate in some cases where descend3.pl will not.

What are the ramifications of our discussion for the practicalities of producing working Prolog programs? It's probably best to say the following. Often you can get the overall idea (the big picture) of how to write the program by thinking declaratively, that is, by thinking in terms of describing the problem accurately. This is an excellent way to approach problems, and certainly the one most in keeping with the spirit of logic programming. But once you've done that, you need to think about how Prolog will work with knowledge bases you have written. In particular, to ensure termination, you need to check that the goal orderings you have given are sensible. The basic rule of thumb is never to write as the leftmost goal of the body something that is identical (modulo variable names) with the goal given in the head. Rather, place such goals (which trigger recursive calls) as far as possible towards the right of the tail. That is, place them after the goals which test for the various (non-recursive) termination conditions. Doing this gives Prolog a sporting chance of fighting it's way through your recursive definitions to find solutions.

Exercises

Exercise 2.1

In the text, we discussed the predicate

```
descend(X,Y) :- child(X,Y).
descend(X,Y) :- child(X,Z), descend(Z,Y).
```

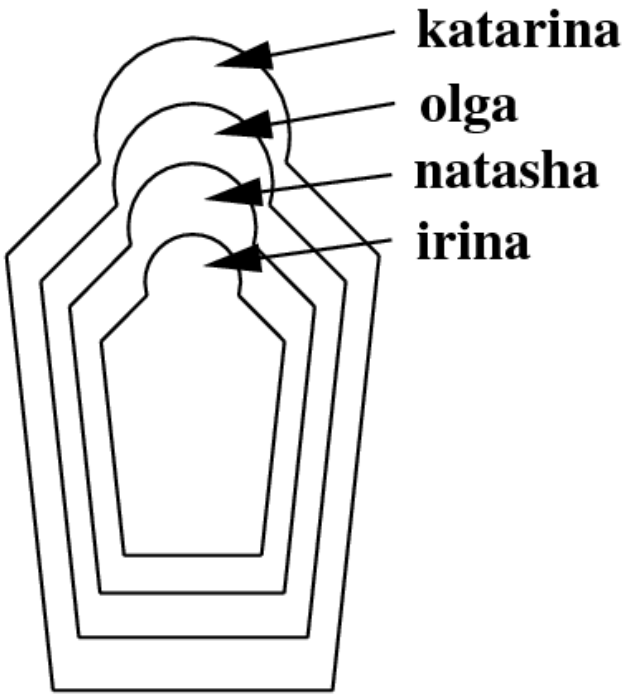
Suppose we reformulated this predicate as follows:

```
descend(X,Y) :- child(X,Y).
descend(X,Y) :- descend(X,Z),descend(Z,Y).
```

Would this be problematic?

Exercise 2.2

Do you know these wooden Russian dolls (Matryoshka dolls) where the smaller ones are contained in bigger ones? Here is a schematic picture:



First, write a knowledge base using the predicate `directlyIn/2` which encodes which doll is directly contained in which other doll. Then, define a recursive predicate `in/2`, that tells us which doll is (directly or indirectly) contained in which other dolls. For example, the query `in(katarina,natasha)` should evaluate to `true`, while `in(olga, katarina)` should fail.

Exercise 2.3

We have the following knowledge base:

```
directTrain(saarbruecken,dudweiler).
directTrain(forbach,saarbruecken).
directTrain(freyming,forbach).
directTrain(stAvold,freyming).
directTrain(fahlquemont,stAvold).
directTrain(metz,fahlquemont).
directTrain(nancy,metz).
```

That is, this knowledge base holds facts about towns it is possible to travel between by taking a direct train. But of course, we can travel further by chaining together direct train journeys. Write a recursive predicate `travelFromTo/2` that tells us when we can travel by train between two towns. For example, when given the query

```
travelFromTo(nancy,saarbruecken).
```

it should reply `yes`.

Exercise 2.4

Define a predicate `greater_than/2` that takes two numerals in the notation that we introduced in the text (that is, `0`, `succ(0)`, `succ(succ(0))`, and so on) as arguments and decides whether the first one is greater than the second one. For example:

```
?- greater_than(succ(succ(succ(0))),succ(0)).
yes
?- greater_than(succ(succ(0)),succ(succ(succ(0)))).
no
```

Exercise 2.5

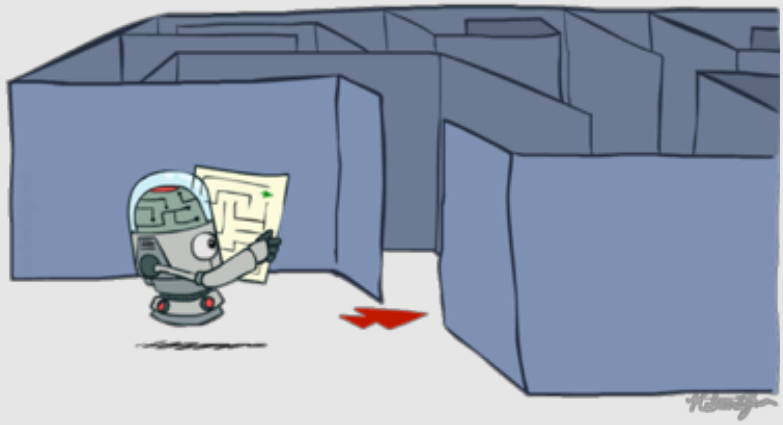
Binary trees are trees where all internal nodes have exactly two children. The smallest binary trees consist of only one leaf node. We will represent leaf nodes as `leaf(Label)`. For instance, `leaf(3)` and `leaf(7)` are leaf nodes, and therefore small binary trees. Given two binary trees B1 and B2 we can combine them into one binary tree using the functor `tree/2` as follows: `tree(B1,B2)`. So, from the leaves `leaf(1)` and `leaf(2)` we can build the binary tree `tree(leaf(1),leaf(2))`. And from the binary trees `tree(leaf(1),leaf(2))` and `leaf(4)` we can build the binary tree `tree(tree(leaf(1), leaf(2)),leaf(4))`.

Now, define a predicate `swap/2`, which produces the mirror image of the binary tree that is its first argument. For example:

```
?- swap(tree(tree(leaf(1), leaf(2)), leaf(4)),T).
T = tree(leaf(4), tree(leaf(2), leaf(1))).
yes
```

CS351-L – Introduction to Artificial Intelligence Lab 3

UNINFORMED SEARCH TECHNIQUES



Contents: Dr. Hashim Ali

Ghulam Ishaq Khan Institute of Engineering Sciences

Faculty of Computer Science & Engineering

CS351L – Introduction to Artificial Intelligence Lab

Lab 3 – Uninformed search techniques

Uninformed Search Techniques for Artificial Intelligence

Objective

The objective of this session is to learn how to formulate search problems in which there is no information about goal state and to implement the basic techniques for a specific problem.

Learning outcomes

1. To explain the basics behind uninformed search problems.
2. To be able to identify the best search technique suitable for an uninformed search problem.
3. To understand, apply and implement solutions for uninformed search problems like traversing a graph.

Instructions

- Read through the handout sitting in front of a computer that has a Python software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering the problems, indicate the commands you entered and output displayed.
- It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document.

Table of Contents

3.0. Overview of uninformed search algorithms

An uninformed (a.k.a. blind, brute-force) search algorithm generates the search tree without using any domain specific knowledge.
The two basic approaches differ as to whether you check for a goal when a node is *generated* or when it is *expanded*.

Checking at generation time:

```
if start_state is a goal state return the empty action list
fringe := [make_node(start_state, null, null)]
while fringe is not empty
        n := select and remove some node from the fringe
        for each action a applicable to n.state
                s := succ(n.state, a)
                n' := make_node(s, a, n)
                if s is a goal state, return n'.actionListFromRoot()
                add n' to fringe
return failure
```

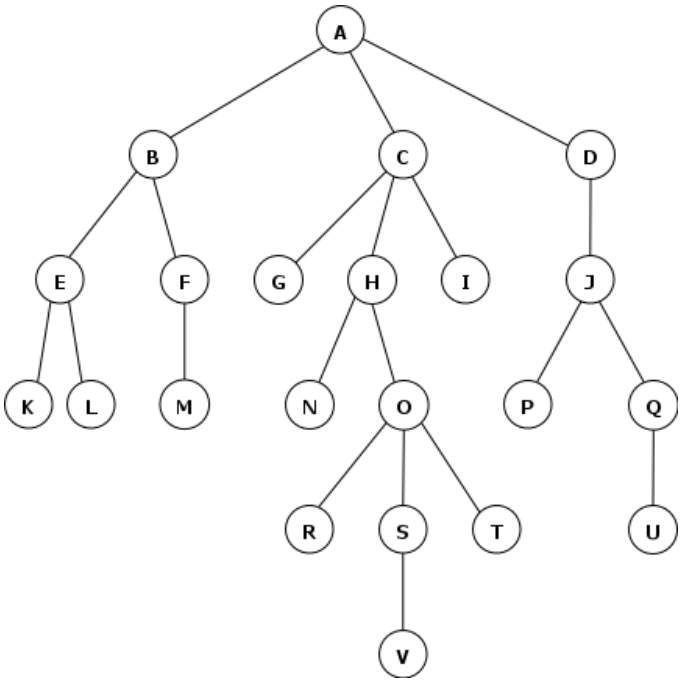
Checking at expansion time:

```
fringe := [make_node(start_state, null, null)]
while fringe is not empty
        n := select and remove some node from the fringe
        if n.state is a goal state return n.actionListFromRoot()
        for each action a applicable to n.state
                add make_node(succ(n.state, a), a, n) to fringe
return failure
```

3.1. Uninformed Search Algorithms

3.1.1. Breadth-First Search

Strategy: expand the shallowest unexpanded node. Implementation: The fringe is a FIFO queue.
For this search tree



The queue evolves like this

A
B C D
C D E F
D E F G H I
E F G H I J
F G H I J K L
G H I J K L M
H I J K L M
I J K L M N O
J K L M N O
K L M N O P Q
L M N O P Q
M N O P Q
N O P Q
O P Q
P Q R S T
Q R S T
R S T U
S T U
T U V
U V
V

That is, the order of generation is ABCDEFGHIJKLMNOPQRSTUV and the order of expansion is the same.

If the goal nodes were M, V, and J, Breadth-First search would find J, the shallowest.

BFS

- is complete (if *b* is finite)
- is optimal if all path costs are the same (because it always finds the shallowest node first)

3.1.2. Depth-First Search

Strategy: expand the deepest unexpanded node. Implementation: The fringe is a LIFO queue (stack)

For the search tree above:

A
B C D
E F C D
K L F C D
L F C D
F C D
M C D
C D

G H I D
H I D
N O I D
O I D
R S T I D
S T I D
V T I D
T I D
I D
D
J
P Q
Q
U

If the goal nodes were M, V, and J, the Depth-First search above would find M.

DFS

- is not complete (because of infinite depth and loops)
- is not optimal

3.1.3. Depth-Limited Search

This is just a depth-first search with a cutoff depth. Here is the algorithm (for the test-at-expansion-time case)

```
fringe := [make_node(start_state, null, null)]
reached_limit := false
while fringe is not empty
      n := fringe.pop()
      if n.state is a goal state return n.actionListFromRoot()
      if n.depth == limit reached_limit := true
      else
              for each action a applicable to n.state
                      fringe.push(make_node(succ(n.state, a), a, n))
return reached_limit ? cutoff : failure
```

DLS

- Won't run forever unless b is infinite
- is not complete because a goal may be below the cutoff
- is not optimal

3.1.4. Depth-First Iterative Deepening

Algorithm:

```
for i in 1..infinity
      run DLS to level i
      if found a goal at level i, return it immediately
```

DFID

- is complete for finite b
- is optimal in terms of the solution depth (and optimal in general if path cost is non-decreasing function of depth)
- generates fewer nodes than the version of BFS that checks for goals at expansion time
- is **asymptotically optimal in terms of time and space among brute-force tree searches that find optimal solutions!**

3.1.5. Uniform Cost Search

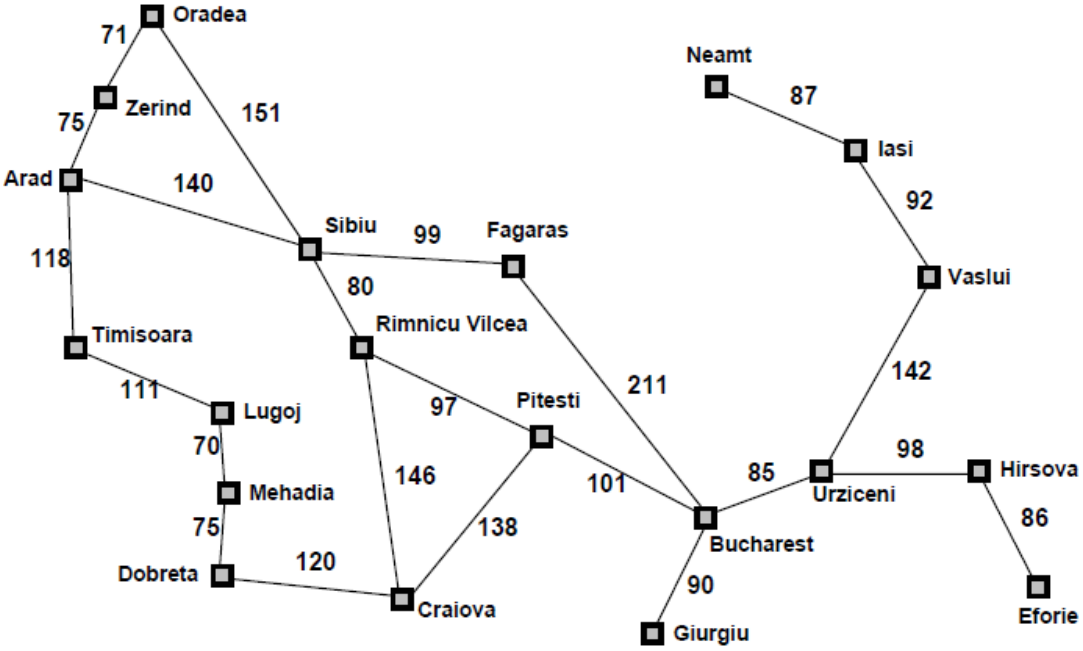
Strategy: Expand the lowest cost node. Implementation: the fringe is a priority queue: lowest cost node has the highest priority. In order to be optimal, must test at expansion, not generation, time.

3.1.6. Backwards Chaining

Run the search backwards from a goal state to a start state. Obviously, this works best when the actions are reversible, and the set of goal states is small.

3.2. Travelling on vacations to Romania

Suppose that you plan to spend your summer vacations in Romania. Following is the map of Romania.



3.3. Exercises

Exercise 3.1:

Write a program that takes in a starting city and a city to reach (e.g., Oradea Zerind), and returns the shortest path, cities (nodes) explored during the search and the distance between them using Breadth first search algorithm.

Exercise 3.2:

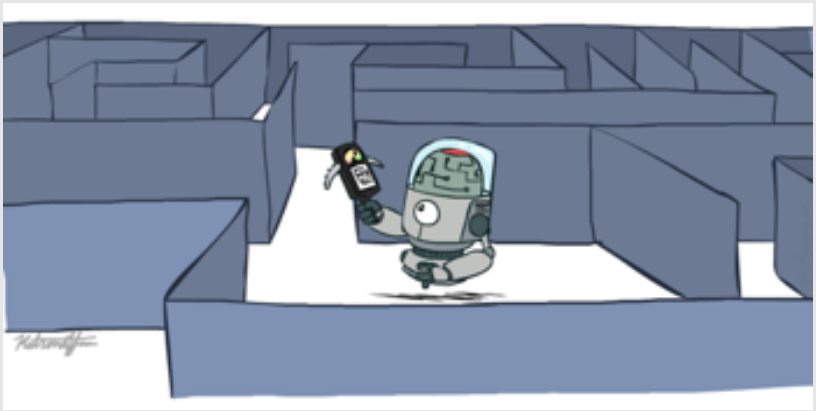
Copy the solution to Exercise 3.1. and modify it such that it now uses Depth first search.

Expected solution:

```
Breadth First Search algorithm
Oradea Zerind
Oradea Sibiu
Oradea Zerind Arad
Oradea Sibiu Fagaras
Oradea Sibiu Rimnicu Vilcea
Oradea Zerind Arad Timisoara
Oradea Sibiu Fagaras Bucharest
Oradea Sibiu Rimnicu Vilcea Craiova
Oradea Sibiu Rimnicu Vilcea Pitesti
Oradea Zerind Arad Timisoara Lugoj
Oradea Sibiu Fagaras Bucharest Urziceni
Oradea Sibiu Fagaras Bucharest Giurgiu
Oradea Sibiu Rimnicu Vilcea Craiova Dobreta
Oradea Zerind Arad Timisoara Lugoj Mehadia
Oradea Sibiu Fagaras Bucharest Urziceni Hirsova
Oradea Sibiu Fagaras Bucharest Urziceni Vaslui
Oradea Sibiu Fagaras Bucharest Urziceni Hirsova Eforie
Oradea Sibiu Fagaras Bucharest Urziceni Vaslui Iasi
Oradea Sibiu Fagaras Bucharest Urziceni Vaslui Iasi Neamt

Depth First Search algorithm
Oradea Zerind
Oradea Sibiu
Oradea Sibiu Arad
Oradea Sibiu Fagaras
Oradea Sibiu Rimnicu Vilcea
Oradea Sibiu Rimnicu Vilcea Craiova
Oradea Sibiu Rimnicu Vilcea Pitesti
Oradea Sibiu Rimnicu Vilcea Pitesti Bucharest
Oradea Sibiu Rimnicu Vilcea Pitesti Bucharest Urziceni
Oradea Sibiu Rimnicu Vilcea Pitesti Bucharest Giurgiu
Oradea Sibiu Rimnicu Vilcea Pitesti Bucharest Urziceni Hirsova
Oradea Sibiu Rimnicu Vilcea Pitesti Bucharest Urziceni Vaslui
Oradea Sibiu Rimnicu Vilcea Pitesti Bucharest Urziceni Vaslui Iasi
Oradea Sibiu Rimnicu Vilcea Pitesti Bucharest Urziceni Vaslui Iasi Neamt

Shortest path
The shortest path from Oradea to Neamt is 835 miles and goes as follows:

Oradea to Sibiu
Sibiu to Rimnicu Vilcea
Rimnicu Vilcea to Pitesti
Pitesti to Bucharest
Bucharest to Urziceni
Urziceni to Vaslui
Vaslui to Iasi
Iasi to Neamt

_
```

CS351-L – Introduction to Artificial Intelligence Lab 4

INFORMED SEARCH TECHNIQUES



Contents: Dr. Hashim Ali

Ghulam Ishaq Khan Institute of Engineering Sciences

Faculty of Computer Science & Engineering

CS351L – Introduction to Artificial Intelligence Lab

Lab 4 – Informed search techniques

Informed Search Techniques for Artificial Intelligence

Objective

The objective of this session is to learn how to formulate search problems in which there is some information about goal state, for instance some heuristic indicating distance/closeness to goal and to implement these basic techniques for a specific problem.

Learning outcomes

1. To explain the basics behind informed search problems.
2. To be able to identify the best search technique suitable for an informed search problem.
3. To apply and implement solutions for informed search problems like graph traversal.

Instructions

- Read through the handout sitting in front of a computer that has a Python software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering the problems, indicate the commands you entered and output displayed.
- It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document.

Table of Contents

4.0. Overview of Informed search algorithms

So far, we have talked about the uninformed search algorithms which looked through search space for all possible solutions of the problem without having any additional knowledge about search space. But informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc. This knowledge help agents to explore less to the search space and find more efficiently the goal node.
The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

Heuristics function: Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal. The heuristic method, however, might not always give the best solution, but it is guaranteed to find a good solution in reasonable time. Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

Admissibility of the heuristic function is given as:

$$h(n) <= h^*(n)$$

Here h(n) is heuristic cost, and h*(n) is the estimated cost. Hence heuristic cost should be less than or equal to the estimated cost.

4.1. Pure Heuristic Search:

Pure heuristic search is the simplest form of heuristic search algorithms. It expands nodes based on their heuristic value h(n). It maintains two lists, OPEN and CLOSED list. In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.
On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list. The algorithm continues unit a goal state is found.
In the informed search we will discuss two main algorithms which are given below:

- Best First Search Algorithm(Greedy search)
- A* Search Algorithm

4.1.1. Best-first Search Algorithm (Greedy Search):

Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.,

$$f(n)= g(n).$$

Where, g(n)= estimated cost from node n to the goal.
The greedy best first algorithm is implemented by the priority queue.
Best first search algorithm:

- **Step 1:** Place the starting node into the OPEN list.
- **Step 2:** If the OPEN list is empty, Stop and return failure.

- **Step 3:** Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
- **Step 4:** Expand the node n, and generate the successors of node n.
- **Step 5:** Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
- **Step 6:** For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.
- **Step 7:** Return to Step 2.

Advantages:

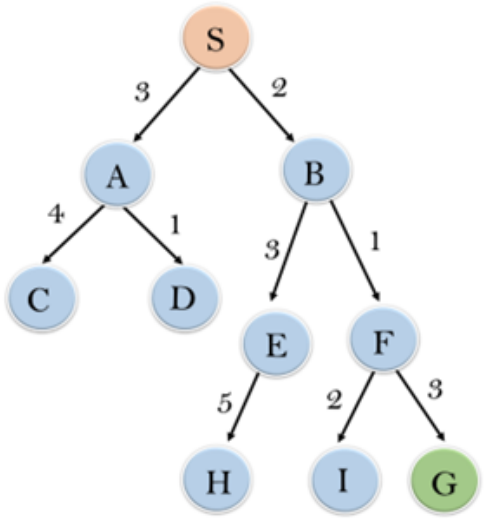
- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

Disadvantages:

- It can behave as an unguided depth-first search in the worst case scenario.
- It can get stuck in a loop as DFS.
- This algorithm is not optimal.

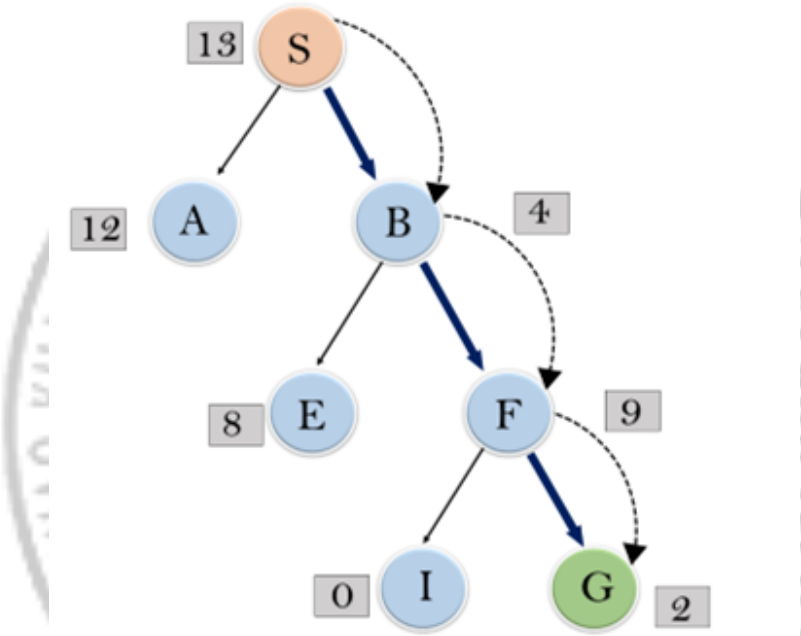
Example:

Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n), which is given in the table below.



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

In this search example, we are using two lists which are **OPEN** and **CLOSED** Lists. Following are the iterations for traversing the above example.



Expand the nodes of S and put in the CLOSED list

Initialization: Open [A, B], Closed [S]

Iteration 1: Open [A], Closed [S, B]

Iteration 2: Open [E, F, A], Closed [S, B]
: Open [E, A], Closed [S, B, F]

Iteration 3: Open [I, G, E, A], Closed [S, B, F]
: Open [I, E, A], Closed [S, B, F, G]

Hence the final solution path will be: **S----> B----->F----> G**

Time Complexity: The worst case time complexity of Greedy best first search is $O(b^m)$.

Space Complexity: The worst case space complexity of Greedy best first search is $O(b^m)$. Where, m is the maximum depth of the search space.

Complete: Greedy best-first search is also incomplete, even if the given state space is finite.

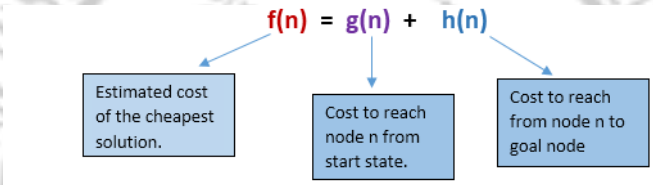
Optimal: Greedy best first search algorithm is not optimal.

4.1.2. A* Search Algorithm:

A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm

expands less search tree and provides optimal result faster. A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).

In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a **fitness number**.



At each point in the search space, only those node is expanded which have the lowest value of f(n), and the algorithm terminates when the goal node is found.

Algorithm of A search:*

Step1: Place the starting node in the OPEN list.

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise

Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.

Step 6: Return to **Step 2**.

Advantages:

- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

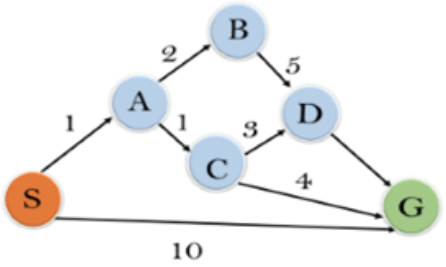
Disadvantages:

- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A* search algorithm has some complexity issues.
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

Example:

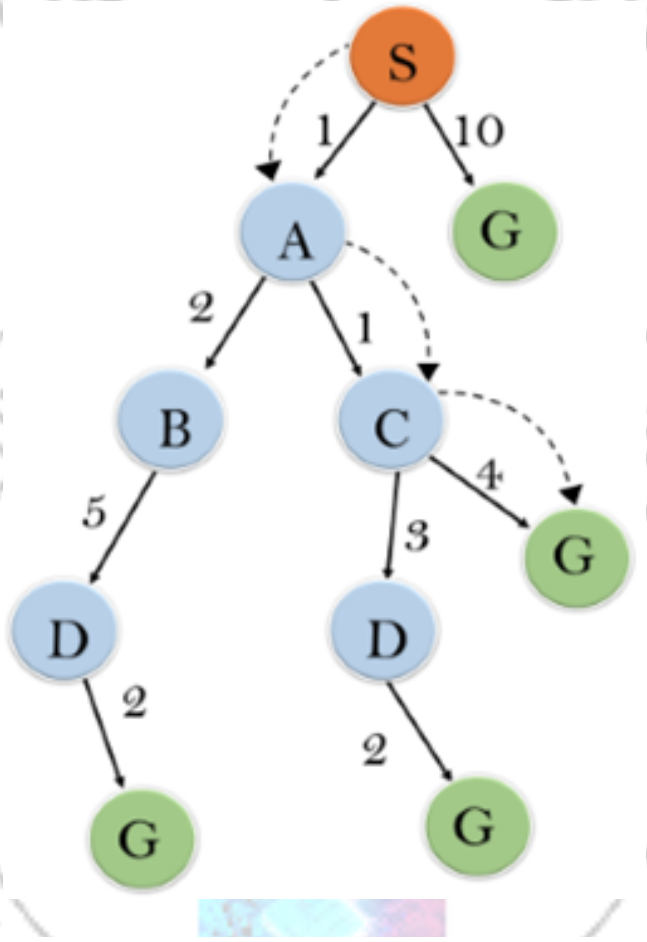
In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.

Here we will use OPEN and CLOSED list.



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Solution:



Initialization: {(S, 5)}

Iteration1: {(S--> A, 4), (S-->G, 10)}

Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}

Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}

Iteration 4 will give the final result, as **S--->A--->C--->G** it provides the optimal path with cost 6.

Points to remember:

- A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition f(n)

Complete: A* algorithm is complete as long as:

- Branching factor is finite.
- Cost at every action is fixed.

Optimal: A* search algorithm is optimal if it follows below two conditions:

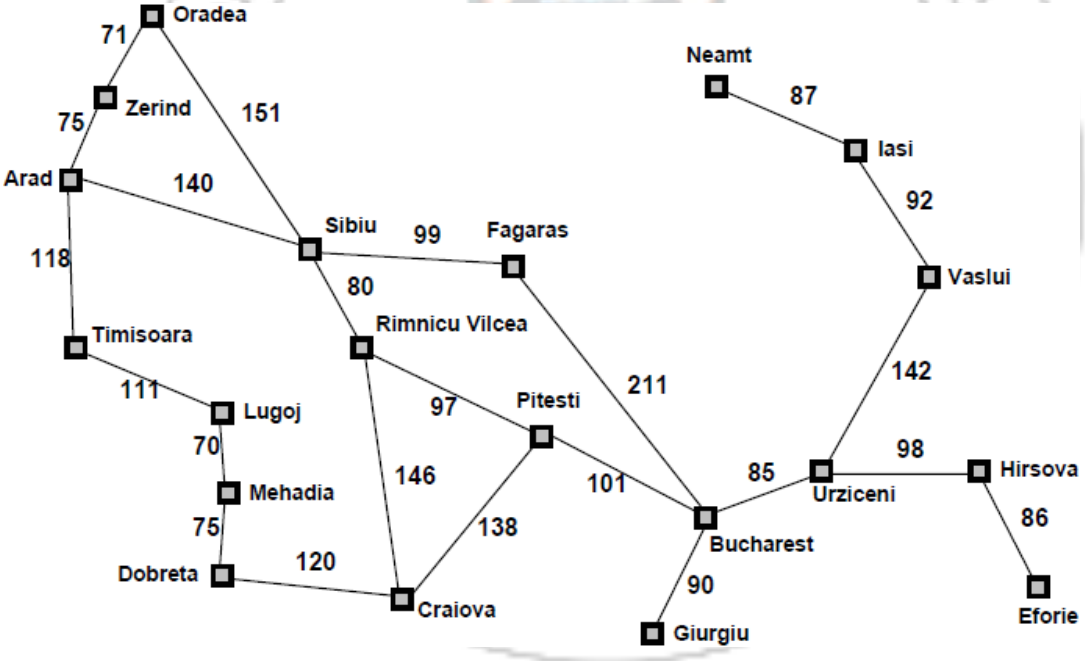
- **Admissible:** the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
- **Consistency:** Second required condition is consistency for only A* graph-search.

If the heuristic function is admissible, then A* tree search will always find the least cost path.

4.2. Informed search vs. Uninformed search

Example 1:

Suppose that you plan to spend vacations in Romania. Following is the map of Romania.



Suppose that you are to travel from Oradea to Neamt and want to determine the shortest path for it. As is seen from the problem, the only information you have is distances between cities and thus the problem at this point can only be solved by using uninformed search techniques, e.g., by using depth first search and breadth first search. Here is the solution, if you applied those search techniques (as already done by you in the previous lab).

```
Breadth First Search algorithm
Oradea Zerind
Oradea Sibiu
Oradea Zerind Arad
Oradea Sibiu Fagaras
Oradea Sibiu Rimnicu Vilcea
Oradea Zerind Arad Timisoara
Oradea Sibiu Fagaras Bucharest
Oradea Sibiu Rimnicu Vilcea Craiova
Oradea Sibiu Rimnicu Vilcea Pitesti
Oradea Zerind Arad Timisoara Lugoj
Oradea Sibiu Fagaras Bucharest Urziceni
Oradea Sibiu Fagaras Bucharest Giurgiu
Oradea Sibiu Rimnicu Vilcea Craiova Dobreta
Oradea Zerind Arad Timisoara Lugoj Mehadia
Oradea Sibiu Fagaras Bucharest Urziceni Hirsova
Oradea Sibiu Fagaras Bucharest Urziceni Vaslui
Oradea Sibiu Fagaras Bucharest Urziceni Hirsova Eforie
Oradea Sibiu Fagaras Bucharest Urziceni Vaslui Iasi
Oradea Sibiu Fagaras Bucharest Urziceni Vaslui Iasi Neamt

Depth First Search algorithm
Oradea Zerind
Oradea Sibiu
Oradea Sibiu Arad
Oradea Sibiu Fagaras
Oradea Sibiu Rimnicu Vilcea
Oradea Sibiu Rimnicu Vilcea Craiova
Oradea Sibiu Rimnicu Vilcea Pitesti
Oradea Sibiu Rimnicu Vilcea Pitesti Bucharest
Oradea Sibiu Rimnicu Vilcea Pitesti Bucharest Urziceni
Oradea Sibiu Rimnicu Vilcea Pitesti Bucharest Giurgiu
Oradea Sibiu Rimnicu Vilcea Pitesti Bucharest Urziceni Hirsova
Oradea Sibiu Rimnicu Vilcea Pitesti Bucharest Urziceni Vaslui
Oradea Sibiu Rimnicu Vilcea Pitesti Bucharest Urziceni Vaslui Iasi
Oradea Sibiu Rimnicu Vilcea Pitesti Bucharest Urziceni Vaslui Iasi Neamt

Shortest path
The shortest path from Oradea to Neamt is 835 miles and goes as follows:

Oradea to Sibiu
Sibiu to Rimnicu Vilcea
Rimnicu Vilcea to Pitesti
Pitesti to Bucharest
Bucharest to Urziceni
Urziceni to Vaslui
Vaslui to Iasi
Iasi to Neamt
```

Example 2:

Now lets suppose that you are to travel to Bucharest starting from Arad. But as compared to previous example, in addition to the map and their mutual distances between cities, you also are given the following table listing the Euclidean distance (heuristic) between a given city and Bucharest.

City	Heuristic value	City	Heuristic value
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Eforie	161	Pitesti	100

Fagaras	176	Rimnicu Vilcea	193
Dobreta	242	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

How will this change the problem? For one, now this becomes an informed search problem because now from each city, we have a heuristic value (Euclidean distance) which indicates if we are going closer to our goal with each move or moving away from it, unlike Example 1, where we were searching in complete darkness (uninformed) in comaprison. Hence the problem has now shifted from being uninformed to informed.

4.3. Exercises

Exercise 4.1:

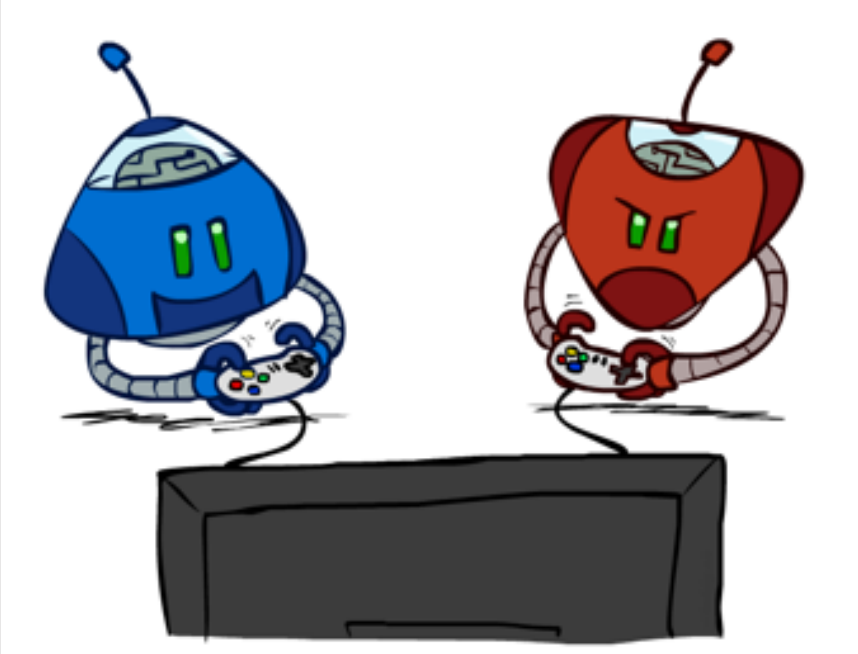
Implement the best-first search algorithm on the map such that for any given city, it should return the shortest distance from that city to Bucharest using the heuristic based on best-first search algorithm.

Exercise 4.2:

Implement the A* algorithm on the map such that for any given city, it should return the shortest distance from that city to Bucharest using the heuristic based on A* search algorithm.

CS351-L – Introduction to Artificial Intelligence Lab 5

ADVERSARIAL SEARCH STRATEGIES



Contents: Dr. Hashim Ali

Ghulam Ishaq Khan Institute of Engineering Sciences

Faculty of Computer Science & Engineering

CS351L – Introduction to Artificial Intelligence Lab

Lab 5 – Adversarial search strategies

Adversarial Search Strategies for Artificial Intelligence

Objective

The objective of this session is to learn how to formulate search problems that arise when we try to plan ahead in a world where other agents are planning against us.

Learning outcomes

After the successful completion of this lab, the student will be able to:

1. Explain the basics behind adversarial search problems and when they are applicable (games/competitive environments between two or more agents).
2. Apply algorithms and techniques designed for adversarial search problem e.g., minimax algorithm.
3. Minimize the search space using pruning techniques.

Instructions

- Read through the handout sitting in front of a computer that has a Python software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering the problems, indicate the commands you entered and output displayed.
- It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document.

Table of Contents

5.0. Overview of Adversarial search algorithms

So far, we have talked about the uninformed and informed search algorithms, which looked through search space for all possible solutions of the problem with and without the additional information. But there are specific cases, where two or more agents are competing together with/against each other for a common goal. For example, let us take the case of robotic football match, where two teams of robots are playing in a controlled environment to make such moves that their team is able to score a goal while stopping the other team from doing so.

Such an environment is substantially different from what you have seen so far in the CS351 lab mainly because there was no need to calculate and counter the expected move from the opponent. The unpredictability of these other agents can introduce contingencies into the agent's problem-solving process.

In this session, we cover competitive environments, in which the agent's goals are in conflict, giving rise to adversarial search problems, often known as games. Real-world or simulated cases in Artificial Intelligence, where two or more agents interact together in a competitive environment for achieving the same goal/maximize a limited resource are solved using adversarial search techniques.

5.1. Games

Introduction:

Mathematical game theory, a branch of economics, views any multiagent environment as a game, provided that the impact of each agent on the others is "significant," regardless of whether the agents are cooperative or competitive. In AI, the most common games are of a rather specialized kind—what game theorists call deterministic, turn-taking, two-player, zero-sum games of perfect information (such as chess). In our terminology, this means deterministic, fully observable environments in which two agents act alternately and in which the utility values at the end of the game are always equal and opposite. For example, if one player wins a game of chess, the other player necessarily loses. It is this opposition between the agents' utility functions that makes the situation adversarial.

Games have engaged the intellectual faculties of humans—sometimes to an alarming degree—for as long as civilization has existed. For AI researchers, the abstract nature of games makes them an appealing subject for study. The state of a game is easy to represent, and agents are usually restricted to a small number of actions whose outcomes are defined by precise rules. Physical games, such as croquet and ice hockey, have much more complicated descriptions, a much larger range of possible actions, and rather imprecise rules defining the legality of actions. With the exception of robot soccer, these physical games have not attracted much interest in the AI community.

Why are games interesting for AI Researchers?

Games are interesting because they are too hard to solve. For example, chess has an average branching factor of about 35, and games often go to 50 moves by each player, so the search tree has about 35100 or 10154 nodes (although the search graph has "only" about 1040 distinct nodes). Games, like the real world, therefore require the ability to make some decision even when calculating the optimal decision is infeasible. Games also penalize inefficiency severely. Whereas an implementation of A∗ search that is half as efficient will simply take twice as long to run to completion, a chess program that is half as efficient in using its available time probably will be beaten into the ground, other things being equal. Game-playing research has therefore spawned a number of interesting ideas on how to make the best possible use of time.

Problem Formulation and Description:

We first consider games with two players, whom we call MAX and MIN for reasons that will soon become obvious. MAX moves first, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser. A game can be formally defined as a kind of search problem with the following elements:

- S_0: The initial state, which specifies how the game is set up at the start.
- PLAYER**(s)**: Defines which player has the move in a state.
- ACTIONS**(s)**: Returns the set of legal moves in a state.
- RESULT(**s**, **a**): The transition model, which defines the result of a move.
- TERMINAL-TEST(s): A terminal test, which is true when the game is over and false otherwise. States where the game has ended are called terminal states.
- UTILITY(s, p): A utility function (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p. In chess, the outcome is a win, loss, or draw, with values +1, 0, or ½. Some games have a wider variety of possible outcomes; the payoffs in backgammon range from 0 to +192.
- A zero-sum game is (confusingly) defined as one where the total payoff to all players is the same for every instance of the game. Chess is zero-sum because every game has payoff of either 0 + 1, 1 + 0 or ½ + ½. "Constant-sum" would have been a better term, but zero-sum is traditional and makes sense if you imagine each player is charged an entry fee of ½.

Game Tree

The initial state, ACTIONS function, and RESULT function define the game tree for the game—a tree where the nodes are game states and the edges are moves. Figure 5.1 shows part of the game tree for tic-tac-toe (noughts and crosses). From the initial state, MAX has nine possible moves. Play alternates between MAX's placing an X and MIN's placing an O until we reach leaf nodes corresponding to terminal states such that one player has three in a row or all the squares are filled. The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN (which is how the players get their names).

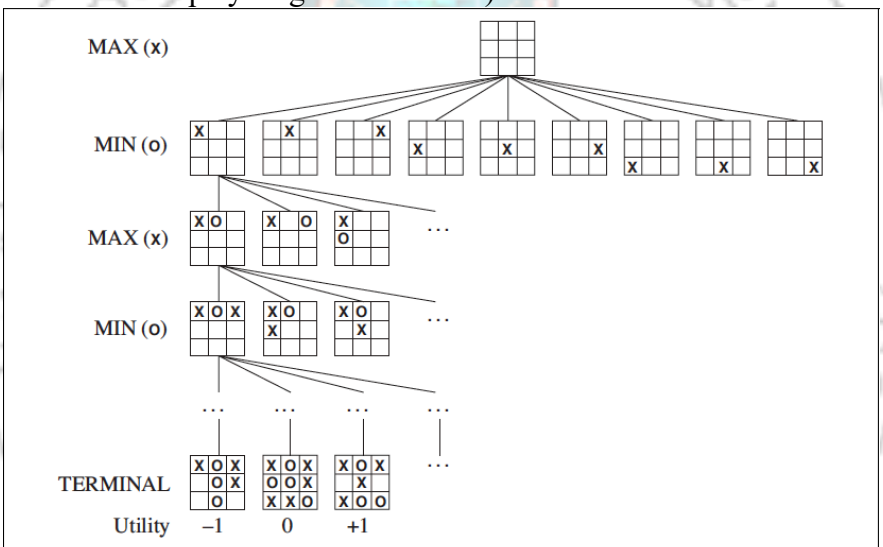


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

Search Tree

For tic-tac-toe the game tree is relatively small—fewer than 9! = 362, 880 terminal nodes. But for chess there are over 1040 nodes, so the game tree is best thought of as a theoretical construct that we cannot realize in the physical world. But regardless of the size of the game tree, it is MAX's job to search for a good move. We use the term search tree for a tree that is superimposed on the full game tree, and examines enough nodes to allow a player to determine what move to make.

5.2. Optimal Decisions in Games:

Strategy

In a normal search problem, the optimal solution would be a sequence of actions leading to a goal state—a terminal state that is a win. In adversarial search, MIN has something to say about it. MAX therefore must find a contingent strategy, which specifies MAX's move in the initial state, then MAX's moves in the states resulting from every possible response by MIN, then MAX's moves in the states resulting from every possible response by MIN to those moves, and so on. Roughly speaking, an optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent. We begin by showing how to find this optimal strategy.

Even a simple game like tic-tac-toe is too complex for us to draw the entire game tree on one page, so we will switch to the trivial game in Figure 5.2. The possible moves for MAX at the root node are labeled a_1, a_2, and a_3. The possible replies to a_1 for MIN are b_1, b_2, b_3, and so on. This particular game ends after one move each by MAX and MIN. (In game parlance, we say that this tree is one move deep, consisting of two half-moves, each of which is called a ply.) The utilities of the terminal states in this game range from 2 to 14.

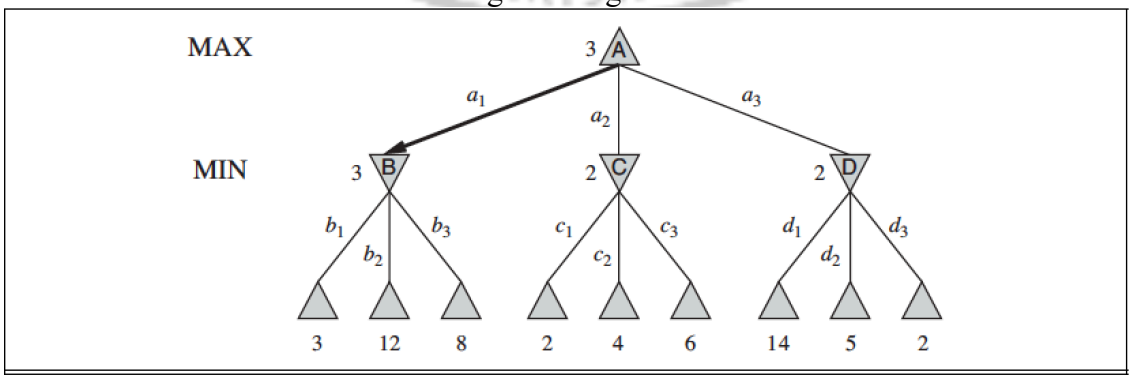


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Given a game tree, the optimal strategy can be determined from the minimax value of each node, which we write as MINIMAX**(n)**. The minimax value of a node is the utility (for MAX) of being in the corresponding state, assuming that both players play optimally from there to the end of the game. Obviously, the minimax value of a terminal state is just its utility. Furthermore, given a choice, MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value. So we have the following:

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

Let us apply these definitions to the game tree in Figure 5.2. The terminal nodes on the bottom level get their utility values from the game's UTILITY function. The first MIN node, labeled **B**, has three successor states with values 3, 12, and 8, so its minimax value is 3. Similarly, the other two MIN nodes have minimax value 2. The root node is a MAX node; its successor states have minimax values 3, 2, and 2; so it has a minimax value of 3. We can also identify the minimax decision at the root: action $\mathbf{a}_1$ is the optimal choice for MAX because it leads to the state with the highest minimax value.

This definition of optimal play for MAX assumes that MIN also plays optimally—it maximizes the worst-case outcome for MAX. What if MIN does not play optimally? Then it is easy to show that MAX will do even better. Other strategies against suboptimal opponents may do better than the minimax strategy, but these strategies necessarily do worse against optimal opponents.

5.2.1. The Minimax Algorithm:

The minimax algorithm (Figure 5.3) computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are backed up through the tree as the recursion unwinds. For example, in Figure 5.2, the algorithm first recurses down to the three bottom left nodes and uses the UTILITY function on them to discover that their values are 3, 12, and 8, respectively. Then it takes the minimum of these values, 3, and returns it as the backed-up value of node **B**. A similar process gives the backed-up values of 2 for **C** and 2 for **D**. Finally, we take the maximum of 3, 2, and 2 to get the backed-up value of 3 for the root node.

The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is **m** and there are **b** legal moves at each point, then the time complexity of the minimax algorithm is $\mathbf{O(b^m)}$. The space complexity is **O(bm)** for an algorithm that generates all actions at once, or **O(m)** for an algorithm that generates actions one at a time. For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

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

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

Figure 5.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation $\text{argmax}_{a \in S} f(a)$ computes the element a of set S that has the maximum value of $f(a)$.

5.2.2. Alpha-Beta Pruning

The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree. Unfortunately, we can't eliminate the exponent, but it turns out we can effectively cut it in half. The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree. That is, we can borrow the idea of pruning to eliminate large parts of the tree from consideration. The particular technique we examine is called alpha–beta pruning. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

Pruning algorithm:

Consider again the two-ply game tree from Figure 5.2. Let's go through the calculation of the optimal decision once more, this time paying careful attention to what we know at each point in the process. The steps are explained in Figure 5.4. The outcome is that we can identify the minimax decision without ever evaluating two of the leaf nodes. Another way to look at this is as a simplification of the formula for MINIMAX. Let the two unevaluated successors of node C in Figure 5.4 have values x and y. Then the value of the root node is given by

$$\begin{aligned}\text{MINIMAX}(root) &= \max(\min(3,12,8),\min(2,x,y),\min(14,5,2)) \\ &= \max(3,\min(2,x,y),2) \\ &= \max(3,z,2) \qquad \text{where } z=\min(2,x,y)\le 2 \\ &= 3.\end{aligned}$$

In other words, the value of the root and hence the minimax decision are independent of the values of the pruned leaves x and y.

Alpha–beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves. The general principle is this: consider a node n somewhere in the tree (see Figure 5.5), such that Player has a choice of moving to that node. If Player has a better choice m either at the parent node of n or at any choice point further up, then n will never be reached in actual play. So once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.

Remember that minimax search is depth-first, so at any one time we just have to consider the nodes along a single path in the tree. Alpha–beta pruning gets its name from the following two parameters that describe bounds on the backed-up values that appear anywhere along the path:

α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.

β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

Alpha–beta search updates the values of α and β as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively.

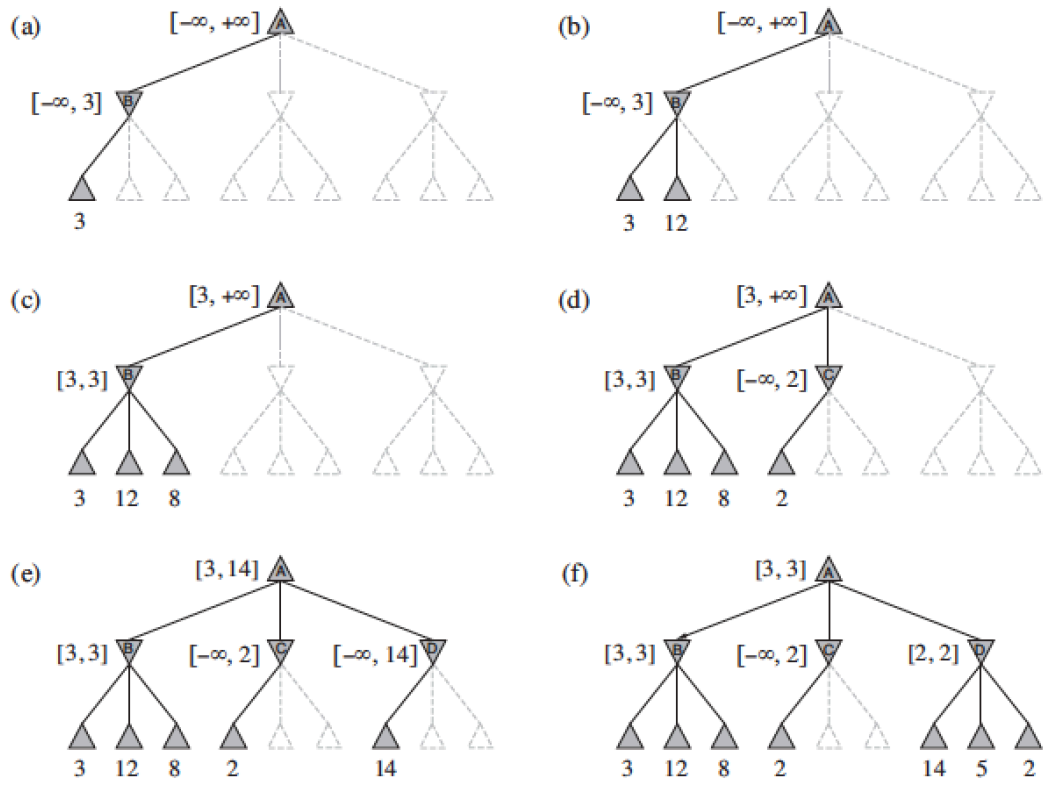


Figure 5.4 Stages in the calculation of the optimal decision for the game tree in Figure 5.2. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of *at most* 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. This is an example of alpha–beta pruning. (e) The first leaf below D has the value 14, so D is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

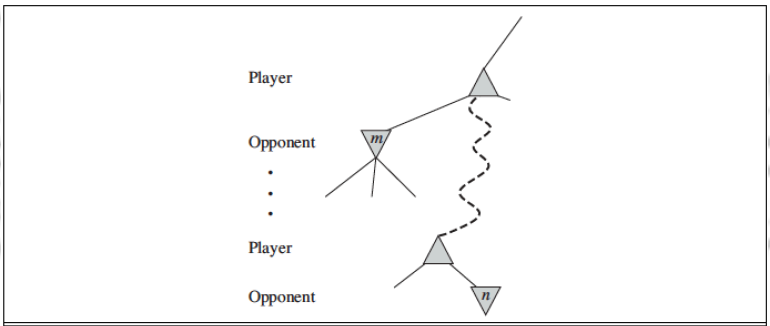
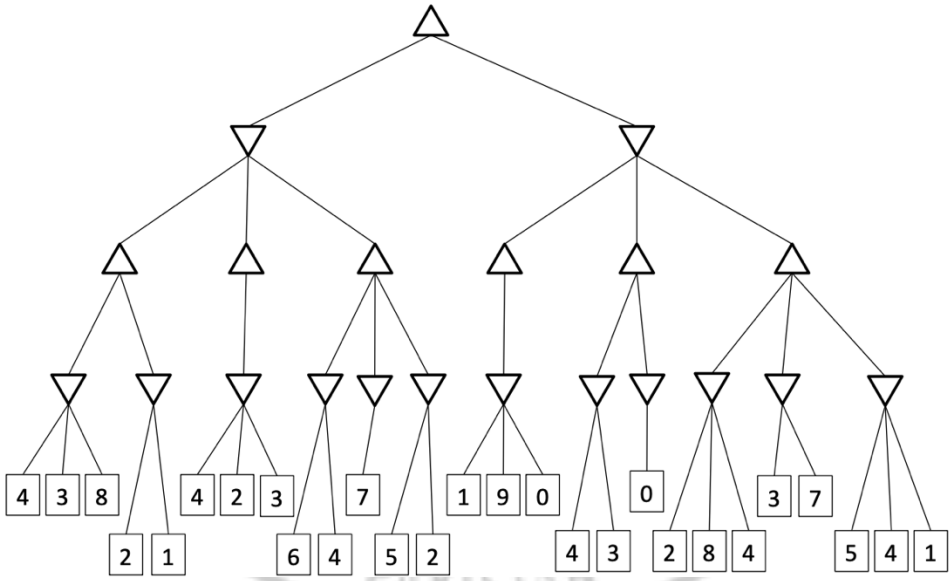


Figure 5.5 The general case for alpha–beta pruning. If m is better than n for Player, we will never get to n in play.

5.3. Exercises

Exercise 5.1:

Implement minimax algorithm for the following search tree. Your implementation should indicate the most optimal path and value of each node using minimax algorithm without pruning. Assume optimal opponent in both max and min.



Exercise 5.2:

Add alpha-beta pruning to your code. Your code should indicate which branches were pruned, the min-max value at each node and the optimal path. Assume optimal opponent in both max and min.

CS351-L – Introduction to Artificial Intelligence Lab 6

OPTIMIZATION AND LOCAL SEARCH ALGORITHMS

Contents: Dr. Hashim Ali

Ghulam Ishaq Khan Institute of Engineering Sciences

Faculty of Computer Science & Engineering

CS351L – Introduction to Artificial Intelligence Lab

Lab 6 – Optimization and Local Search Algorithms I

Optimization Problems and Local Search Techniques I

Objective

The objective of this session is to learn how to formulate search problems where the solution is not known, we start with an approximate solution, can generate new solutions in the proximity of current solution and can evaluate each solution for its goodness.

Learning outcomes

After the successful completion of this lab, the student will be able to:

1. Explain the basics behind optimization problems and when they are applicable.
2. Apply algorithms and techniques designed for optimization problems e.g., local hill climbing, simulated annealing and genetic algorithms.

Instructions

- Read through the handout sitting in front of a computer that has a Python software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering the problems, indicate the commands you entered and output displayed.
- It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document.

Table of Contents

6.1. Overview of Optimization problems

Optimization is an important tool in making decisions and in analyzing physical systems. In mathematical terms, an ***optimization problem*** is the problem of finding the *best* solution from among the set of all *feasible* solutions.

Constructing a Model

The first step in the optimization process is constructing an appropriate model; modeling is the process of identifying and expressing in mathematical terms the objective, the variables, and the constraints of the problem.

- An *objective* is a quantitative measure of the performance of the system that we want to minimize or maximize. In manufacturing, we may want to maximize the profits or minimize the cost of production, whereas in fitting experimental data to a model, we may want to minimize the total deviation of the observed data from the predicted data.
- The *variables* or the *unknowns* are the components of the system for which we want to find values. In manufacturing, the variables may be the amount of each resource consumed or the time spent on each activity, whereas in data fitting, the variables would be the parameters of the model.
- The *constraints* are the functions that describe the relationships among the variables and that define the allowable values for the variables. In manufacturing, the amount of a resource consumed cannot exceed the available amount.

Finding optimal solution for optimization problems

Finding an optimal solution for certain optimisation problems can be an incredibly difficult task, often practically impossible. This is because when a problem gets sufficiently large we need to search through an enormous number of possible solutions to find the optimal one. Even with modern computing power there are still often too many possible solutions to consider. In this case because we can't realistically expect to find the optimal one within a sensible length of time, we have to settle for something that's close enough.

An example optimisation problem which usually has a large number of possible solutions would be the traveling salesman problem or determining the minimum/maximum value for a given equation with variables. In order to find a solution to a problem such as the traveling salesman problem we need to use an algorithm that's able to find a good enough solution in a reasonable amount of time. In this session the algorithms we will be using are simpler optimization algorithms, e.g., 'local hill climbing' and 'simulated annealing'. In the next session, we will work with more complex optimization algorithms, namely, genetic algorithms.

6.2. Local hill climbing

Introduction & Background:

Hill Climbing is 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.

1. In the above definition, **mathematical optimization problems** implies 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 salesman.
2. 'Heuristic search' means that this search algorithm may not find the optimal solution to the problem. However, it will give a good solution in **reasonable time.**

3. A **heuristic function** is a function that will rank all the possible alternatives at any branching step in 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 generate and test algorithm. The generate and test algorithm is as follows:
   a) *Generate a possible solution.*
   b) *Test to see if this is the expected solution.*
   c) *If the solution has been found quit else go to step a).*

   Hence we call Hill climbing as a variant of generate and test algorithm as it takes the feedback from 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 next node. **Algorithm for Simple Hill climbing**:

```
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 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 next node.

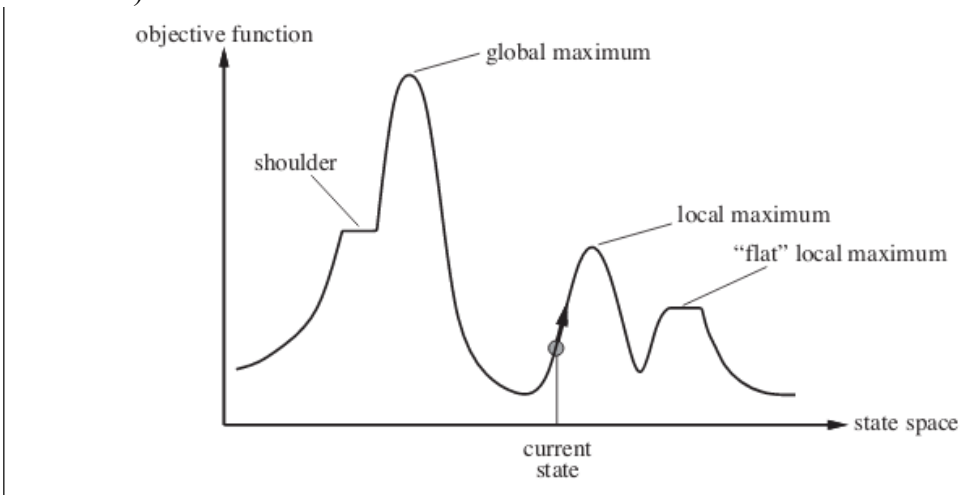
```
Step 1: Evaluate the initial state. If it is goal state then exit else make
the current state as initial state.
Step 2: Repeat these steps until a solution is found or current state does
not change.
    i. Let 'target' be a state such that any successor of the current state
will be better than it;
   ii.   for   each   operator   that   applies   to   the   current   state
        a. apply the new operator and create a new state
        b. evaluate the new state
        c. if this state is goal state then quit else compare with 'target'
        d. if this state is better than 'target', set this state as 'target'
        e. if target is better than current state set current state to Target
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.

State Space diagram for Hill Climbing

State space diagram is a graphical representation of the set of states our search algorithm can reach vs the value of our objective function(the function which we wish to maximize). **X-axis :** denotes the state space ie states or configuration our algorithm may reach. **Y-axis :** denotes the values of objective function corresponding to to a particular state. The best solution will be that state space where objective function has maximum value(global maximum).



Different regions in the State Space Diagram

- **Local maximum:** It is a state which is better than its neighboring state however there exists a state which is better than it(global maximum). This state is better because here value of objective function is higher than its neighbors.
- **Global maximum:** It is the best possible state in the state space diagram. This because at this state, objective function has highest value.
- **Plateau/flat local maximum:** It is a flat region of state space where neighboring states have the same value.
- **Ridge:** It is region which is higher than its neighbors but itself has a slope. It is a special kind of local maximum.
- **Current state:** The region of state space diagram where we are currently present during the search.
- **Shoulder:** It is a plateau that has an uphill edge.

Problems in different regions in Hill climbing

Hill climbing cannot reach the optimal/best state(global maximum) if it enters any of the following regions:

i. **Local maximum :** At a local maximum all neighboring states have a values which is worse than than the current state. Since hill climbing uses greedy approach, it will not move to the worse state and terminate itself. The process will end even though a better solution may exist.
   To overcome local maximum problem : Utilize backtracking technique. Maintain a list of visited states. If the search reaches an undesirable state, it can backtrack to the previous configuration and explore a new path.

ii. **Plateau :** On plateau all neighbors have same value . Hence, it is not possible to select the best direction.
   To overcome plateaus : Make a big jump. Randomly select a state far away from current state. Chances are that we will land at a non-plateau region.

iii. **Ridge :** Any point on a ridge can look like peak because movement in all possible directions is downward. Hence the algorithm stops when it reaches this state.
To overcome Ridge : In this kind of obstacle, use two or more rules before testing. It implies moving in several directions at once.

6.3. Simulated Annealing

Introduction & Background:

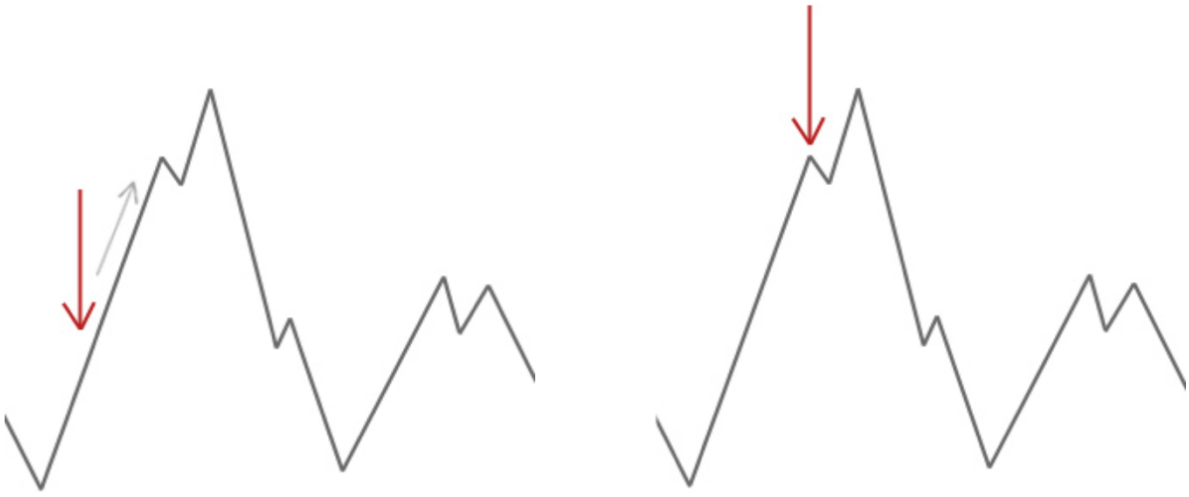
First, let's look at how simulated annealing works, and why it's good at finding solutions to the traveling salesman problem in particular. The simulated annealing algorithm was originally inspired from the process of annealing in metal work. Annealing involves heating and cooling a material to alter its physical properties due to the changes in its internal structure. As the metal cools its new structure becomes fixed, consequently causing the metal to retain its newly obtained properties. In simulated annealing we keep a temperature variable to simulate this heating process. We initially set it high and then allow it to slowly 'cool' as the algorithm runs. While this temperature variable is high the algorithm will be allowed, with more frequency, to accept solutions that are worse than our current solution. This gives the algorithm the ability to jump out of any local optimums it finds itself in early on in execution. As the temperature is reduced so is the chance of accepting worse solutions, therefore allowing the algorithm to gradually focus in on a area of the search space in which hopefully, a close to optimum solution can be found. This gradual 'cooling' process is what makes the simulated annealing algorithm remarkably effective at finding a close to optimum solution when dealing with large problems which contain numerous local optimums. The nature of the traveling salesman problem makes it a perfect example.

Advantages:

You may be wondering if there is any real advantage to implementing simulated annealing over something like a simple hill climber. Although hill climbers can be surprisingly effective at finding a good solution, they also have a tendency to get stuck in local optimums. As we previously determined, the simulated annealing algorithm is excellent at avoiding this problem and is much better on average at finding an approximate global optimum.

To help better understand let's quickly take a look at why a basic hill climbing algorithm is so prone to getting caught in local optimums.

A hill climber algorithm will simply accept neighbour solutions that are better than the current solution. When the hill climber can't find any better neighbours, it stops.



In the example above we start our hill climber off at the red arrow and it works its way up the hill until it reaches a point where it can't climb any higher without first descending. In this

example we can clearly see that it's stuck in a local optimum. If this were a real-world problem, we wouldn't know how the search space looks so unfortunately, we wouldn't be able to tell whether this solution is anywhere close to a global optimum.

Simulated annealing works slightly differently than this and will occasionally accept worse solutions. This characteristic of simulated annealing helps it to jump out of any local optimums it might have otherwise got stuck in.

Acceptance Function:

Let's take a look at how the algorithm decides which solutions to accept so we can better understand how it is able to avoid these local optimums.

First, we check if the neighbour solution is better than our current solution. If it is, we accept it unconditionally. If however, the neighbour solution isn't better we need to consider a couple of factors. Firstly, how much worse the neighbour solution is; and secondly, how high the current 'temperature' of our system is. At high temperatures the system is more likely accept solutions that are worse.

The math for this is pretty simple:

```
exp( (solutionEnergy – neighbourEnergy) / temperature )
```

Basically, the smaller the change in energy (the quality of the solution), and the higher the temperature, the more likely it is for the algorithm to accept the solution.

Algorithm Overview:

So how does the algorithm look? Well, in its most basic implementation it's pretty simple.

- First we need set the initial temperature and create a random initial solution.
- Then we begin looping until our stop condition is met. Usually either the system has sufficiently cooled, or a good-enough solution has been found.
- From here we select a neighbour by making a small change to our current solution.
- We then decide whether to move to that neighbour solution.
- Finally, we decrease the temperature and continue looping

Temperature Initilization:

For better optimisation, when initialising the temperature variable we should select a temperature that will initially allow for practically any move against the current solution. This gives the algorithm the ability to better explore the entire search space before cooling and settling in a more focused region.

6.3. Exercises

Exercise 6.1:

Find the minimum to the objective function

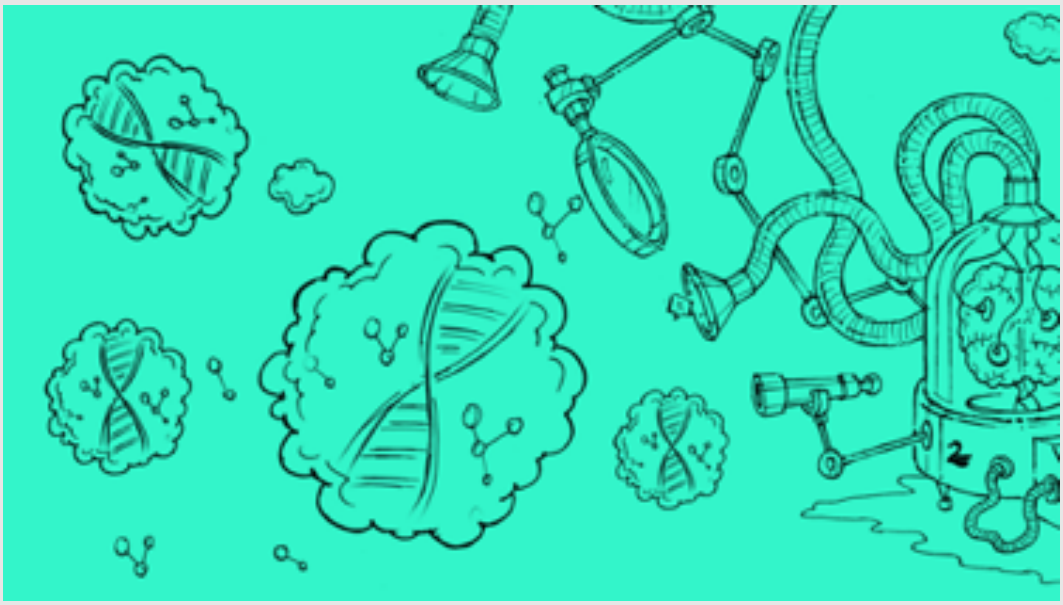
$$obj = 0.2 + x_1^2 + x_2^2 - 0.1\ \cos(6\pi x_1) - 0.1\cos(6\pi x_2)$$

by adjusting the values of x_1 and x_2. Implement local hill climbing algorithm to solve this problem. Divide your reg# with 4. Depending on the remainder, you must implement

- Remainder 0 – simple hill climbing,
- Remainder 1 – steepest ascent hill climbing,
- Remainder 2 – stochastic hill climbing, and
- Remainder 3 – simulated annealing

CS351-L – Introduction to Artificial Intelligence Lab 7

OPTIMIZATION AND LOCAL SEARCH ALGORITHMS



Contents: Dr. Hashim Ali

Ghulam Ishaq Khan Institute of Engineering Sciences

Faculty of Computer Science & Engineering

CS351L – Introduction to Artificial Intelligence Lab

Lab 7 – Optimization and Local Search Algorithms II

Optimization Problems and Local Search Techniques II

Objective

The objective of this session is to learn how to formulate search problems where the solution is not known, we start with an approximate solution, can generate new solutions in the proximity of current solution and can evaluate each solution for its goodness.

Learning outcomes

After the successful completion of this lab, the student will be able to:

1. Explain the basics behind optimization problems and when they are applicable.
2. Apply algorithms and techniques designed for optimization problems e.g., local hill climbing, simulated annealing and genetic algorithms.

Instructions

- Read through the handout sitting in front of a computer that has a Python software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering the problems, indicate the commands you entered and output displayed.
- It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document.

Table of Contents

7.0. What we have learnt so far about optimization problems

Optimization is the process of **making something better**. In any process, we have a set of inputs and a set of outputs as shown in the following figure.

Optimization refers to finding the values of inputs in such a way that we get the "best" output values. The definition of "best" varies from problem to problem, but in mathematical terms, it refers to maximizing or minimizing one or more objective functions, by varying the input parameters.

The set of all possible solutions or values which the inputs can take make up the search space. In this search space, lies a point or a set of points which gives the optimal solution. The aim of optimization is to find that point or set of points in the search space.

7.1. Overview of Genetic algorithms

Genetic Algorithm (GA) is a search-based optimization technique based on the principles of **Genetics and Natural Selection**. It is frequently used to find optimal or near-optimal solutions to difficult problems which otherwise would take a lifetime to solve. It is frequently used to solve optimization problems, in research, and in machine learning.

What are Genetic Algorithms?

Nature has always been a great source of inspiration to all mankind. Genetic Algorithms (GAs) are search based algorithms based on concepts of natural selection and genetics. GAs are a subset of a much larger branch of computation known as **Evolutionary Computation**.

GAs were developed by John Holland and his students and colleagues at the University of Michigan, most notably David E. Goldberg and has since been tried on various optimization problems with a high degree of success.

In GAs, we have a **pool or a population of possible solutions** to the given problem. These solutions then undergo recombination and mutation (like in natural genetics), producing new children, and the process is repeated over various generations. Each individual (or candidate solution) is assigned a fitness value (based on its objective function value) and the fitter individuals are given a higher chance to mate and yield more "fitter" individuals. This is in line with the Darwinian Theory of "Survival of the Fittest".

In this way we keep "evolving" better individuals or solutions over generations, till we reach a stopping criterion.

Genetic Algorithms are sufficiently randomized in nature, but they perform much better than random local search (in which we just try various random solutions, keeping track of the best so far), as they exploit historical information as well.

A genetic algorithm (GA) is great for finding solutions to complex search problems. They're often used in fields such as engineering to create incredibly high quality products thanks to their ability to search a through a huge combination of parameters to find the best match. For example, they can search through different combinations of materials and designs to find the perfect combination of both which could result in a stronger, lighter and overall,

better final product. They can also be used to design computer algorithms, to schedule tasks, and to solve other optimization problems. Genetic algorithms are based on the process of evolution by natural selection which has been observed in nature. They essentially replicate the way in which life uses evolution to find solutions to real world problems. Surprisingly although genetic algorithms can be used to find solutions to incredibly complicated problems, they are themselves pretty simple to use and understand.

Advantages of GAs

GAs have various advantages which have made them popular. These include:

- Does not require any derivative information (which may not be available for many real-world problems).
- Is faster and more efficient as compared to the traditional methods.
- Has very good parallel capabilities.
- Optimizes both continuous and discrete functions and also multi-objective problems.
- Provides a list of "good" solutions and not just a single solution.
- Always gets an answer to the problem, which gets better over the time.
- Useful when the search space is very large and there are a large number of parameters involved.

Limitations of GAs

Like any technique, GAs also suffer from a few limitations. These include −

- GAs are not suited for all problems, especially problems which are simple and for which derivative information is available.
- Fitness value is calculated repeatedly which might be computationally expensive for some problems.
- Being stochastic, there are no guarantees on the optimality or the quality of the solution.
- If not implemented properly, the GA may not converge to the optimal solution.

GA – Motivation

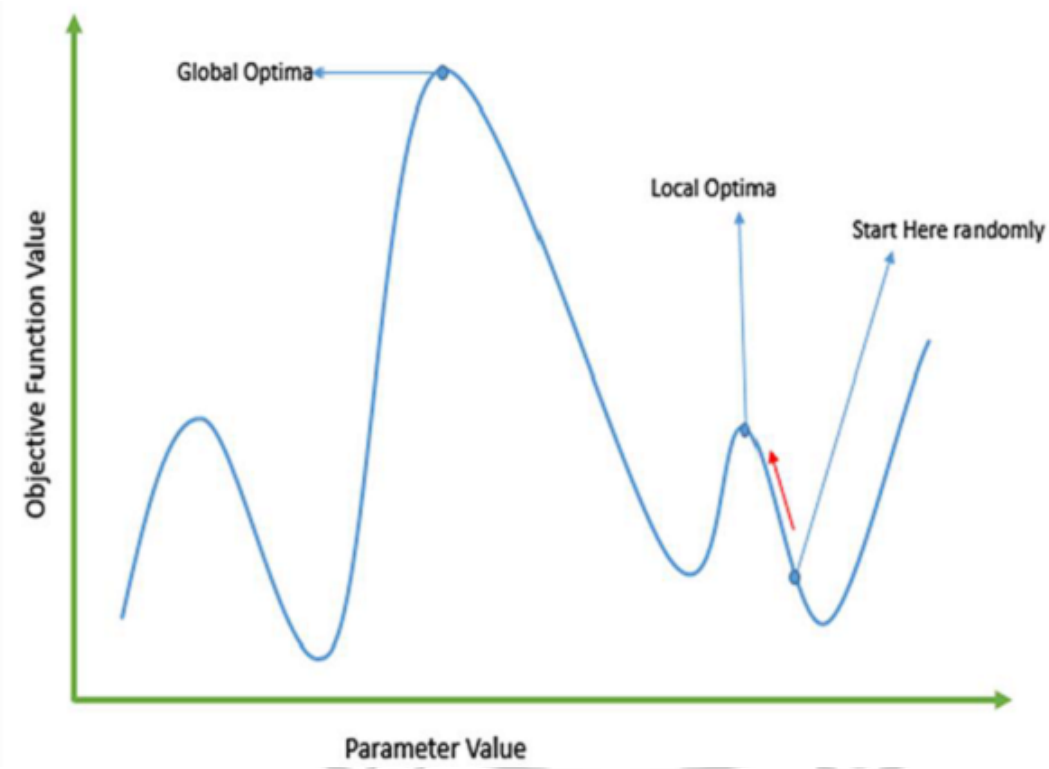
Genetic Algorithms have the ability to deliver a "good-enough" solution "fast-enough". This makes genetic algorithms attractive for use in solving optimization problems. The reasons why GAs are needed are as follows −

Solving Difficult Problems

In computer science, there is a large set of problems, which are **NP-Hard**. What this essentially means is that, even the most powerful computing systems take a very long time (even years!) to solve that problem. In such a scenario, GAs prove to be an efficient tool to provide **usable near-optimal solutions** in a short amount of time.

Failure of Gradient Based Methods

Traditional calculus based methods work by starting at a random point and by moving in the direction of the gradient, till we reach the top of the hill. This technique is efficient and works very well for single-peaked objective functions like the cost function in linear regression. But, in most real-world situations, we have a very complex problem called as landscapes, which are made of many peaks and many valleys, which causes such methods to fail, as they suffer from an inherent tendency of getting stuck at the local optima as shown in the following figure.



Getting a Good Solution Fast

Some difficult problems like Travelling Salesperson Problem (TSP), have real-world applications like path finding and VLSI Design. Now imagine that you are using your GPS Navigation system, and it takes a few minutes (or even a few hours) to compute the "optimal" path from the source to destination. Delay in such real world applications is not acceptable and therefore a "good-enough" solution, which is delivered "fast" is what is required.

Algorithm:

As we now know they're based on the process of natural selection, this means they take the fundamental properties of natural selection and apply them to whatever problem it is we're trying to solve. The basic process for a genetic algorithm is:

1. **Initialization** - Create an initial population. This population is usually randomly generated and can be any desired size, from only a few individuals to thousands.
2. **Evaluation** - Each member of the population is then evaluated and we calculate a 'fitness' for that individual. The fitness value is calculated by how well it fits with our desired requirements. These requirements could be simple, 'faster algorithms are better', or more complex, 'stronger materials are better but they shouldn't be too heavy'.
3. **Selection** - We want to be constantly improving our populations overall fitness. Selection helps us to do this by discarding the bad designs and only keeping the best individuals in the population. There are a few different selection methods but the basic idea is the same, make it more likely that fitter individuals will be selected for our next generation.
4. **Crossover** - During crossover we create new individuals by combining aspects of our selected individuals. We can think of this as mimicking how sex works in nature. The hope is that by combining certain traits from two or more individuals we will create an even 'fitter' offspring which will inherit the best traits from each of it's parents.
5. **Mutation** - We need to add a little bit randomness into our populations' genetics otherwise every combination of solutions we can create would be in our initial population. Mutation works by making very small changes at random to an individual's genome.

6. And **repeat**! - Now we have our next generation we can start again from step two until we reach a termination condition.

Termination

There are a few reasons why you would want to terminate your genetic algorithm from continuing it's search for a solution. The most likely reason is that your algorithm has found a solution which is good enough and meets a pre-defined minimum criteria. Offer reasons for terminating could be constraints such as time or money.

Example

The example starts by presenting the equation that we are going to implement. The equation is shown below:

```
Y = w1x1 + w2x2 + w3x3 + w4x4 + w5x5 + w6x6
```

The equation has 6 inputs (x_1 to x_6) and 6 weights (w_1 to w_6) as shown and inputs values are (x_1, x_2, x_3, x_4, x_5, x_6)=(4,-2,7,5,11,1). We are looking to find the parameters (weights) that maximize such equation. The idea of maximizing such equation seems simple. The positive input is to be multiplied by the largest possible positive number and the negative number is to be multiplied by the smallest possible negative number. But the idea we are looking to implement is how to make GA do that its own in order to know that it is better to use positive weight with positive inputs and negative weights with negative inputs. Let us start implementing GA.

First we implement the GA module defined in file GA.py:

```
import numpy
def cal_pop_fitness(equation_inputs, pop):
    # Calculating the fitness value of each solution in the current
population.
    # The fitness function caulcuates the sum of products between each
input and its corresponding weight.
    fitness = numpy.sum(pop*equation_inputs, axis=1)
    return fitness

def select_mating_pool(pop, fitness, num_parents):
    # Selecting the best individuals in the current generation as
parents for producing the offspring of the next generation.
    parents = numpy.empty((num_parents, pop.shape[1]))
    for parent_num in range(num_parents):
        max_fitness_idx = numpy.where(fitness == numpy.max(fitness))
        max_fitness_idx = max_fitness_idx[0][0]
        parents[parent_num, :] = pop[max_fitness_idx, :]
        fitness[max_fitness_idx] = -99999999999
    return parents
```

```
def crossover(parents, offspring_size):
    offspring = numpy.empty(offspring_size)
    # The point at which crossover takes place between two parents.
Usually it is at the center.
    crossover_point = numpy.uint8(offspring_size[1]/2)

    for k in range(offspring_size[0]):
        # Index of the first parent to mate.
        parent1_idx = k%parents.shape[0]
        # Index of the second parent to mate.
        parent2_idx = (k+1)%parents.shape[0]
        # The new offspring will have its first half of its genes taken
from the first parent.
        offspring[k, 0:crossover_point] = parents[parent1_idx,
0:crossover_point]
        # The new offspring will have its second half of its genes
taken from the second parent.
        offspring[k, crossover_point:] = parents[parent2_idx,
crossover_point:]
    return offspring

def mutation(offspring_crossover):
    # Mutation changes a single gene in each offspring randomly.
    for idx in range(offspring_crossover.shape[0]):
        # The random value to be added to the gene.
        random_value = numpy.random.uniform(-1.0, 1.0, 1)
        offspring_crossover[idx, 4] = offspring_crossover[idx, 4] +
random_value
    return offspring_crossover
```

Then using numpy and GA we can implement the system for our sample equation.

```
import numpy
import ga

"""
The y=target is to maximize this equation ASAP:
    y = w1x1+w2x2+w3x3+w4x4+w5x5+6wx6
    where (x1,x2,x3,x4,x5,x6)=(4,-2,3.5,5,-11,-4.7)
    What are the best values for the 6 weights w1 to w6?
```

```
    We are going to use the genetic algorithm for the best possible
values after a number of generations.
"""

# Inputs of the equation.
equation_inputs = [4,-2,3.5,5,-11,-4.7]

# Number of the weights we are looking to optimize.
num_weights = 6

"""
Genetic algorithm parameters:
    Mating pool size
    Population size
"""
sol_per_pop = 8
num_parents_mating = 4

# Defining the population size.
pop_size = (sol_per_pop,num_weights) # The population will have
sol_per_pop chromosome where each chromosome has num_weights genes.
#Creating the initial population.
new_population = numpy.random.uniform(low=-4.0, high=4.0,
size=pop_size)
print(new_population)

num_generations = 5
for generation in range(num_generations):
    print("Generation : ", generation)
    # Measing the fitness of each chromosome in the population.
    fitness = ga.cal_pop_fitness(equation_inputs, new_population)

    # Selecting the best parents in the population for mating.
    parents = ga.select_mating_pool(new_population, fitness,
                                    num_parents_mating)

    # Generating next generation using crossover.
    offspring_crossover = ga.crossover(parents,
offspring_size=(pop_size[0]-parents.shape[0], num_weights))
```

```
# Adding some variations to the offsrping using mutation.
      offspring_mutation = ga.mutation(offspring_crossover)
    # Creating the new population based on the parents and offspring.
    new_population[0:parents.shape[0], :] = parents
    new_population[parents.shape[0]:, :] = offspring_mutation
    # The best result in the current iteration.
    print("Best result : ",
numpy.max(numpy.sum(new_population*equation_inputs, axis=1)))

# Getting the best solution after iterating finishing all generations.
  #At first, the fitness is calculated for each solution in the final
  generation.
fitness = ga.cal_pop_fitness(equation_inputs, new_population)
# Then return the index of that solution corresponding to the best
fitness.
best_match_idx = numpy.where(fitness == numpy.max(fitness))
print("Best solution : ", new_population[best_match_idx, :])
print("Best solution fitness : ", fitness[best_match_idx])
```

7.2. Exercises

Exercise 7.1:

For the exercise given in the previous lab, formulate the problem using genetic algorithms and apply genetic algorithm to solve the equation.

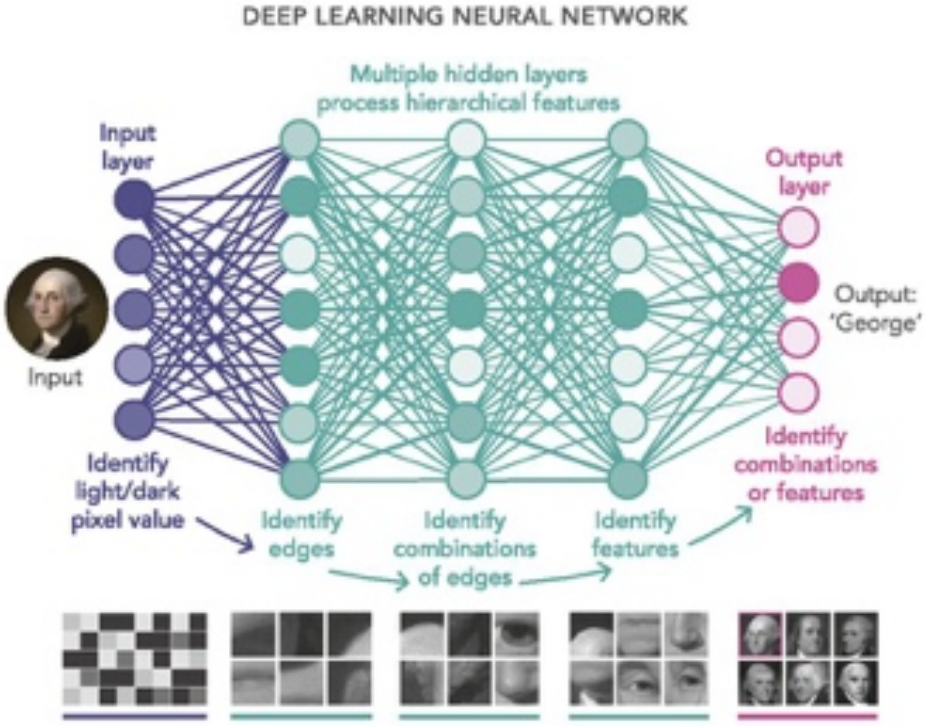
The question in the last lab was:
Find the minimum to the objective function

$$obj = 0.2 + x_1^2 + x_2^2 - 0.1\ \cos(6\pi x_1) - 0.1\cos(6\pi x_2)$$

by adjusting the values of x_1 and x_2.

CS351-L – Introduction to Artificial Intelligence Lab 8

SUPERVISED LEARNING – NEURAL NETWORKS



Contents: Dr. Sajid Anwar

Ghulam Ishaq Khan Institute of Engineering Sciences

Faculty of Computer Science & Engineering

CS351L – Introduction to Artificial Intelligence Lab

Lab 8 – Supervised Learning I

Neural Networks with and without NNT

Objective

The objective of this session is to

- Introduce students to neural networks, using neural network tool box (NNT) and also without NNT.
- Implementation of neural networks in MATLAB involving creation of neural network, training and simulation.

Learning outcomes

After the successful completion of this lab, the student will be able to:

1. Explain the basics of neural networks.
2. Apply and implement solutions for classification problems using neural networks through NNT toolbox as well as without NNT.

Instructions

- Read through the handout sitting in front of a computer that has a Python software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering the problems, indicate the commands you entered and output displayed.
- It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document.

Table of Contents

8.1. NEURAL NETWORK TOOLBOX (NNT)

Matlab's Neural Network Toolbox (NNT) is powerful, yet at times completely incomprehensible. This is mainly due to the complexity of the network object. Even though high-level network creation functions, like **newp** and **newff**, are included in the Toolbox, there will probably come a time when it will be necessary to directly edit the network object properties.

Part of my job is teaching a neural networks practicum. Since we wanted the students to concern themselves with the ideas behind neural networks rather than with implementation and programming issues, some software was written to hide the details of the network object behind a Matlab GUI. In the course of writing this software, I learned a lot about the NNT. Some of it I learned from the manual, but most of it I learned through trial-and-error.

And that's the reason I wrote this introduction. In order to save you a lot of time I already had to spent learning about the NNT. This document is far from extensive, and is naturally restricted to my own field of application. Therefore, only feed-forward networks will be treated. I do think however that reading this can give you a firm enough background to start building your own custom networks, relying on the Matlab documentation for specific details.

All Matlab commands given in this document assume the existence of a NNT network object named `net'. To construct such an object from scratch, type

```
>> net = network;
```

which gives you a `blank' network, i.e., without any properties. Which properties to set and how to set them is the subject of this document.

8.1.1. The Network Layers

The term `layer' in the neural network sense means different things to different people. In the NNT, a layer is defined as a layer of neurons, with the exception of the input layer. So, in NNT terminology this would be a one-layer network.

We will use the last network as an example throughout the text. Each layer has a number of properties, the most important being the transfer functions of the neurons in that layer, and the function that defines the net input of each neuron given its weights and the output of the previous layer.

Constructing Layers

You have an empty network object named `net' in your workspace, if not, type

```
>> net = network;
```

to get one.

Let's start with defining the properties of the input layer. The NNT supports networks which have multiple input layers. I've never used such networks, and don't know of anybody who has, so let's set this to 1:

```
>> net.numInputs = 1;
```

Now we should define the number of neurons in the input layer. This should of course be equal to the dimensionality of your data set. The appropriate property to set is **net.inputs{i}.size**, where i is the index of the input layers. So to make a network which has 2 dimensional points as inputs, type:

```
>> net.inputs{1}.size = 2;
```

This defines (for now) the input layer.

The next properties to set are **net.numLayers**, which not surprisingly sets the total number of layers in the network, and **net.layers{i}.size**, which sets the number of neurons in

the ith layer. To build our example network, we define 2 extra layers (a hidden layer with 3 neurons and an output layer with 1 neuron), using:

```
>> net.numLayers = 2;
>> net.layers{1}.size = 3;
>> net.layers{2}.size = 1;
```

Connecting Layers

Now it's time to define which layers are connected. First, define to which layer the inputs are connected by setting net.inputConnect(i) to 1 for appropriate layer i (usually first, so i=1).

The connections between the rest of the layers are defined a connectivity matrix called net.layerConnect, which can have either 0 or 1 as element entries. If element (i,j) is 1, then the outputs of layer j are connected to the inputs of layer i.

We also have to define which layer is the output layer by setting net.outputConnect(i) to 1 for the appropriate layer i.

Finally, if we have a supervised training set, we also have to define which layers are connected to the target values. (Usually, this will be the output layer.) This is done by setting net.targetConnect(i) to 1 for the appropriate layer i. So, for our example, the appropriate commands would be

```
>> net.inputConnect(1) = 1;
>> net.layerConnect(2, 1) = 1;
>> net.outputConnect(2) = 1;
>> net.targetConnect(2) = 1;
```

Setting Transfer Functions

Each layer has its own transfer function which is set through the **net.layers{i}.transferFcn** property. So to make the first layer use sigmoid transfer functions, and the second layer linear transfer functions, use

```
>> net.layers{1}.transferFcn = 'logsig';
>> net.layers{2}.transferFcn = 'purelin';
```

For a list of possible transfer functions, check the Matlab documentation.

Weights and Biases

Now, define which layers have biases by setting the elements of **net.biasConnect** to either 0 or 1, where **net.biasConnect(i)** = 1 means layer i has biases attached to it. To attach biases to each layer in our example network, we'd use

```
>> net.biasConnect = [ 1 ; 1];
```

Now you should decide on an initialization procedure for the weights and biases. When done correctly, you should be able to simply issue a

```
>> net = init(net);
```

to reset all weights and biases according to your choices. The first thing to do is to set **net.initFcn**. Unless you have built your own initialization routine, the value 'initlay' is the way to go. This let's each layer of weights and biases use their own initialisation routine to initialise.

```
>> net.initFcn = 'initlay';
```

Exactly which function this is should of course be specified as well. This is done through the property **net.layers{i}.initFcn** for each layer. The two most practical options here are Nguyen-Widrow initialisation ('initnw', type 'help initnw' for details), or 'initwb', which let's you choose the initialisation for each set of weights and biases separately. When using 'initnw' you only have to set

```
>> net.layers{i}.initFcn = 'initnw';
```

for each layer i and you're done.

When using 'initwb', you have to specify the initialisation routine for each set of weights and biases separately. The most common option here is 'rands', which sets all weights or biases to a random number between -1 and 1. First, use

```
>> net.layers{i}.initFcn = 'initwb';
```

for each layer i. Next, define the initialisation for the input weights,

```
>> net.inputWeights{1,1}.initFcn = 'rands';
```

and for each set of biases

```
>> net.biases{i}.initFcn = 'rands';
```

and weight matrices

```
>> net.layerWeights{i,j}.initFcn = 'rands';
```

where **net.layerWeights{i,j}** denotes the weights from layer j to layer i.

8.1.2. Training Functions and Parameters

The difference between train and adapt

One of the more counterintuitive aspects of the NNT is the distinction between **train** and **adapt**. Both functions are used for training a neural network, and most of the time both can be used for the same network.

What then is the difference between the two? The most important one has to do with incremental training (updating the weights after the presentation of each single training sample) versus batch training (updating the weights after each presenting the complete data set).

When using **adapt**, both incremental and batch training can be used. Which one is actually used depends on the format of your training set. If it consists of two matrices of input and target vectors, like

```
>> P = [ 0.3 0.2 0.54 0.6 ; 1.2 2.0 1.4 1.5]
P =
        0.3000 0.2000 0.5400 0.6000
        1.2000 2.0000 1.4000 1.5000
>> T = [ 0 1 1 0 ]
T =  0 1 1 0
```

the network will be updated using batch training. (In this case, we have 4 samples of 2 dimensional input vectors, and 4 corresponding 1D target vectors). If the training set is given in the form of a cell array,

```
>> P = {[0.3 ; 1.2] [0.2 ; 2.0] [0.54 ; 1.4] [0.6 ; 1.5]}
P =
        [2x1 double] [2x1 double] [2x1 double] [2x1 double]
>> T = { [0] [1] [1] [0] }
T =  [0] [1] [1] [0]
```

then incremental training will be used.

When using **train** on the other hand, only batch training will be used, regardless of the format of the data (you can use both).

The big plus of **train** is that it gives you a lot more choice in training functions (gradient descent, gradient descent w/ momentum, Levenberg-Marquardt, etc.) which are implemented very efficiently. So, when you don't have a good reason for doing incremental training, **train** is probably your best choice. (And it usually saves you setting some parameters).

To conclude this section difference between **train** and **adapt** is, *the difference between passes and epochs*. When using **adapt**, the property that determines how many times the complete training data set is used for training the network is called **net.adaptParam.passes**.

Fair enough. But, when using **train**, the exact same property is now called **net.trainParam.epochs**.

Performance Functions

The two most common options here are the Mean Absolute Error (**mae**) and the Mean Squared Error (**mse**). The mae is usually used in networks for classification, while the mse is most commonly seen in function approximation networks.
The performance function is set with the **net.performFcn** property, for instance:

```
>> net.performFcn = 'mse';
```

Train Parameters

If you are going to train your network using **train**, the last step is defining **net.trainFcn**, and setting the appropriate parameters in **net.trainParam**. Which parameters are present depends on your choice for the training function.

So if you for example want to train your network using a Gradient Descent w/ Momentum algorithm, you'd set

```
>> net.trainFcn = 'traingdm';
```

and then set the parameters

```
>> net.trainParam.lr = 0.1;
>> net.trainParam.mc = 0.9;
```

to the desired values. (In this case, **lr** is the learning rate, and **mc** the momentum term.)

Check the Matlab documentation for possible training functions and their parameters. Two other useful parameters are **net.trainParam.epochs**, which is the maximum number of times the complete data set may be used for training, and **net.trainParam.show**, which is the time between status reports of the training function. For example,

```
>> net.trainParam.epochs = 1000;
>> net.trainParam.show = 100;
```

Adapt Parameters

The same general scheme is also used in setting **adapt** parameters. First, set **net.adaptFcn** to the desired adaptation function. We'll use **adaptwb** (from 'adapt weights and biases'), which allows for a separate update algorithm for each layer. Again, check the Matlab documentation for a complete overview of possible update algorithms.

```
>> net.adaptFcn = 'adaptwb';
```

Next, since we're using **adaptwb**, we'll have to set the learning function for all weights and biases:

```
>> net.inputWeights{1,1}.learnFcn = 'learnp';
>> net.biases{1}.learnFcn = 'learnp';
```

where in this example we've used **learnp**, the Perceptron learning rule. (Type 'help learnp', etc.).

Finally, a useful parameter is **net.adaptParam.passes**, which is the maximum number of times the complete training set may be used for updating the network:

```
>> net.adaptParam.passes = 10;
```

8.2. NEURAL NETWORK WITHOUT TOOLBOX

8.2.1. Introduction:

Matlab has a suite of programs designed to build neural networks (the Neural Networks Toolbox). Additionally, there are demonstrations available through Matlab's help feature. There are three steps to using neural networks:

- Design,
- Training, and
- Testing (the same for modeling in general!).

Design

- Define the number of nodes in each of the three layers (input, hidden, output).
- Define what σ is for each of the nodes (usually all nodes in a layer will all use the same σ). These are called the transfer functions.
- Tell Matlab what optimization (or training) routine to use. Generally, we will use either traingdx, which is gradient descent, or trainlm (for Levenburg-Marquardt, which is a combination of gra-dient descent and Newton's Method).
- Optional parts of the design: Error function (Mean Square Error is the default), plot the progress of training, etc.

Training

- Once the network has been designed, we "train" the network (by optimizing the error function).
- This process determines the "best" set of weights and biases for our data set.

Testing

- We need to test our network to see if it has found a good balance between memorization (accuracy) and generalization.

8.2.2. Examples

Example 1:

Generate ANDNOT function using neural net by a MATLAB program.

Solution

The truth table for the ANDNOT function is as follows:

X_1	X_2	Y
0	0	0
0	1	0
1	0	1
1	1	0

The MATLAB program is given by,

```
%Getting weights and threshold value
disp('Enter weights');
w1 = input('Weight w1=');
w2 = input('weight w2=');
disp('Enter Threshold Value');
theta = input('theta=');
y = [0 0 0 0];
x1 = [0 0 1 1];
x2 = [0 1 0 1];
z = [0 0 1 0];
```

```
con = 1;
while con
        zin = x1 * w1 + x2 * w2;
        for i=1:4
                if zin(i)>=theta
                        y(i)=1;
                else
                        y(i)=0;
                end
        end
        disp('Output of Net');
        disp(y);
        if y == z
                con = 0;
        else
                disp('Net is not learning enter another set of weights and
Threshold value');
                w1 = input('weight w1=');
                w2 = input('weight w2=');
                theta = input('theta=');
        end
end
disp('Mcculloch-Pitts Net for ANDNOT function');
disp('Weights of Neuron');
disp(w1);
disp(w2);
disp('Threshold value');
disp(theta);
```

Output

```
Enter weights
Weight w1=1
weight w2=1
Enter Threshold Value
theta=0.1
Output of Net
0 1 1 1
Net is not learning enter another set of weights and Threshold value
Weight w1=1
Weight w2=-1
theta=1
Output of Net
0 0 1 0
Mcculloch-Pitts Net for ANDNOT function
Weights of Neuron
1
-1
Threshold value
1
```

Example 2:

Generate XOR function.

Solution

The truth table for the XOR function is,

X_1	X_2	Y
0	0	0
0	1	1
1	0	1
1	1	0

The MATLAB program is given by,

```
%Getting weights and threshold value
disp('Enter weights');
w11 = input('Weight w11=');
w12 = input('weight w12=');
w21 = input('Weight w21=');
w22 = input('weight w22=');
v1 = input('weight v1=');
v2 = input('weight v2=');
disp('Enter Threshold Value');
theta = input('theta=');
x1 = [0 0 1 1];
x2 = [0 1 0 1];
z = [0 1 1 0];
con = 1;
while con
        zin1 = x1 * w11 + x2 * w21;
        zin2 = x1 * w21 + x2 * w22;
        for i = 1:4
                if zin1(i) >= theta
                        y1(i) = 1;
                else
                        y1(i) = 0;
                end
                if zin2(i) >= theta
                        y2(i) = 1;
                else
                        y2(i) = 0;
                end
        end
        yin = y1 * v1 + y2 * v2;
        for i = 1:4
                if yin(i) >= theta;
                        y(i) = 1;
                else
                        y(i) = 0;
                end
        end
        disp('Output of Net');
        disp(y);
        if y == z
                con = 0;
        else
                disp('Net is not learning enter another set of weights and
Threshold value');
                w11 = input('Weight w11=');
                w12 = input('weight w12=');
```

```
                w21 = input('Weight w21=');
                w22 = input('weight w22=');
                v1 = input('weight v1=');
                v2 = input('weight v2=');
                theta = input('theta=');
        end
end
```

8.3. Exercises

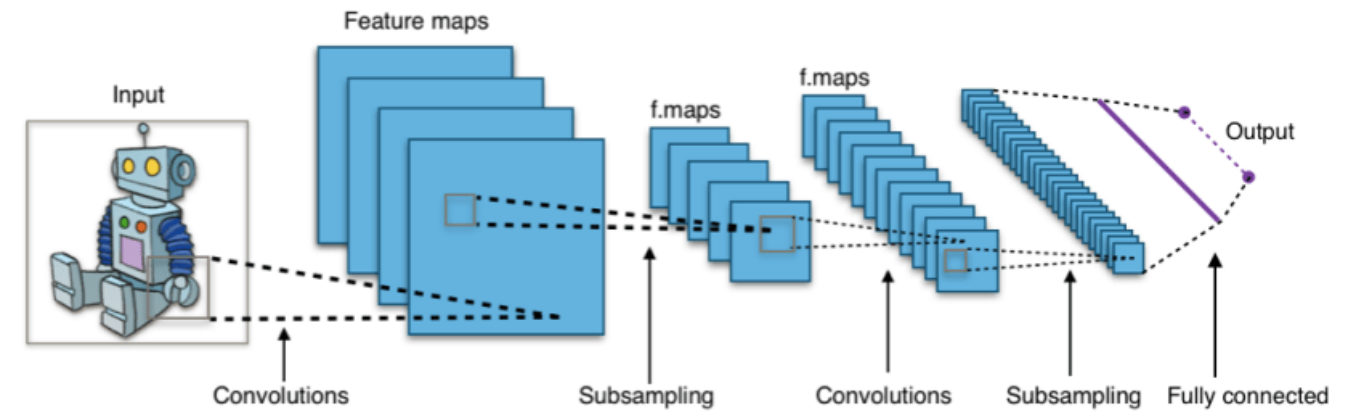
Exercise 8.1.

Exercise 8.2.

Exercise 8.3.

CS351-L – Introduction to Artificial Intelligence Lab 9

SUPERVISED LEARNING – CONVOLUTIONAL NEURAL NETWORKS



Contents: Dr. Sajid Anwar

Ghulam Ishaq Khan Institute of Engineering Sciences

Faculty of Computer Science & Engineering

CS351L – Introduction to Artificial Intelligence Lab

Lab 9 – Supervised Learning II

Introduction to Convolutional Neural Networks

Objective

The objective of this session is to understand the concepts of Convolution, Correlation, Pooling, and Convolutional Neural Networks (CNN).

Learning outcomes

After the successful completion of this lab, the student will be able to:

1. Understand and explain layers, feature maps, max pooling, and fully connected layers.
2. Train the specified CNN architecture for the MNIST handwritten digit recognition problem and report the training and inference results with training, validation, and test sets.
3. Perform inference with a pre-trained model to understand importance of transfer learning.

Instructions

- Read through the handout sitting in front of a computer that has a Python software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering the problems, indicate the commands you entered and output displayed.
- It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document.

Table of Contents

9.1. MATLAB Deep Learning Toolbox

Deep Learning Toolbox™ (formerly Neural Network Toolbox™) provides a framework for designing and implementing deep neural networks with algorithms, pre-trained models, and apps. You can use convolutional neural networks (ConvNets, CNNs) and long short-term memory (LSTM) networks to perform classification and regression on image, time-series, and text data. Apps and plots help you visualize activations, edit network architectures, and monitor training progress.

For small training sets, you can perform transfer learning with pre-trained deep network models (ResNet-101, GoogLeNet, and VGG-19 etc.) and models imported from TensorFlow™-Keras and Caffe.

9.1.1. Convolutional NEURAL NETWORK

Convolutional neural network (CNN) is a very important technique primarily used for image recognition, machine vision, and other natural signals. CNN is major breakthrough in deep learning field which achieved human level accuracy on many computer vision problems. CNN is a neuro-morphic computational model inspired from mammal's brain. The main inspiration comes from Hubel and Wiesel's work. Experiments on monkeys revealed that the visual area contains two types of cells: simple cells responsible for feature extraction and complex cells combining these features locally. Yann Lecun et al. developed handwritten digit recognition CNN called LeNet-5. A sample CNN network is depicted in Fig. 1.

Unlike ANN, CNN is divided into two parts. The first part performs feature extraction from the input image while the second part acts as classifier. CNN has three types of layers: convolution, pooling and fully connected DNN layers. Convolution layers exhibit local connectivity and weights sharing. The pooling layer performs subsampling on k x k (e.g., k = 2) region from the preceding layer. The operation can be average, max or stochastic pooling. Conceptually CNN can be divided into two parts. The frontal part learns useful features for classification while rear part is a multilayer fully connected deep neural network (DNN).

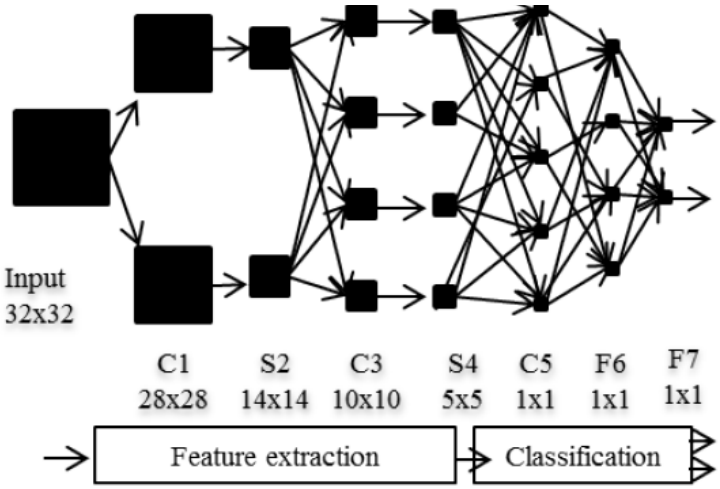
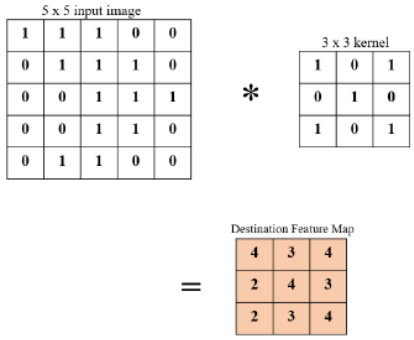


Fig. 1. Convolutional neural network with eight layers. The prefix "C", "S" and "F" stands for convolution, subsampling and classifier layers respectively. Layers C1 to C5 and C5 to F7 constitute the frontal feature extraction and classifier parts respectively. The proposed work represents the depicted architecture with a string of 1-2-2-4-4-5-4-2 where each number denotes the count of feature maps in that layer.

Convolution Layer

These layers are the core building blocks of CNN where most of the computations are conducted. It maintains pixel relationship by learning features using learnable weight matrices called kernels or filters. The input image is convolved with the kernel and the resulting signal is called feature map. The convolutional layer in CNN learn features which are important for classification. It is important to note that correlation is different than convolution. The kernel needs to be flipped vertically and horizontally before computing the dot product. Figure 1.3 shows a simple convolutional layer of CNN. 5x5 image is convolved with 3x3 kernel. The resulting signal, feature map, size is 3x3.



Simple Convolutional Layer of CNN

9.1.2. Create Simple Deep Learning Network for Classification

This example shows how to create and train a simple convolutional neural network for deep learning classification. Convolutional neural networks are essential tools for deep learning, and are especially suited for image recognition.

The example demonstrates how to:

- Load and explore image data.
- Define the network architecture.
- Specify training options.
- Train the network.
- Predict the labels of new data and calculate the classification accuracy.

Load and explore image data

Load the digit sample data as an image datastore. imageDatastore automatically labels the images based on folder names and stores the data as an ImageDatastore object. An image datastore enables you to store large image data, including data that does not fit in memory, and efficiently read batches of images during training of a convolutional neural network.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos', ...
    'nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true,'LabelSource','foldernames');
```

Calculate the number of images in each category. labelCount is a table that contains the labels and the number of images having each label. The datastore contains 1000 images for each of the digits 0-9, for a total of 10000 images. You can specify the number of classes in the last fully connected layer of your network as the OutputSize argument.

```
labelCount = countEachLabel(imds);
```

You must specify the size of the images in the input layer of the network. Check the size of the first image in digitData. Each image is 28-by-28-by-1 pixels.

Specify Training and Validation Sets

Divide the data into training and validation data sets, so that each category in the training set contains 750 images, and the validation set contains the remaining images from each label. splitEachLabel splits the datastore digitData into two new datastores, trainDigitData and valDigitData.

```
numTrainFiles = 750;
[imdsTrain,imdsValidation] = splitEachLabel(imds,numTrainFiles,'randomize');
```

Define Network Architecture

Define the convolutional neural network architecture.

```
layers = [
    imageInputLayer([28 28 1])

    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,16,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Image Input Layer

An imageInputLayer is where you specify the image size, which, in this case, is 28-by-28-by-1. These numbers correspond to the height, width, and the channel size. The digit data consists of grayscale images, so the channel size (color channel) is 1. For a color image, the channel size is 3, corresponding to the RGB values. You do not need to shuffle the data because trainNetwork, by default, shuffles the data at the beginning of training. trainNetwork can also automatically shuffle the data at the beginning of every epoch during training.

Convolutional Layer

In the convolutional layer, the first argument is filterSize, which is the height and width of the filters the training function uses while scanning along the images. In this example, the number 3 indicates that the filter size is 3-by-3. You can specify different sizes for the height and width of the filter. The second argument is the number of filters, numFilters, which is the number of neurons that connect to the same region of the input. This parameter determines the number of feature maps. Use the 'Padding' name-value pair to add padding to the input feature map. For a convolutional layer with a default stride of 1, 'same' padding ensures that the spatial output size is the same as the input size. You can also define the stride and learning rates for this layer using name-value pair arguments of convolution2dLayer.

Batch Normalization Layer

Batch normalization layers normalize the activations and gradients propagating through a network, making network training an easier optimization problem. Use batch normalization layers between convolutional layers and nonlinearities, such as ReLU layers, to speed up network training and reduce the sensitivity to network initialization. Use batchNormalizationLayer to create a batch normalization layer.

ReLU Layer

The batch normalization layer is followed by a nonlinear activation function. The most common activation function is rectified linear unit (ReLU). Use reluLayer to create a ReLU layer.

Max Pooling Layer

Convolutional layers (with activation functions) are sometimes followed by a down-sampling operation that reduces the spatial size of the feature map and removes redundant spatial information. Down-sampling makes it possible to increase the number of filters in deeper convolutional layers without increasing the required amount of computation per layer. One way of down-sampling is using a max pooling, which you create using maxPooling2dLayer. The max pooling layer returns the maximum values of rectangular regions of inputs, specified by the first argument, poolSize. In this example, the size of the rectangular region is [2,2]. The 'Stride' name-value pair argument specifies the step size that the training function takes as it scans along the input.

Fully Connected Layer

The convolutional and down-sampling layers are followed by one or more fully connected layers. As its name suggests, a fully connected layer is a layer in which the neurons connect to all the neurons in the preceding layer. This layer combines all the features learned by the previous layers across the image to identify the larger patterns. The last fully connected layer combines the features to classify the images. Therefore, the OutputSize parameter in the last fully connected layer is equal to the number of classes in the target data. In this example, the output size is 10, corresponding to the 10 classes. Use fullyConnectedLayer to create a fully connected layer.

Softmax Layer

The softmax activation function normalizes the output of the fully connected layer. The output of the softmax layer consists of positive numbers that sum to one, which can then be used as classification probabilities by the classification layer. Create a softmax layer using the softmaxLayer function after the last fully connected layer.

Classification Layer

The final layer is the classification layer. This layer uses the probabilities returned by the softmax activation function for each input to assign the input to one of the mutually exclusive classes and compute the loss. To create a classification layer, use classificationLayer.

Specify Training Options

After defining the network structure, specify the training options. Train the network using stochastic gradient descent with momentum (SGDM) with an initial learning rate of 0.01. Set the maximum number of epochs to 4. An epoch is a full training cycle on the entire training data set. Monitor the network accuracy during training by specifying validation data and validation frequency. Shuffle the data every epoch. The software trains the network on the training data and calculates the accuracy on the validation data at regular intervals during training. The validation data is not used to update the network weights. Turn on the training progress plot, and turn off the command window output.

```
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.01, ...
    'MaxEpochs',4, ...
    'Shuffle','every-epoch', ...
    'ValidationData',imdsValidation, ...
    'ValidationFrequency',30, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

Train Network Using Training Data

Train the network using the architecture defined by layers, the training data, and the training options. By default, trainNetwork uses a GPU if one is available (requires Parallel Computing Toolbox™ and a CUDA® enabled GPU with compute capability 3.0 or higher). Otherwise, it uses a CPU. You can also specify the execution environment by using the 'ExecutionEnvironment' name-value pair argument of trainingOptions. The training progress plot shows the mini-batch loss and accuracy and the validation loss and accuracy. For more information on the training progress plot, see Monitor Deep Learning Training Progress. The loss is the cross-entropy loss. The accuracy is the percentage of images that the network classifies correctly.

```
net = trainNetwork(imdsTrain, layers, options);
```

Classify Validation Images and Compute Accuracy

Predict the labels of the validation data using the trained network, and calculate the final validation accuracy. Accuracy is the fraction of labels that the network predicts correctly. In this case, more than 99% of the predicted labels match the true labels of the validation set.

```
YPred = classify(net, imdsValidation);
YValidation = imdsValidation.Labels;

Accuracy = sum(YPred == YValidation)/numel(YValidation)
```

accuracy = 0.9924

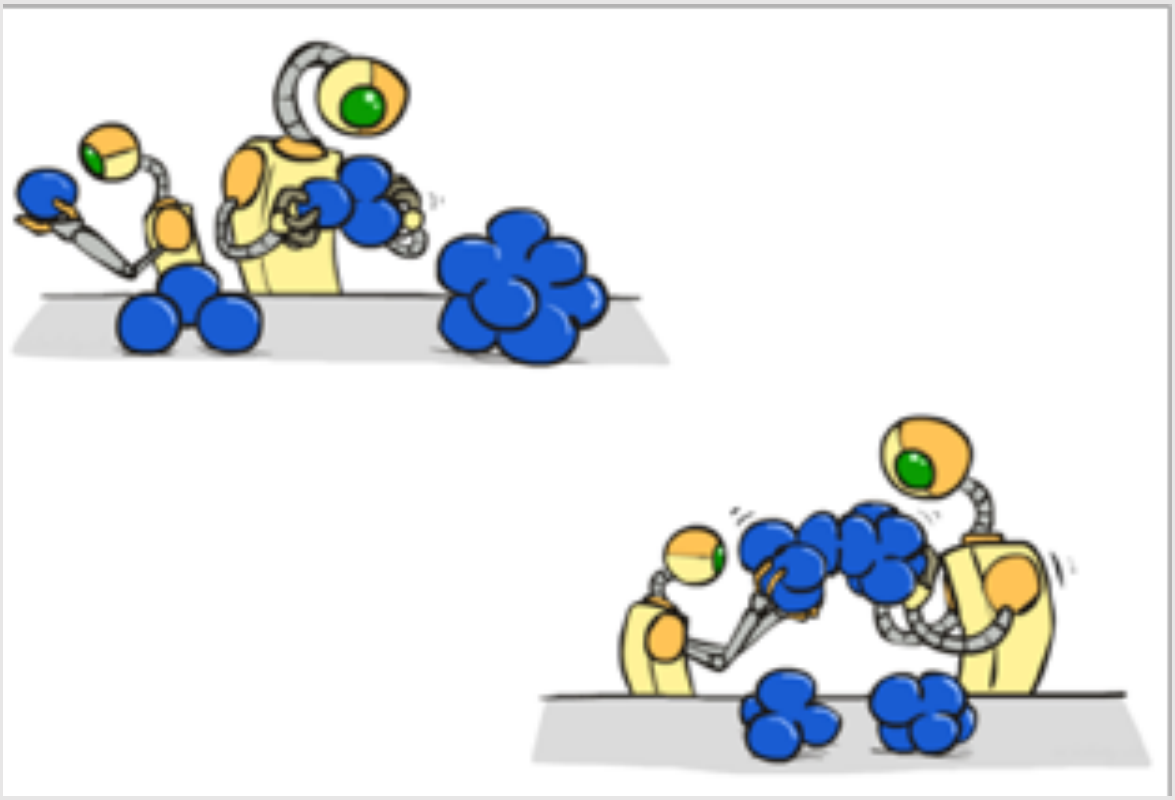
9.2. Exercises

Exercise 9.1.

Exercise 9.2.

CS351-L – Introduction to Artificial Intelligence Lab 10

UNSUPERVISED LEARNING – HIERARCHICAL CLUSTERING



Contents: Dr. Sajid Anwar

Ghulam Ishaq Khan Institute of Engineering Sciences

Faculty of Computer Science & Engineering

CS351L – Introduction to Artificial Intelligence Lab

Lab 10 – Unsupervised Learning I – Hierarchical Clustering

Hierarchical Clustering

Objective

The objective of this session is to

- Introduce students to clustering, especially hierarchical clustering and its types.
- Implementation of hierarchical clustering in MATLAB/Python.

Learning outcomes

After the successful completion of this lab, the student will be able to:

1. Explain the basics of hierarchical clustering.
2. Apply and implement solutions for clustering problems using hierarchical clustering.

Instructions

- Read through the handout sitting in front of a computer that has a Python/Matlab software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering the problems, indicate the commands you entered and output displayed.
- It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document.

Table of Contents

10. UNSUPERVISED LEARNING – HIERARCHICAL CLUSTERING

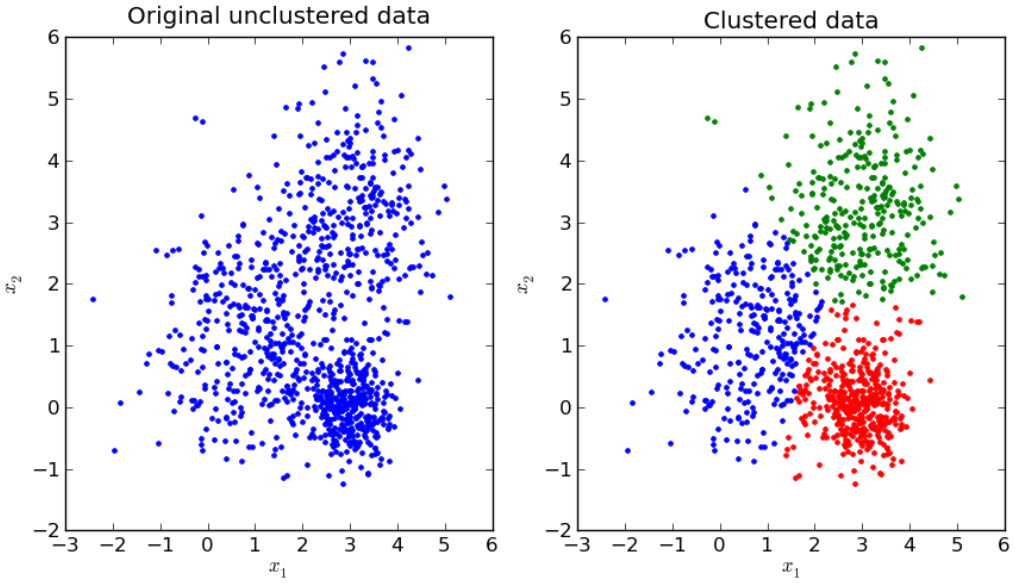
The hierarchical clustering Technique is one of the popular Clustering techniques in Machine Learning. Before we try to understand the concept of the Hierarchical clustering Technique let us understand about the Clustering…

10.1. Clustering – Introduction

Clustering is one of the most common exploratory data analysis technique used to get an intuition about the structure of the data. It can be defined as the task of identifying subgroups in the data such that data points in the same subgroup (cluster) are very similar while data points in different clusters are very different. In other words, we try to find homogeneous subgroups within the data such that data points in each cluster are as similar as possible according to a similarity measure such as euclidean-based distance or correlation-based distance. The decision of which similarity measure to use is application-specific.

Clustering analysis can be done on the basis of features where we try to find subgroups of samples based on features or on the basis of samples where we try to find subgroups of features based on samples. We'll cover here clustering based on features. Clustering is used in market segmentation; where we try to fined customers that are similar to each other whether in terms of behaviors or attributes, image segmentation/compression; where we try to group similar regions together, document clustering based on topics, etc.

Unlike supervised learning, clustering is considered an unsupervised learning method since we don't have the ground truth to compare the output of the clustering algorithm to the true labels to evaluate its performance. We only want to try to investigate the structure of the data by grouping the data points into distinct subgroups.



10.2. Differences between Clustering & Classification/Regression models

In classification and regression models, we are given a data set(D) which contains data points(X_i) and class labels(Y_i). Where, Y_i's belong to {0,1} or {0,1,2,…,n) for Classification models and Y_i's belong to real values for regression models.

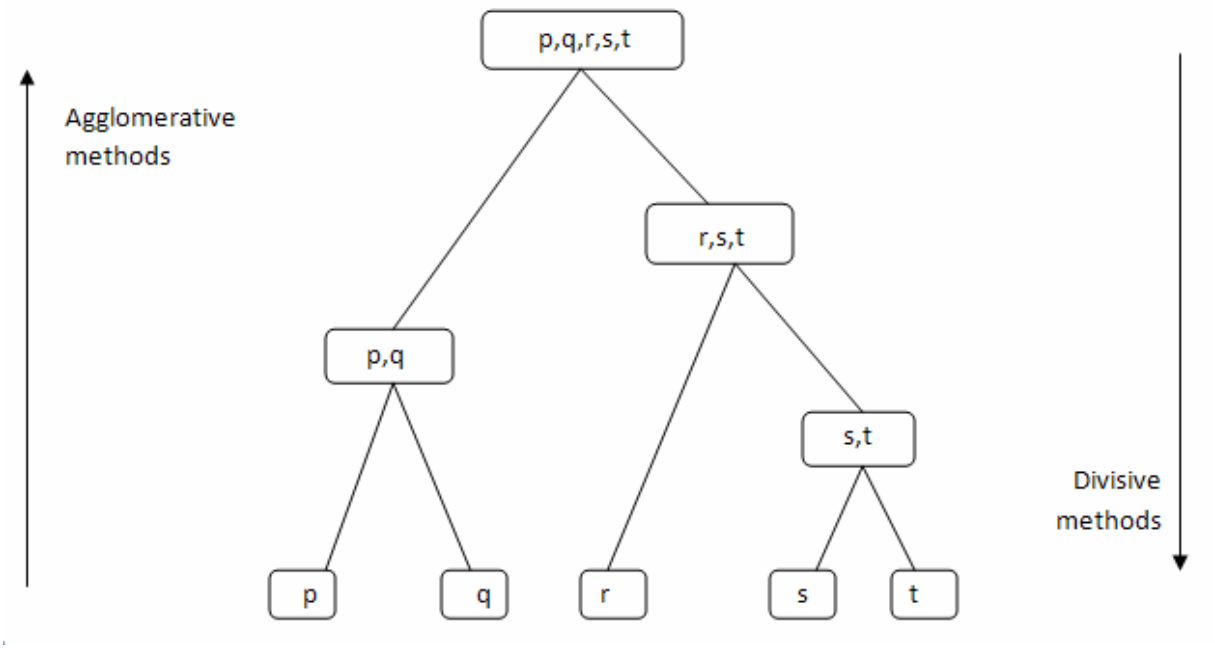
When comes to clustering, we're provided with a data set that contain only data points (X_i). Here we're not provided with the class labels (Y_i).

Now, let's go to our original topic which is the Hierarchical clustering Technique.

10.3. Hierarchical Clustering

Hierarchical clustering algorithms group similar objects into groups called clusters. There are two types of hierarchical clustering algorithms:

- Agglomerative — Bottom up approach. Start with many small clusters and merge them together to create bigger clusters.
- Divisive — Top down approach. Start with a single cluster, then break it up into smaller clusters.

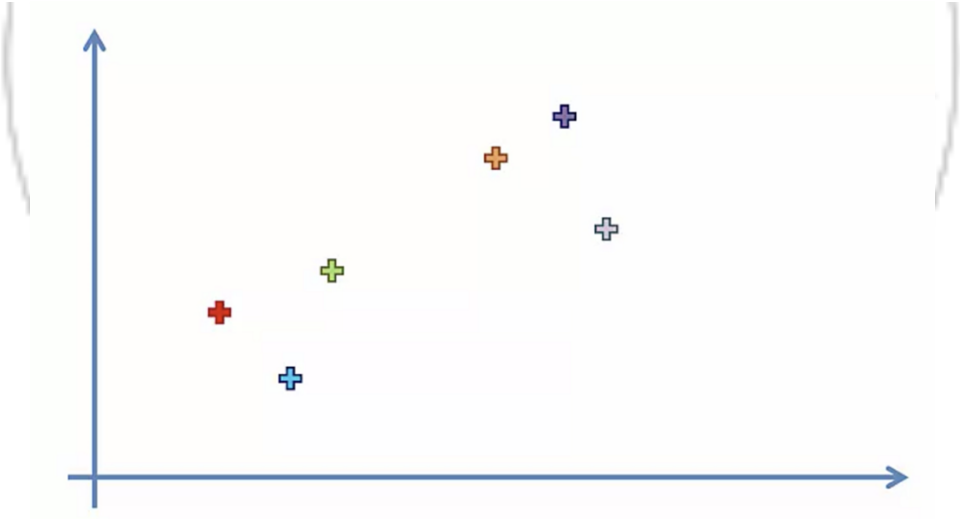


Both these algorithms are exactly reverse of each other. So, we will be covering Agglomerative Hierarchical clustering algorithm in detail.

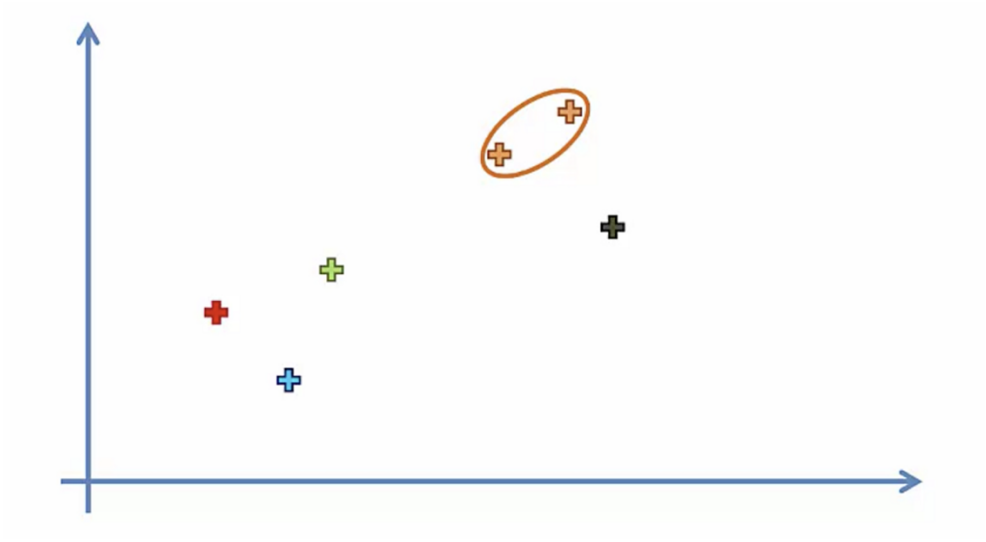
10.4. Agglomerative Hierarchical Clustering Technique

10.4.1. How the algorithm works?

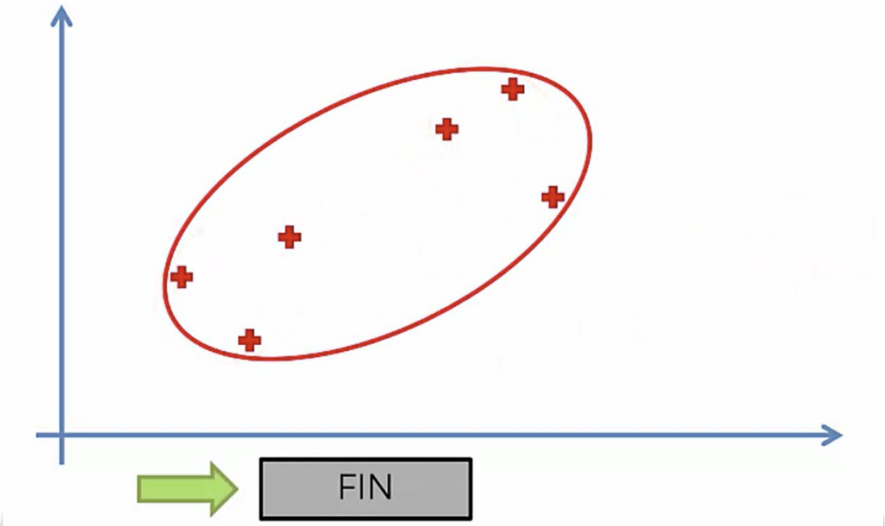
In this technique, initially each data point is considered as an individual cluster.



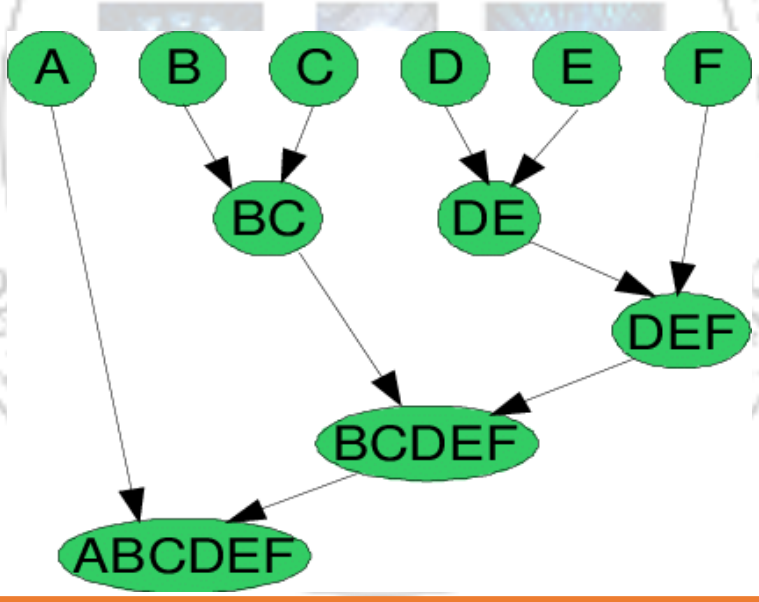
Take the two closest clusters and make them one cluster



Repeat the previous step until there is only one cluster.



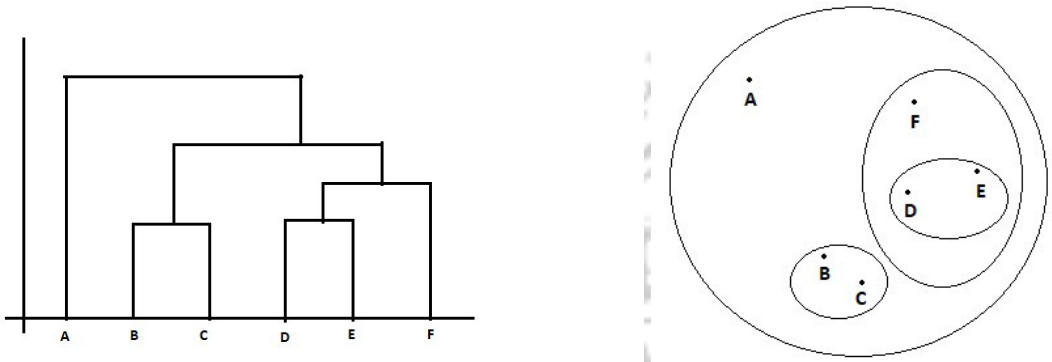
10.4.2. Pictorial Representation of Agglomerative Hierarchical Clustering with Toy Example



10.4.3. Dendrograms

We can use a dendrogram to visualize the history of groupings and figure out the optimal number of clusters.

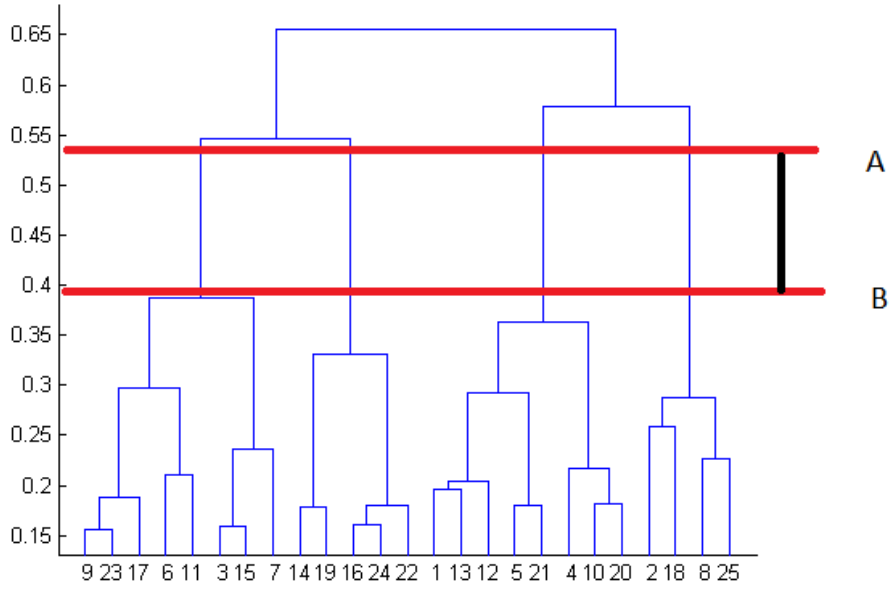
A Dendrogram is a tree-like diagram that records the sequences of merges or splits.



Here is how we can use dendrograms to estimate the number of clusters in the data.

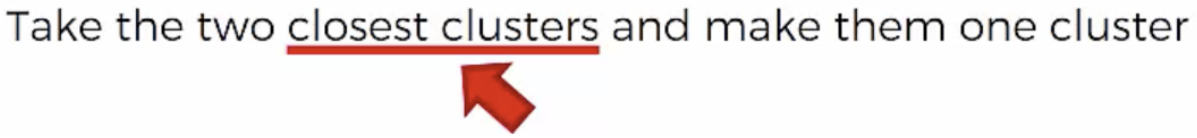
1. Determine the largest vertical distance that doesn't intersect any of the other clusters.
2. Draw a horizontal line at both extremities.
3. The optimal number of clusters is equal to the number of vertical lines going through the horizontal line

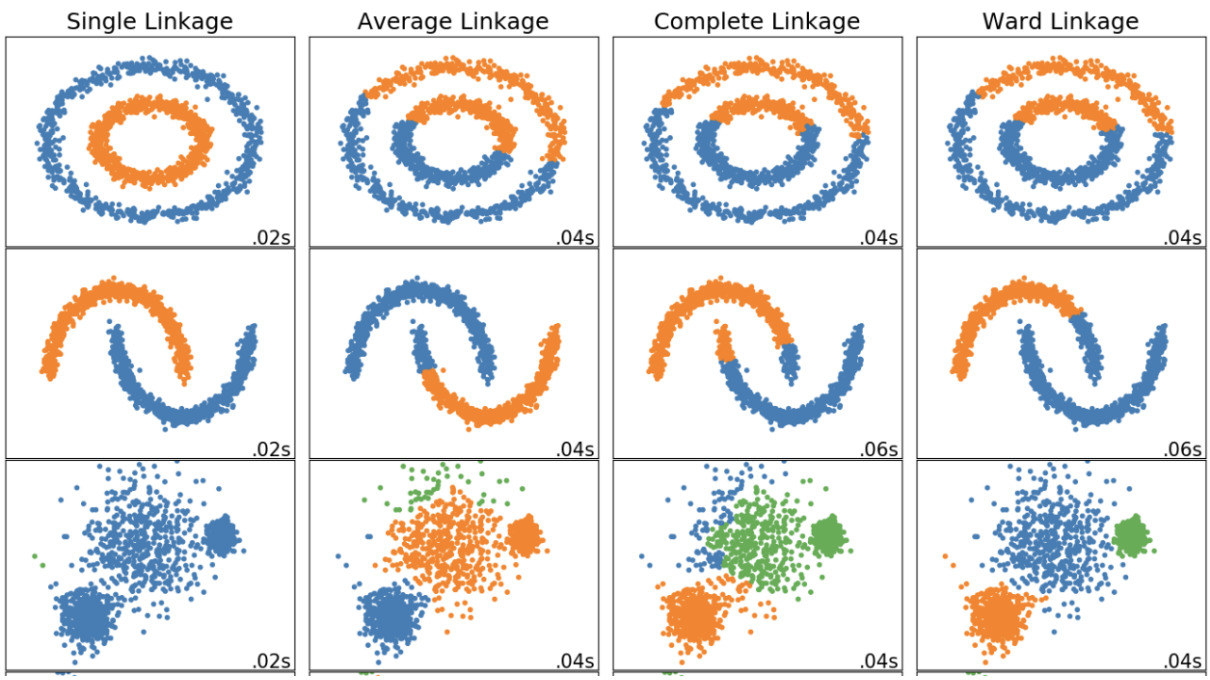
For example, in the following figure, best choice for no. of clusters will be 4.



10.4.4. Linkage criterion

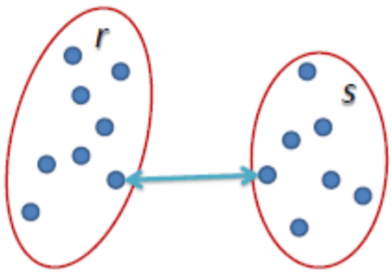
The linkage criterion refers to how the distance between clusters is calculated. Similar to gradient descent, you can tweak the definition of "closest clusters" to get drastically different results.





Single Linkage

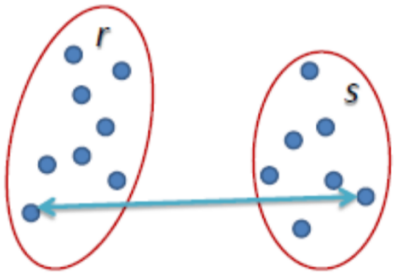
The distance between two clusters is the shortest distance between two points in each cluster. In simple words, pick the two closest points such that one point lies in cluster one and the other point lies in cluster 2 and take their similarity and declare it as the similarity between two clusters.



$$L(r,s) = \min(D(x_{ri}, x_{sj}))$$

Complete Linkage

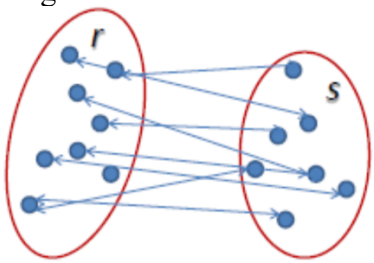
The distance between two clusters is the longest distance between two points in each cluster. In simple words, pick the two farthest points such that one point lies in cluster one and the other point lies in cluster 2 and take their similarity and declare it as the similarity between two clusters.



$$L(r,s) = \max(D(x_{ri}, x_{sj}))$$

Average Linkage

The distance between clusters is the average distance between each point in one cluster to every point in other cluster. In simple words, take all the pairs of points and compute their similarities and calculate the average of the similarities.



$$L(r,s) = \frac{1}{n_r n_s} \sum_{i=1}^{n_r} \sum_{j=1}^{n_s} D(x_{ri}, x_{sj})$$

10.5. Example in python

Let's take a look at a concrete example of how we could go about labelling data using hierarchical agglomerative clustering.

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from sklearn.cluster import AgglomerativeClustering
import scipy.cluster.hierarchy as sch
```

In this tutorial, we use the csv file "Data.csv" containing a list of customers with their gender, age, annual income and spending score.

	CustomerID	Genre	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40
5	6	Female	22	17	76
6	7	Female	35	18	6
7	8	Female	23	18	94
8	9	Male	64	19	3
9	10	Female	30	19	72
10	11	Male	67	19	14

To display our data on a graph at a later point, we can only take two variables (annual income and spending score).

```
dataset = pd.read_csv('./data.csv')
X = dataset.iloc[:, [3, 4]].values
```

Looking at the dendrogram, the highest vertical distance that doesn't intersect with any clusters is the middle green one. Given that 5 vertical lines cross the threshold, the optimal number of clusters is 5.

```
dendrogram = sch.dendrogram(sch.linkage(X, method='average'))
```

We create an instance of AgglomerativeClustering using the euclidean distance as the measure of distance between points and average linkage to calculate the proximity of clusters.

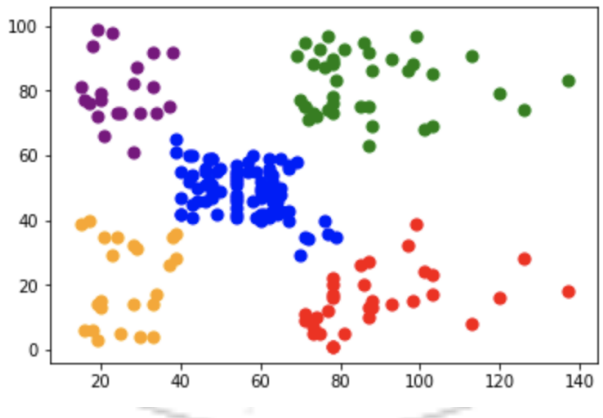
```
model = AgglomerativeClustering(n_clusters=5, affinity='euclidean',
linkage='average')
model.fit(X)
labels = model.labels_
```

The labels_ property returns an array of integers where the values correspond to the distinct categories.

```
array([4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3,
       4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 1,
       4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 1, 2, 0, 2, 0, 2,
       1, 2, 0, 2, 0, 2, 0, 2, 0, 2, 1, 2, 0, 2, 1, 2, 0, 2, 0, 2, 0, 2,
       0, 2, 0, 2, 0, 2, 1, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2,
       0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2,
       0, 2])
```

We can use a shorthand notation to display all the samples belonging to a category as a specific color.

```
plt.scatter(X[labels==0, 0], X[labels==0, 1], s=50, marker='o',
color='red')
plt.scatter(X[labels==1, 0], X[labels==1, 1], s=50, marker='o',
color='blue')
plt.scatter(X[labels==2, 0], X[labels==2, 1], s=50, marker='o',
color='green')
plt.scatter(X[labels==3, 0], X[labels==3, 1], s=50, marker='o',
color='purple')
plt.scatter(X[labels==4, 0], X[labels==4, 1], s=50, marker='o',
color='orange')
plt.show()
```



10.6. Exercises

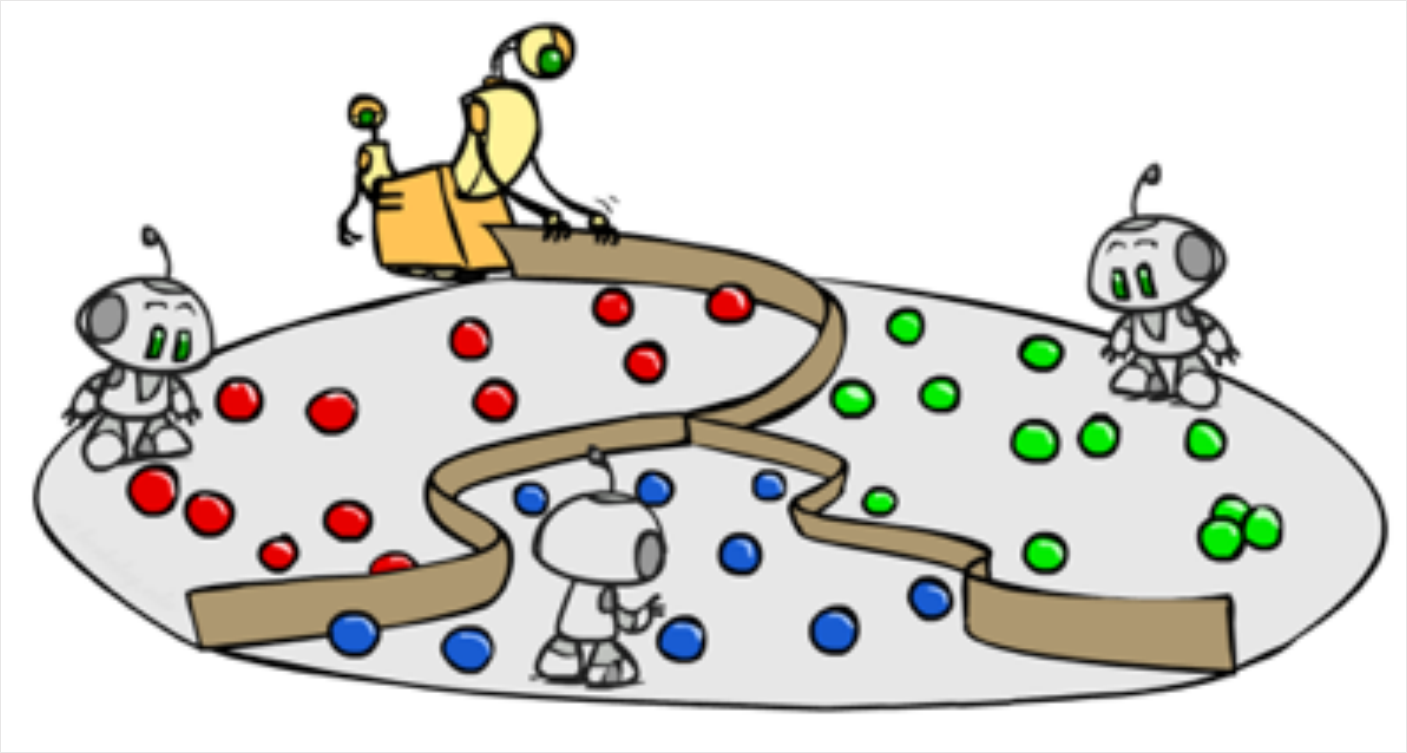
Exercise 10.1.

With the given Data file, write your own program that performs clustering on the data **without using scipy and sklearn** libraries. Your program should take as input the Data file, process it to extract the two variables (annual income and spending score), construct the distance matrix using Euclidean distances. Perform **Single Linkage** (if your registration number is divisible by 3), perform **Average Linkage** (if the remainder of dividing your registration number by three is 2) or perform **Complete Linkage** (if the remainder of dividing your registration number by three is 1).

Using the first 15 data points in the file, display the result (the two clusters to be merged and the updated distance matrix after the merger) of each iteration as proof of your solution.

CS351-L – Introduction to Artificial Intelligence Lab 11

UNSUPERVISED LEARNING – K-MEANS CLUSTERING



Contents: Dr. Hashim Ali

Ghulam Ishaq Khan Institute of Engineering Sciences

Faculty of Computer Science & Engineering

CS351L – Introduction to Artificial Intelligence Lab

Lab 11 – Unsupervised Learning II – K-Means Clustering

Unsupervised Learning – K-Means Clustering

Objective

The objective of this session is to

- Introduce and implement K-Means clustering.
- Understand and apply K-Means clustering to machine learning problems and as a simple data compression technique in images.
- Evaluate when K-Means clustering is useful and when it is faulty.

Learning outcomes

After the successful completion of this lab, the student will be able to:

1. Explain the basics of unsupervised learning.
2. Apply and implement solutions for clustering problems using K-means clustering.

Instructions

- Read through the handout sitting in front of a computer that has a Python software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering the problems, indicate the commands you entered and output displayed.
- It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document.

Table of Contents

11. K-MEANS CLUSTERING

In this lab, we will cover K-means clustering which is considered as one of the most used clustering algorithms due to its simplicity.

11.1. Introduction

K-means algorithm is an iterative algorithm that tries to partition the dataset into K pre-defined distinct non-overlapping subgroups (clusters) where each data point belongs to only one group. It tries to make the inter-cluster data points as similar as possible while also keeping the clusters as different (far) as possible. It assigns data points to a cluster such that the sum of the squared distance between the data points and the cluster's centroid (arithmetic mean of all the data points that belong to that cluster) is at the minimum. The less variation we have within clusters, the more homogeneous (similar) the data points are within the same cluster.

11.2. Algorithm

The way k-means algorithm works is as follows:

1. Specify number of clusters *K*.
2. Initialize centroids by first shuffling the dataset and then randomly selecting *K* data points for the centroids without replacement.
3. Keep iterating until there is no change to the centroids. i.e assignment of data points to clusters isn't changing.

- Compute the sum of the squared distance between data points and all centroids.
- Assign each data point to the closest cluster (centroid).
- Compute the centroids for the clusters by taking the average of the all data points that belong to each cluster.

The approach k-means follows to solve the problem is called **Expectation-Maximization**. The E-step is assigning the data points to the closest cluster. The M-step is computing the centroid of each cluster.

Few things to note here:

- Since clustering algorithms including k-means use distance-based measurements to determine the similarity between data points, it's recommended to standardize the data to have a mean of zero and a standard deviation of one since almost always the features in any dataset would have different units of measurements such as age vs income.
- Given k-means iterative nature and the random initialization of centroids at the start of the algorithm, different initializations may lead to different clusters since kmeans algorithm may stuck in a local optimum and may not converge to global optimum. Therefore, it's recommended to run the algorithm using different initializations of centroids and pick the results of the run that that yielded the lower sum of squared distance.
- Assignment of examples isn't changing is the same thing as no change in within-cluster variation.

11.3. Exercise

Exercise 11.1.

Implement the following algorithm in Python.

Algorithm 1: K-Means Algorithm

Input: $E = \{e_1, e_2, \ldots, e_n\}$ (set of entities to be clustered)
k (number of clusters)
$MaxIters$ (limit of iterations)
Output: $C = \{c_1, c_2, \ldots, c_k\}$ (set of cluster centroids)
$L = \{l(e) \mid e = 1, 2, \ldots, n\}$ (set of cluster labels of E)

foreach $c_i \in C$ **do**
  $c_i \leftarrow e_j \in E$ (e.g. random selection)
end
foreach $e_i \in E$ **do**
  $l(e_i) \leftarrow argminDistance(e_i, c_j) j \in \{1 \ldots k\}$
end

$changed \leftarrow false$;
$iter \leftarrow 0$;
repeat
  foreach $c_i \in C$ **do**
    $UpdateCluster(c_i)$;
  end
  foreach $e_i \in E$ **do**
    $minDist \leftarrow argminDistance(e_i, c_j)\ j \in \{1 \ldots k\}$;
    if $minDist \neq l(e_i)$ **then**
      $l(e_i) \leftarrow minDist$;
      $changed \leftarrow true$;
    end
  end
  $iter++$;
until $changed = true$ *and* $iter \leq MaxIters$;

Then complete the following examples and show the demo and code for both the exercise and the examples to the instructor.

11.4. Applications

k-means algorithm is very popular and used in a variety of applications such as market segmentation, document clustering, image segmentation and image compression, etc. The goal usually when we undergo a cluster analysis is either:

Get a meaningful intuition of the structure of the data we're dealing with.

Cluster-then-predict where different models will be built for different subgroups if we believe there is a wide variation in the behaviors of different subgroups. An example of that is clustering patients into different subgroups and build a model for each subgroup to predict the probability of the risk of having heart attack.

In this lecture, we'll apply clustering on two cases:

- Geyser eruptions segmentation (2D dataset).
- Image compression.

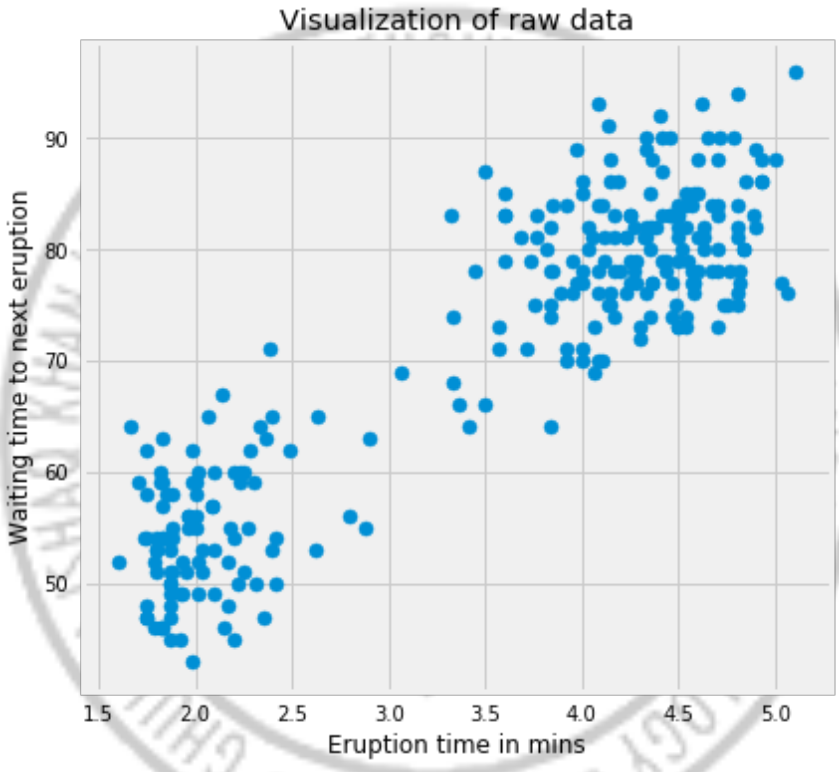
11.4.1. K-means on Geyser's Eruptions Segmentation

We'll first implement the k-means algorithm on 2D dataset and see how it works. The dataset has 272 observations and 2 features. The data covers the waiting time between eruptions and the duration of the eruption for the Old Faithful geyser in Yellowstone National Park, Wyoming, USA. We will try to find K subgroups within the data points and group them accordingly. Below is the description of the features:

- eruptions (float): Eruption time in minutes.
- waiting (int): Waiting time to next eruption.

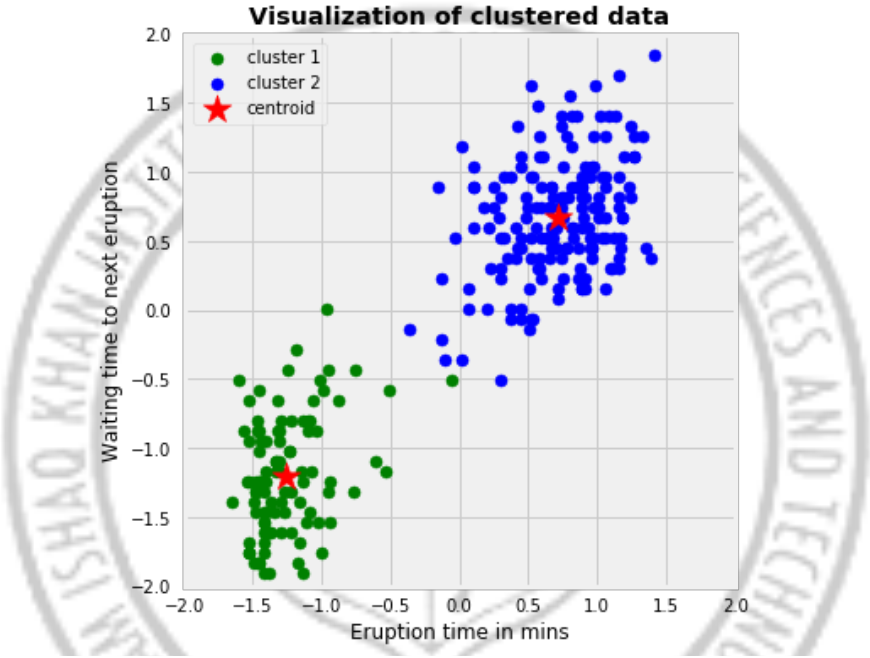
Let's plot the data first:

```
# Modules
import matplotlib.pyplot as plt
from matplotlib.image import imread
import pandas as pd
import seaborn as sns
from sklearn.datasets.samples_generator import (make_blobs,
                                                make_circles,
                                                make_moons)
from sklearn.cluster import KMeans, SpectralClustering
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import silhouette_samples, silhouette_score

%matplotlib inline
sns.set_context('notebook')
plt.style.use('fivethirtyeight')
from warnings import filterwarnings
filterwarnings('ignore')

# Import the data
df = pd.read_csv('../data/old_faithful.csv')

# Plot the data
plt.figure(figsize=(6, 6))
plt.scatter(df.iloc[:, 0], df.iloc[:, 1])
plt.xlabel('Eruption time in mins')
plt.ylabel('Waiting time to next eruption')
plt.title('Visualization of raw data');
```



We'll use this data because it's easy to plot and visually spot the clusters since it's a 2-dimension dataset. It's obvious that we have 2 clusters. Let's standardize the data first and run the k-means algorithm on the standardized data with K=2.

```
# Standardize the data
X_std = StandardScaler().fit_transform(df)

# Run local implementation of kmeans
km = Kmeans(n_clusters=2, max_iter=100)
km.fit(X_std)
centroids = km.centroids

# Plot the clustered data
fig, ax = plt.subplots(figsize=(6, 6))
plt.scatter(X_std[km.labels == 0, 0], X_std[km.labels == 0, 1],
            c='green', label='cluster 1')
plt.scatter(X_std[km.labels == 1, 0], X_std[km.labels == 1, 1],
            c='blue', label='cluster 2')
plt.scatter(centroids[:, 0], centroids[:, 1], marker='*', s=300,
            c='r', label='centroid')
plt.legend()
plt.xlim([-2, 2])
plt.ylim([-2, 2])
plt.xlabel('Eruption time in mins')
plt.ylabel('Waiting time to next eruption')
plt.title('Visualization of clustered data', fontweight='bold')
ax.set_aspect('equal');
```

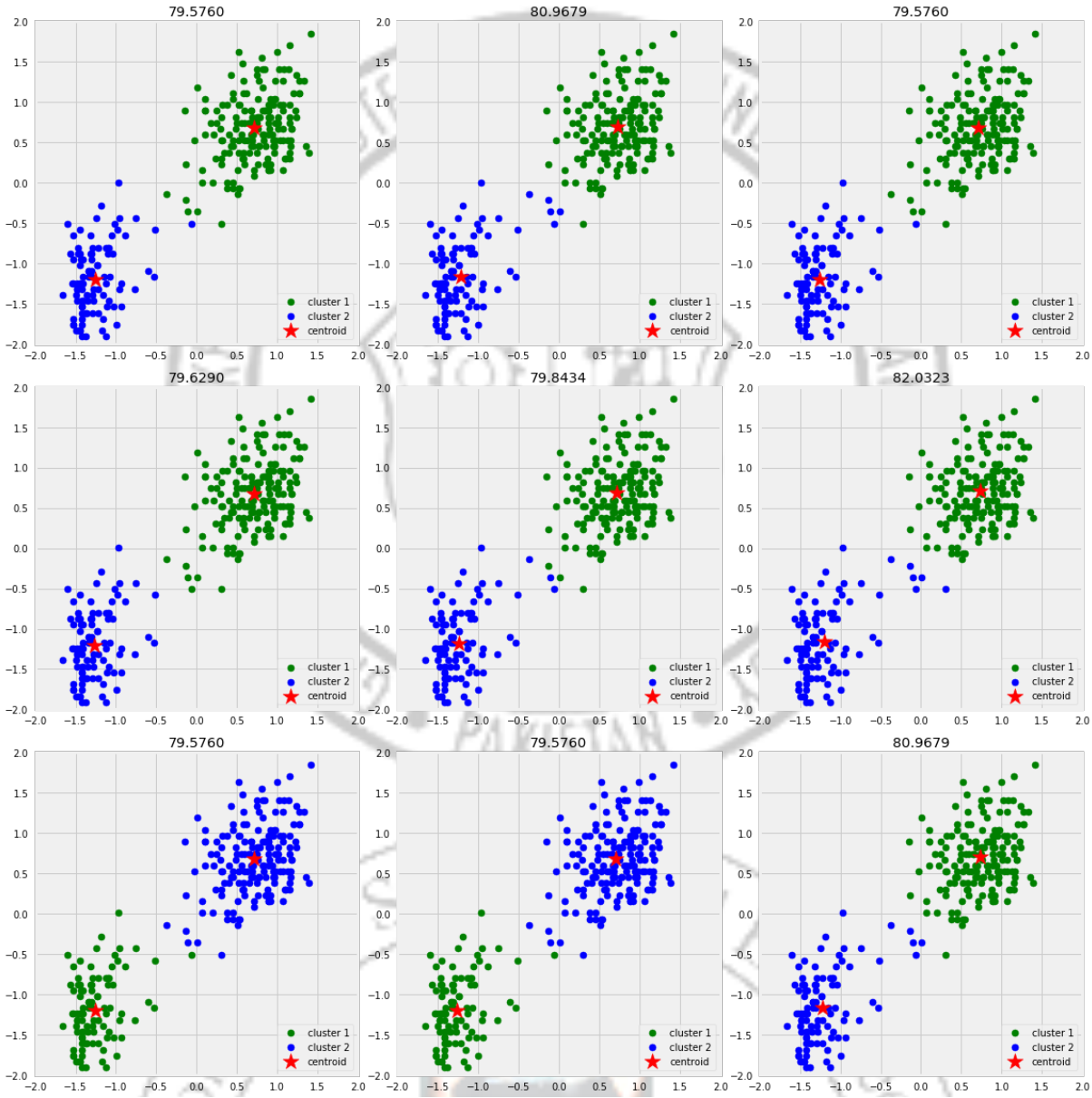


The above graph shows the scatter plot of the data colored by the cluster they belong to. In this example, we chose K=2. The symbol '*' is the centroid of each cluster. We can think of those 2 clusters as geyser had different kinds of behaviors under different scenarios.

Next, we'll show that different initializations of centroids may yield to different results. I'll use 9 different random_state to change the initialization of the centroids and plot the results. The title of each plot will be the sum of squared distance of each initialization.

As a side note, this dataset is considered very easy and converges in less than 10 iterations. Therefore, to see the effect of random initialization on convergence, I am going to go with 3 iterations to illustrate the concept. However, in real world applications, datasets are not at all that clean and nice!

```
n_iter = 9
fig, ax = plt.subplots(3, 3, figsize=(16, 16))
ax = np.ravel(ax)
centers = []
for i in range(n_iter):
    # Run local implementation of kmeans
    km = Kmeans(n_clusters=2,
                max_iter=3,
                random_state=np.random.randint(0, 1000, size=1))
    km.fit(X_std)
    centroids = km.centroids
    centers.append(centroids)
    ax[i].scatter(X_std[km.labels == 0, 0], X_std[km.labels == 0, 1],
                  c='green', label='cluster 1')
    ax[i].scatter(X_std[km.labels == 1, 0], X_std[km.labels == 1, 1],
                  c='blue', label='cluster 2')
    ax[i].scatter(centroids[:, 0], centroids[:, 1],
                  c='r', marker='*', s=300, label='centroid')
    ax[i].set_xlim([-2, 2])
    ax[i].set_ylim([-2, 2])
    ax[i].legend(loc='lower right')
    ax[i].set_title(f'{km.error:.4f}')
    ax[i].set_aspect('equal')
plt.tight_layout();
```



As the graph above shows that we only ended up with two different ways of clustering based on different initializations. We would pick the one with the lowest sum of squared distance.

11.4.2. K-means on image compression

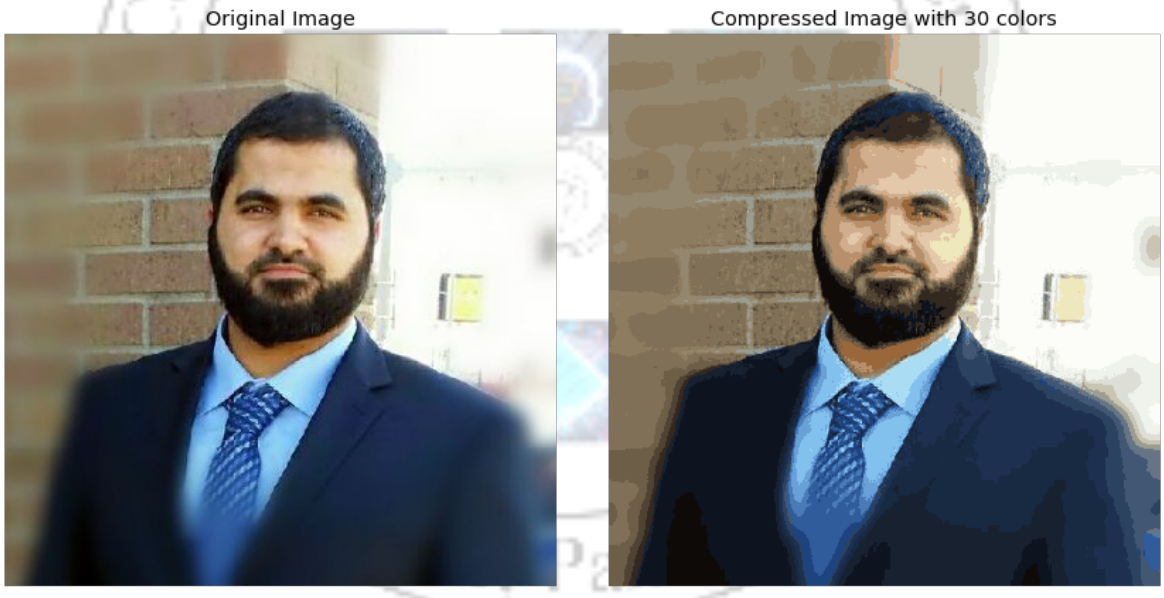
In this part, we'll implement k-means to compress an image. The image that we'll be working on is 396 x 396 x 3. Therefore, for each pixel location we would have 3 8-bit integers that specify the red, green, and blue intensity values. Our goal is to reduce the number of colors to 30 and represent (compress) the photo using those 30 colors only. To pick which colors to use, we'll use k-means algorithm on the image and treat every pixel as a data point. That means reshape the image from height x width x channels to (height * width) x channel, i.e., we would have 396 x 396 = 156,816 data points in 3-dimensional space which are the intensity of RGB. Doing so will allow us to represent the image using the 30 centroids for each pixel and would significantly reduce the size of the image by a factor of 6. The original image size was 396 x 396 x 24 = 3,763,584 bits; however, the new compressed image would be 30 x 24 + 396 x 396 x 4 = 627,984 bits. The huge difference comes from the fact that we'll be using centroids as a

lookup for pixels' colors and that would reduce the size of each pixel location to 4-bit instead of 8-bit.

From now on we will use sklearn implementation of kmeans. Few things to note here:

- n_init is the number of times of running the kmeans with different centroid's initialization. The result of the best one will be reported.
- tol is the within-cluster variation metric used to declare convergence.
- The default of init is k-means++ which is supposed to yield a better result than just random initialization of centroids.

```
# Read the image
img = imread('images/my_image.jpg')
img_size = img.shape

# Reshape it to be 2-dimension
X = img.reshape(img_size[0] * img_size[1], img_size[2])

# Run the Kmeans algorithm
km = KMeans(n_clusters=30)
km.fit(X)

# Use the centroids to compress the image
X_compressed = km.cluster_centers_[km.labels_]
X_compressed = np.clip(X_compressed.astype('uint8'), 0, 255)

# Reshape X_recovered to have the same dimension as the original image 128
* 128 * 3
X_compressed = X_compressed.reshape(img_size[0], img_size[1], img_size[2])

# Plot the original and the compressed image next to each other
fig, ax = plt.subplots(1, 2, figsize = (12, 8))
ax[0].imshow(img)
ax[0].set_title('Original Image')
ax[1].imshow(X_compressed)
ax[1].set_title('Compressed Image with 30 colors')
for ax in fig.axes:
    ax.axis('off')
plt.tight_layout();
```



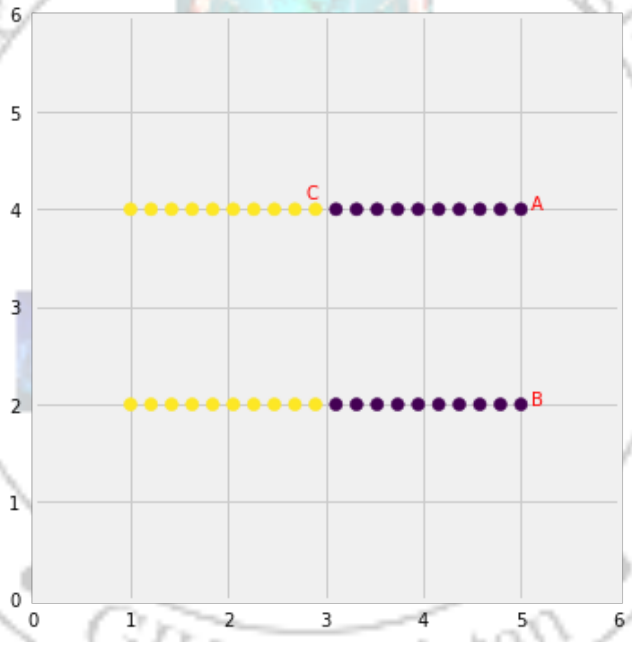
We can see the comparison between the original image and the compressed one. The compressed image looks close to the original one which means we're able to retain the majority of the characteristics of the original image. With smaller number of clusters, we would have higher compression rate at the expense of image quality. As a side note, this image compression method is called lossy data compression because we can't reconstruct the original image from the compressed image.

11.5. Drawbacks

K-means algorithm is good in capturing structure of the data if clusters have a spherical-like shape. It always tries to construct a nice spherical shape around the centroid. That means, the minute the clusters have a complicated geometric shape, k-means does a poor job in clustering the data. We'll illustrate three cases where k-means will not perform well.

First, k-means algorithm doesn't let data points that are far-away from each other share the same cluster even though they obviously belong to the same cluster.

```
# Create horizantal data
X = np.tile(np.linspace(1, 5, 20), 2)
y = np.repeat(np.array([2, 4]), 20)
df = np.c_[X, y]

km = KMeans(n_clusters=2)
km.fit(df)
labels = km.predict(df)
centroids = km.cluster_centers_

fig, ax = plt.subplots(figsize=(6, 6))
plt.scatter(X, y, c=labels)
plt.xlim([0, 6])
plt.ylim([0, 6])
plt.text(5.1, 4, 'A', color='red')
plt.text(5.1, 2, 'B', color='red')
plt.text(2.8, 4.1, 'C', color='red')
ax.set_aspect('equal')
```

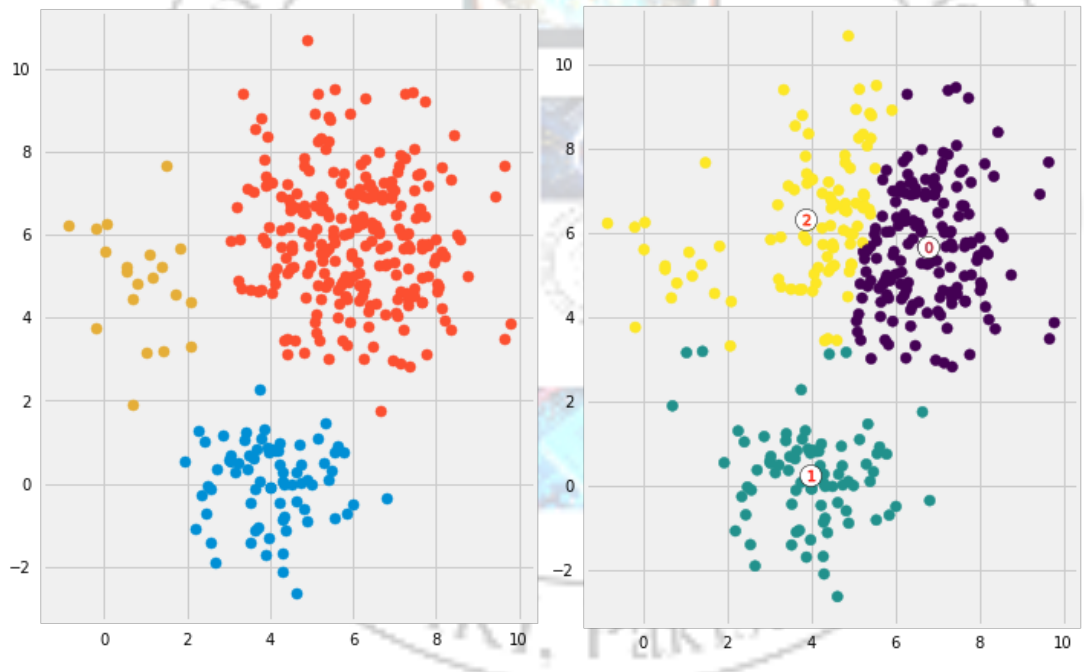


K-means considers the point 'B' closer to point 'A' than point 'C' since they have non-spherical shape. Therefore, points 'A' and 'B' will be in the same cluster but point 'C' will be

in a different cluster. Note the Single Linkage hierarchical clustering method gets this right because it doesn't separate similar points).

Second, we'll generate data from multivariate normal distributions with different means and standard deviations. So, we would have 3 groups of data where each group was generated from different multivariate normal distribution (different mean/standard deviation). One group will have a lot more data points than the other two combined. Next, we'll run k-means on the data with K=3 and see if it will be able to cluster the data correctly. To make the comparison easier, I am going to plot first the data colored based on the distribution it came from. Then I will plot the same data but now colored based on the clusters they have been assigned to.

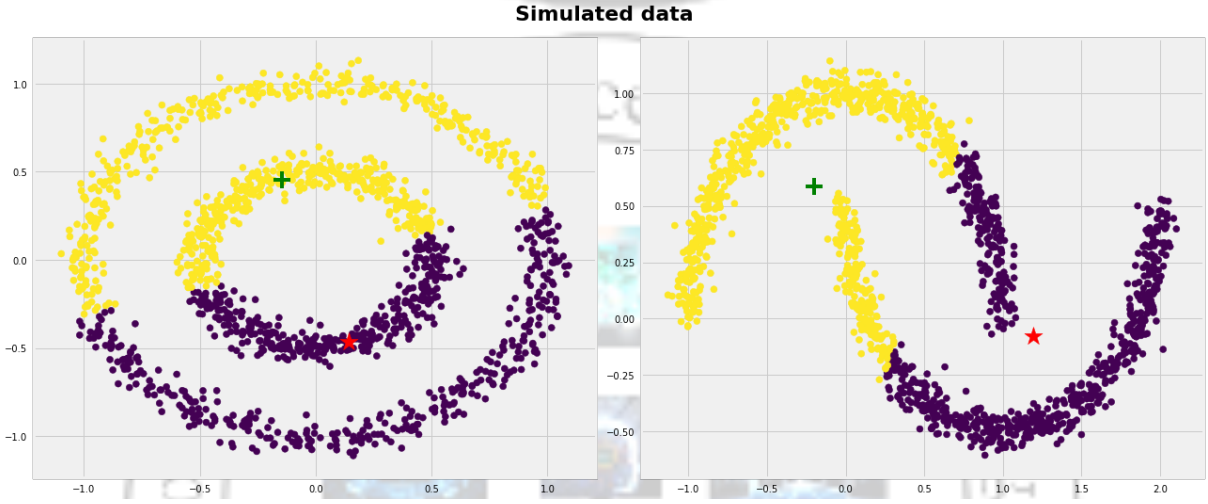
```
# Create data from three different multivariate distributions
X_1=np.random.multivariate_normal(mean=[4, 0],cov=[[1, 0],[0, 1]], size=75)
X_2=np.random.multivariate_normal(mean=[6, 6],cov=[[2, 0],[0, 2]],size=250)
X_3=np.random.multivariate_normal(mean=[1, 5],cov=[[1, 0],[0, 2]],size=20)
df = np.concatenate([X_1, X_2, X_3])

# Run kmeans
km = KMeans(n_clusters=3)
km.fit(df)
labels = km.predict(df)
centroids = km.cluster_centers_

# Plot the data
fig, ax = plt.subplots(1, 2, figsize=(10, 10))
ax[0].scatter(X_1[:, 0], X_1[:, 1])
ax[0].scatter(X_2[:, 0], X_2[:, 1])
ax[0].scatter(X_3[:, 0], X_3[:, 1])
ax[0].set_aspect('equal')
ax[1].scatter(df[:, 0], df[:, 1], c=labels)
ax[1].scatter(centroids[:, 0], centroids[:, 1], marker='o',
                c="white", alpha=1, s=200, edgecolor='k')
for i, c in enumerate(centroids):
    ax[1].scatter(c[0], c[1], marker='$%d$' % i, s=50, alpha=1,
edgecolor='r')
ax[1].set_aspect('equal')
plt.tight_layout()
```



Looks like k-means couldn't figure out the clusters correctly. Since it tries to minimize the within-cluster variation, it gives more weight to bigger clusters than smaller ones. In other words, data points in smaller clusters may be left away from the centroid in order to focus more on the larger cluster.

Last, we'll generate data that have complicated geometric shapes such as moons and circles within each other and test k-means on both of the datasets.

```
# Circles
X1 = make_circles(factor=0.5, noise=0.05, n_samples=1500)

# Moons
X2 = make_moons(n_samples=1500, noise=0.05)

fig, ax = plt.subplots(1, 2)
for i, X in enumerate([X1, X2]):
    fig.set_size_inches(18, 7)
    km = KMeans(n_clusters=2)
    km.fit(X[0])
    labels = km.predict(X[0])
    centroids = km.cluster_centers_

    ax[i].scatter(X[0][:, 0], X[0][:, 1], c=labels)
    ax[i].scatter(centroids[0, 0], centroids[0, 1], marker='*', s=400,
c='r')
    ax[i].scatter(centroids[1, 0], centroids[1, 1], marker='+', s=300,
c='green')
plt.suptitle('Simulated data', y=1.05, fontsize=22, fontweight='semibold')
```



As expected, k-means couldn't figure out the correct clusters for both datasets.

11.6. Conclusion

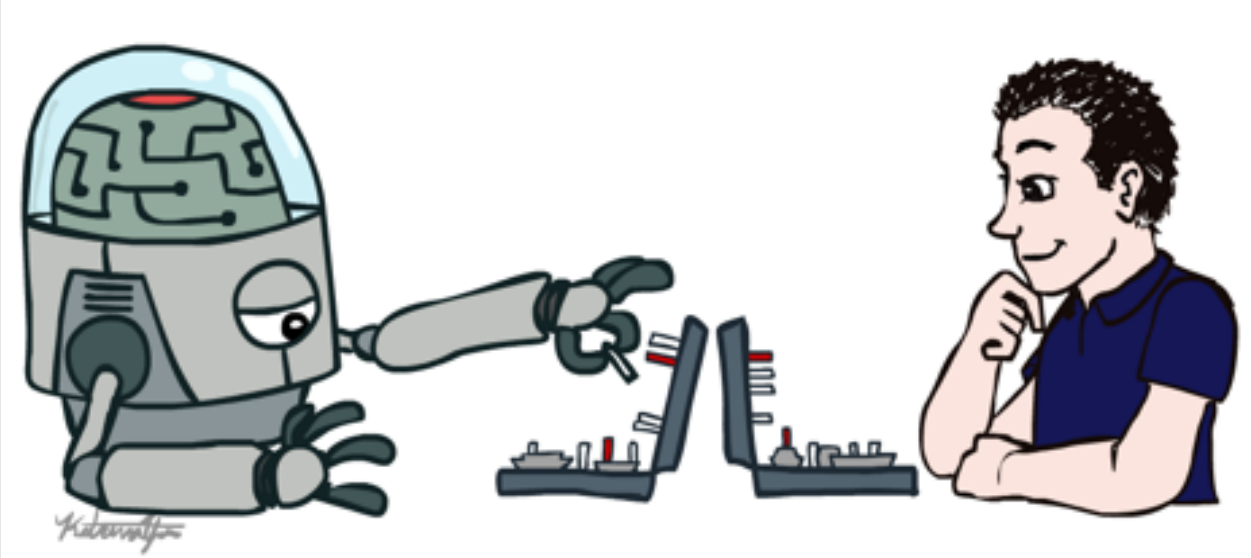
K-means clustering is one of the most popular clustering algorithms and usually the first thing practitioners apply when solving clustering tasks to get an idea of the structure of the dataset. The goal of k-means is to group data points into distinct non-overlapping subgroups. It does a very good job when the clusters have a kind of spherical shapes. However, it suffers as the geometric shapes of clusters deviates from spherical shapes. Moreover, it also doesn't learn the number of clusters from the data and requires it to be pre-defined. To be a good practitioner, it's good to know the assumptions behind algorithms/methods so that you would have a pretty good idea about the strength and weakness of each method. This will help you decide when to use each method and under what circumstances. In this lab, we covered both strength, weaknesses, and some evaluation methods related to k-means.

Below are the main takeaways:

- Scale/standardize the data when applying k-means algorithm.
- K-means gives more weight to the bigger clusters.
- K-means assumes spherical shapes of clusters (with radius equal to the distance between the centroid and the furthest data point) and doesn't work well when clusters are in different shapes such as elliptical clusters.
- If there is overlapping between clusters, k-means doesn't have an intrinsic measure for uncertainty for the examples belong to the overlapping region in order to determine for which cluster to assign each data point.
- K-means may still cluster the data even if it can't be clustered such as data that comes from uniform distributions.

CS351-L – Introduction to Artificial Intelligence Lab 12

HANDLING UNCERTAINTY – FUZZY LOGIC



Contents: Dr. Hashim Ali

Ghulam Ishaq Khan Institute of Engineering Sciences

Faculty of Computer Science & Engineering

CS351L – Introduction to Artificial Intelligence Lab

Lab 12 – Handling Uncertainty – Fuzzy Logic

Handling Uncertainty – Fuzzy Logic

Objective

The objective of this session is to

- Introduce and understand the working of Fuzzy logic sytems.
- Evaluate when Fuzzy logic systems are useful and when it is not.

Learning outcomes

After the successful completion of this lab, the student will be able to:

1. Explain the basics of uncertainty and fuzzy logic systems.
2. Implement solution for handling uncertainty using Fuzzy logic in control systems.

Instructions

- Read through the handout sitting in front of a computer that has a Python software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering the problems, indicate the commands you entered and output displayed.
- It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document.

Table of Contents

12. Fuzzy Logic

The term fuzzy mean things which are not very clear or vague. In real life, we may come across a situation where we can't decide whether the statement is true or false. At that time, fuzzy logic offers very valuable flexibility for reasoning. We can also consider the uncertainties of any situation.

Fuzzy logic algorithm helps to solve a problem after considering all available data. Then it takes the best possible decision for the given the input. The FL method imitates the way of decision making in a human which consider all possibilities between digital values T and F.

12.1. Characteristics

Here, are some important characteristics of fuzzy logic:

- Flexible and easy to implement machine learning technique
- Helps you to mimic the logic of human thought
- Logic may have two values which represent two possible solutions
- Highly suitable method for uncertain or approximate reasoning
- Fuzzy logic views inference as a process of propagating elastic constraints
- Fuzzy logic allows you to build nonlinear functions of arbitrary complexity.
- Fuzzy logic should be built with the complete guidance of experts

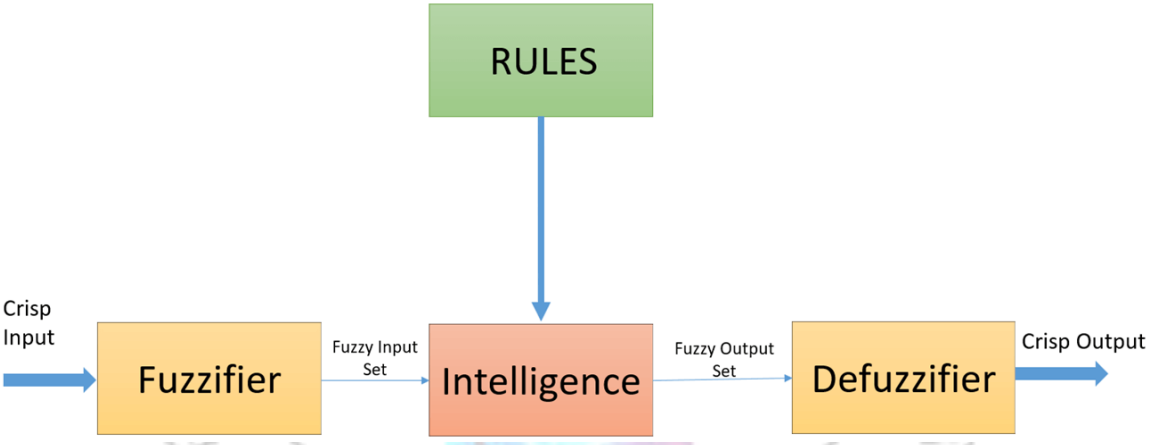
12.2. When not to use Fuzzy Logic

However, fuzzy logic is never a cure for all. Therefore, it is equally important to understand that where we should not use fuzzy logic.

Here, are certain situations when you better not use Fuzzy Logic:

- If you don't find it convenient to map an input space to an output space
- Fuzzy logic should not be used when you can use common sense
- Many controllers can do the fine job without the use of fuzzy logic

12.3. Fuzzy Logic Architecture



Fuzzy Logic architecture has four main parts as shown in the diagram:

12.3.1. Rule Base:

It contains all the rules and the if-then conditions offered by the experts to control the decision-making system. The recent update in fuzzy theory provides various methods for the design and tuning of fuzzy controllers. This updates significantly reduce the number of the fuzzy set of rules.

12.3.2. Fuzzification:

Fuzzification step helps to convert inputs. It allows you to convert, crisp numbers into fuzzy sets. Crisp inputs measured by sensors and passed into the control system for further processing. Like Room temperature, pressure, etc.

12.3.3. Inference Engine:

It helps you to determines the degree of match between fuzzy input and the rules. Based on the % match, it determines which rules need implment according to the given input field. After this, the applied rules are combined to develop the control actions.

12.3.4. Defuzzification:

At last the Defuzzification process is performed to convert the fuzzy sets into a crisp value. There are many types of techniques available, so you need to select it which is best suited when it is used with an expert system.

12.4. Comparison of Fuzzy Logic with other terms

12.4.1. Fuzzy Logic vs. Probability

Fuzzy Logic	**Probability**
Fuzzy: Tom's degree of membership within the set of old people is 0.90.	Probability: There is a 90% chance that Tom is old.
Fuzzy logic takes truth degrees as a mathematical basis on the model of the vagueness phenomenon.	Probability is a mathematical model of ignorance.

12.4.2 Crisp vs. Fuzzy

Crisp	**Fuzzy**
It has strict boundary T or F	Fuzzy boundary with a degree of membership
Some crisp time set can be fuzzy	It can't be crisp
True/False {0,1}	Membership values on [0,1]
In Crisp logic law of Excluded Middle and Non- Contradiction may or may not hold	In the fuzzy logic law of Excluded Middle and Non- Contradiction hold

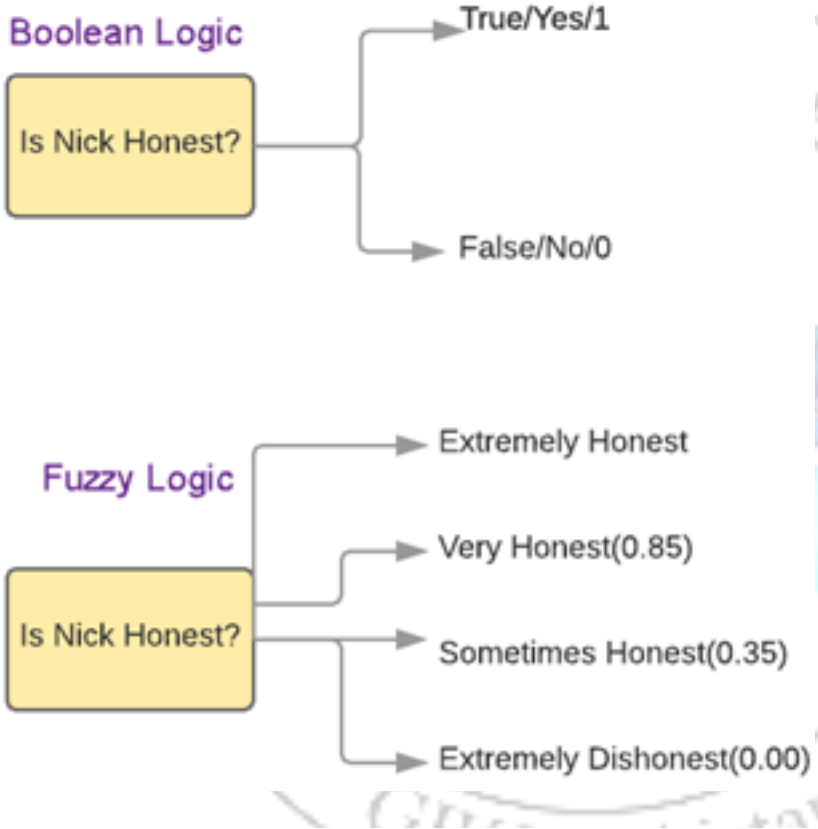
12.4.3. Classical set vs. Fuzzy set Theory

Classical Set	**Fuzzy Set Theory**
Classes of objects with sharp boundaries.	Classes of objects do not have sharp boundaries.
A classical set is defined by crisp boundaries, i.e., there is clarity about the location of the set boundaries.	A fuzzy set always has ambiguous boundaries, i.e., there may be uncertainty about the location of the set boundaries.
Widely used in digital system design	Used only in fuzzy controllers.

12.5. Examples of Fuzzy Logic

12.5.1. Example 1 – Simple honesty ratings

See the below-given diagram. It shows that in fuzzy systems, the values are denoted by a 0 to 1 number. In this example, 1.0 means absolute truth and 0.0 means absolute falseness.



12.5.2. Example 2 – Tipping problem

Problem: You went to a restaurant and wish to tip the waiter at the end. What should be the exact amount of tip given to the waiter?

Solution:

- Antecednets (Inputs)
  - – service
    - * Universe (i.e., crisp value range): How good was the service of the waiter, on a scale of 1 to 10?
    - * Fuzzy set (i.e., fuzzy value range): poor, acceptable, amazing
  - – food quality
    - * Universe: How tasty was the food, on a scale of 1 to 10?
    - * Fuzzy set: bad, decent, great
- Consequents (Outputs)
  - – tip
    - * Universe: How much should we tip, on a scale of 0% to 25%
    - * Fuzzy set: low, medium, high
- Rules
  - – IF the service was good or the food quality was good, THEN the tip will be high.
  - – IF the service was average, THEN the tip will be medium.
  - – IF the service was poor and the food quality was poor THEN the tip will be low.
- Usage
  - – If I tell this controller that I rated:
    - * the service as 9.8, and
    - * the quality as 6.5,
  - – it would recommend I leave:
    - * a 20.2% tip.

12.5.3. Example 3 – Fuzzy Control System Applied in a Room cooler

Problem: A room cooler has a fan encased in a box with wool. The wool is continuously moistened by water that flows through a pump connected to a motor. The rate of flow of water is to be determined; it is a function of room temperature and the speed of motor.

What is the *water flow rate*, if *fan speed of the motor* is *31 RPM* and *temperature* is *42 degree Celsius*?

Given: Two input variables – Room temperature and cooler fan speed control the output variable – flow rate of the water. The fuzzy regions using fuzzy terms for input-output are defined as follows:

Variable name	Fuzzy terms
Temperature	Cold, Cool, Moderate, Warm and Hot
Fan speed ((rotations per minute)	Slack, Low, Medium, Brisk, fast
Flow rate of water	Strong Negative (SN), Negative (N), Low-Negative (LN), Medium (M), Low-Positive (LP), Positive (P), and High-Positive (HP)

Fuzzy profiles are defined for each of the three parameters by assigning memberships to their respective values. The profiles have to be carefully designed after studying the nature and desired behavior of the system.

Fuzzy relationships for the inputs – Temperature

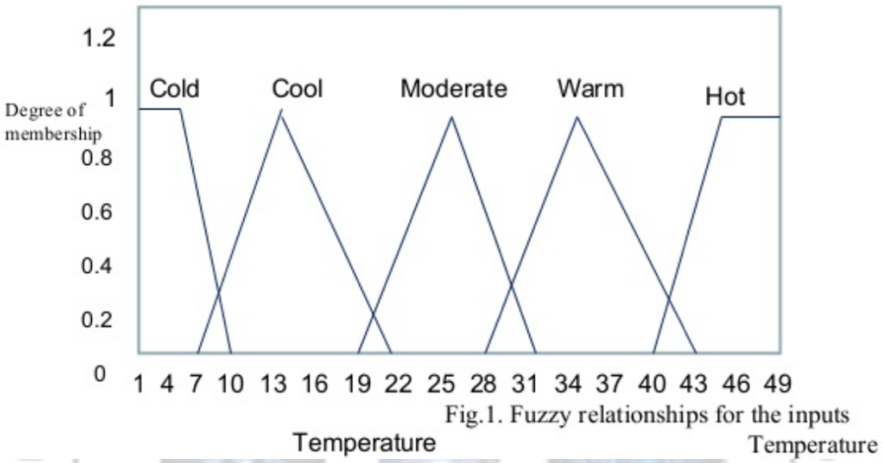
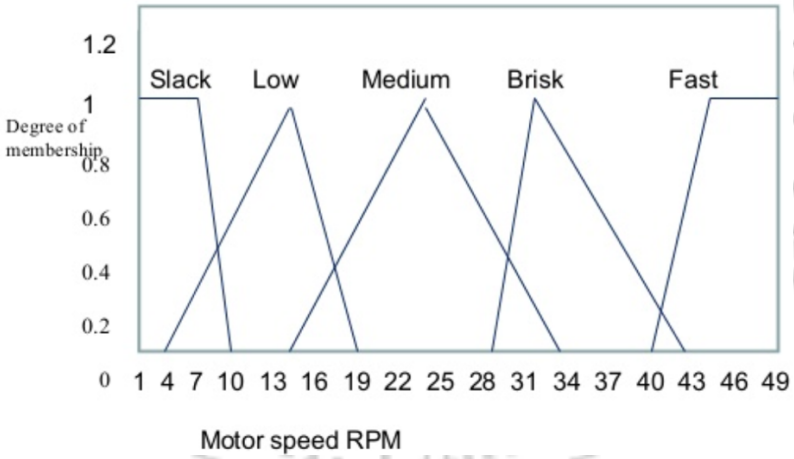
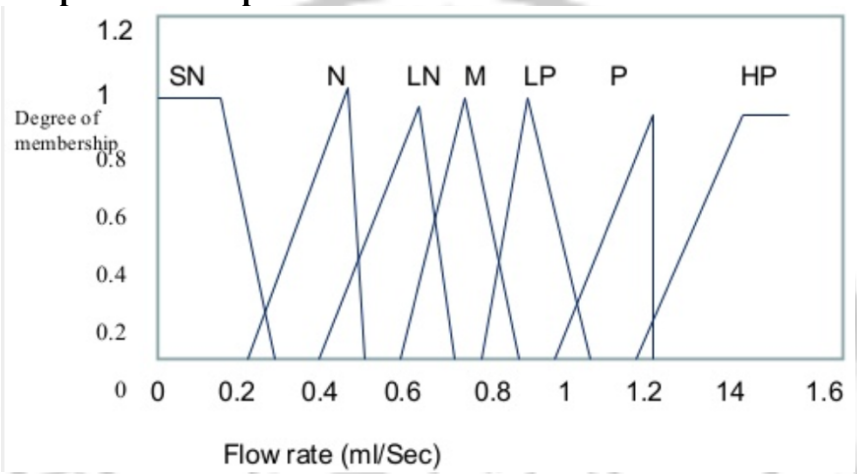


Fig.1. Fuzzy relationships for the inputs Temperature

Fuzzy relationships for the inputs – Fan motor speed



Fuzzy relationships for the outputs – Water flow rate



Fuzzy rules for Fuzzy Room Cooler

The Fuzzy rules form the triggers of the fuzzy engine. After a study of the system, the rules could be written as follows:

R1: If temperature is HOT and fan motor speed is SLACK then the flow-rate is HIGH-POSITIVE.

R2: If temperature is HOT and fan motor speed is LOW then the flow-rate is HIGH-POSITIVE.

R3: If temperature is HOT and fan motor speed is MEDIUM then the flow-rate is POSITIVE.

R4: If temperature is HOT and fan motor speed is BRISK then the flow-rate is HIGH-POSITIVE.

R5: If temperature is WARM and fan motor speed is MEDIUM then the flow-rate is LOW-POSITIVE.

R6: If temperature is WARM and fan motor speed is BRISK then the flow-rate is POSITIVE.

R7: If temperature is COOL and fan motor speed is LOW then the flow-rate is NEGATIVE.

R8: If temperature is MODERATE and fan motor speed is LOW then the flow-rate is MEDIUM.

Fuzzification

The fuzzifier that performs the mapping of the membership values of the input parameters – temperature and fan speed – to the respective fuzzy regions is known as fuzzification. This is the most important step in fuzzy systems.

The corresponding membership values and the fuzzy regions are shown below:

Parameters	Fuzzy Regions	Memberships
Temperature	Warm, hot	0.142, 0.2
Fan Speed	medium, brisk	0.25, 0.286

From Figure 1 above, the temperature 42 degrees corresponds to two membership values 0.142 and 0.2 that belong to WARM and HOT fuzzy regions respectively.

Similarly, from Figure 2 above, the fan speed 31 rpm corresponds to two membership values 0.25 and 0.286 that belong to the MEDIUM and BRISK fuzzy regions respectively.

Inference

From the fuzzification table, there are four combinations possible:

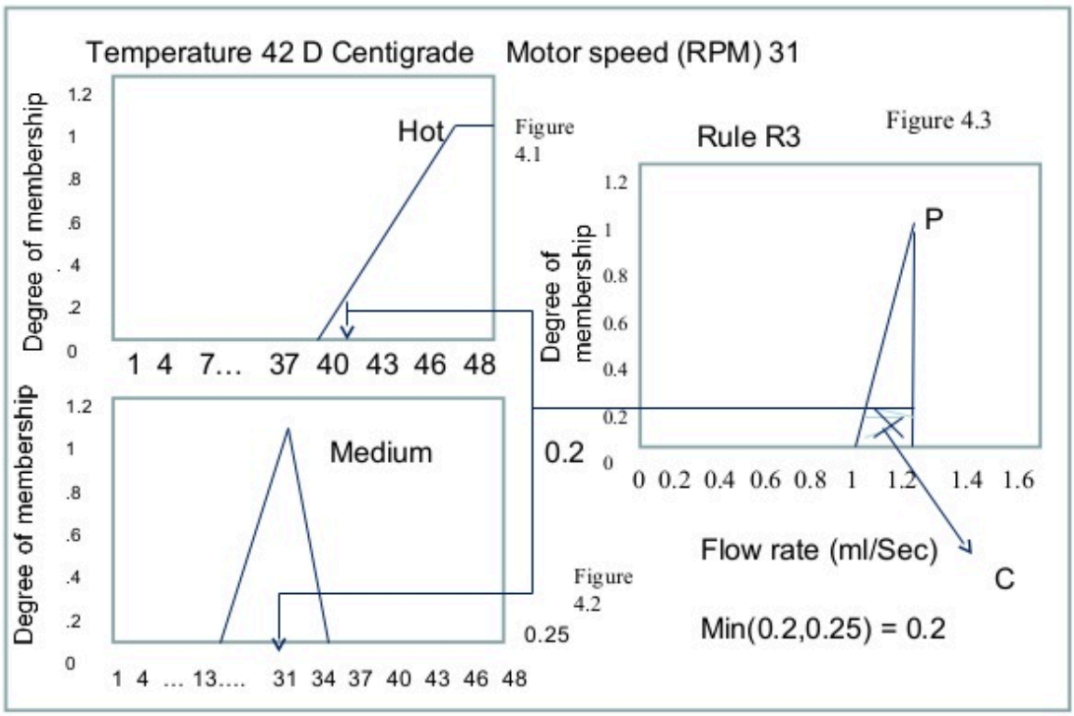
- If temperature is WARM and fan speed is MEDIUM
- If temperature is WARM and fan speed is BRISK
- If temperature is HOT and fan speed is MEDIUM
- If temperature is HOT and fan speed is BRISK

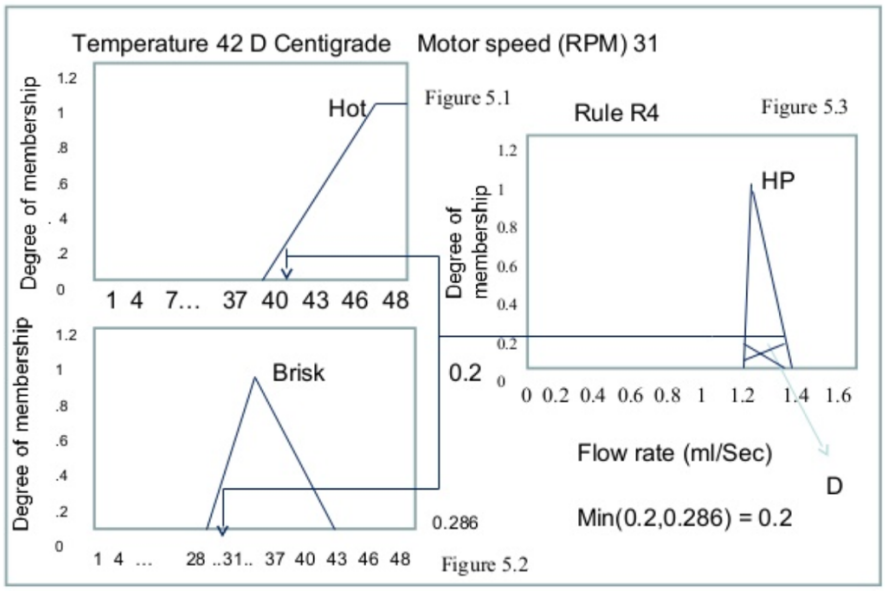
Comparing the above combinations with the left side of fuzzy rules R5, R6, R3, and R4 respectively, the flow rate should be LOW-POSITIVE, POSITIVE, POSITIVE and HIGH-POSITIVE. The conflict should be resolved and the fuzzy region is to be given as a value for the parameter water flow-rate.

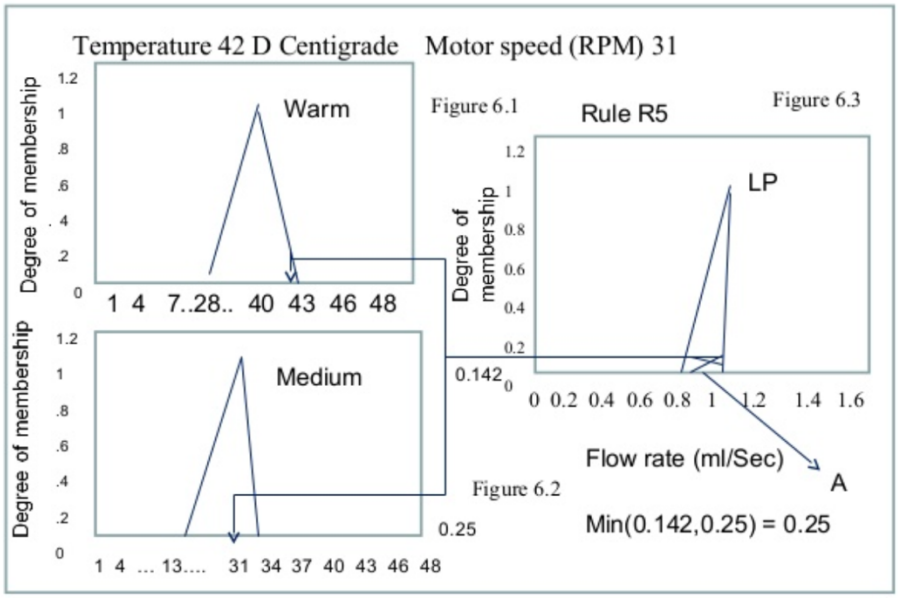
Defuzzification

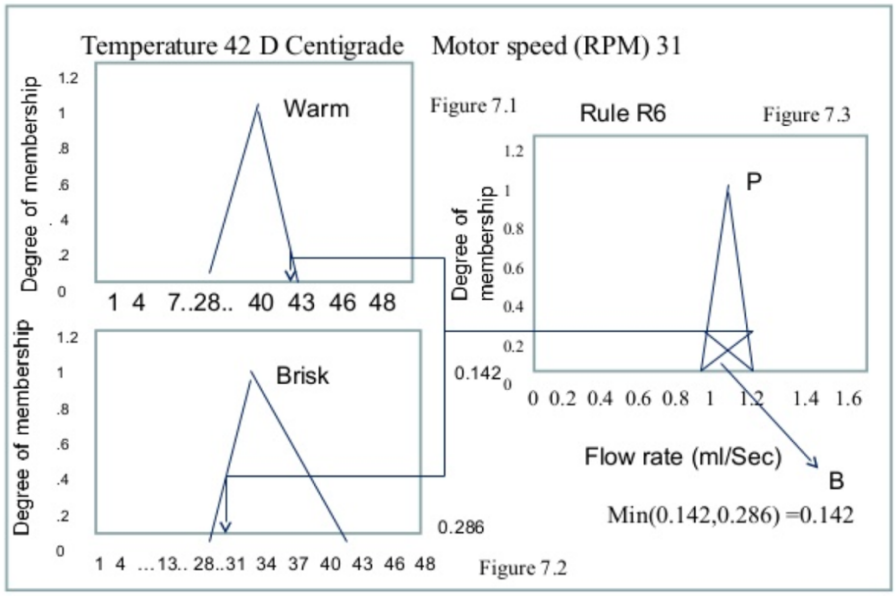
The fuzzy outputs LOW-POSITIVE, POSITIVE, and HIGH-POSITIVE are to be converted to a single crisp value that is provided to the fuzzy cooler system; this process is called defuzzification. Several methods can be used for defuzzification. We will use the centre of gravity method in this lab.

The centroid, of a two-dimensional shape X is the intersection of all straight lines that divide X into two parts of equal moment about the line or the average of all points of X. The composite region formed by the portions A, B, C, and D (corresponding to rules R3, R4, R5 and R6 respectively) on the output profile is to be computed as shown below.

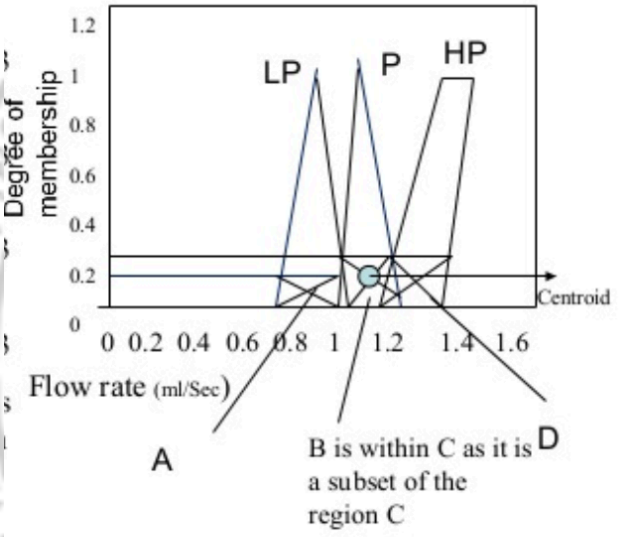



Temperature 42 D Centigrade
Motor speed (RPM) 31
Degree of membership
Hot
1 4 7... 37 40 43 46 48
Figure 5.1
Brisk
1 4 ... 28 ..31.. 37 40 43 46 48
Figure 5.2
Rule R4
Figure 5.3
HP
Degree of membership
0.2
0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6
Flow rate (ml/Sec)
D
0.286
Min(0.2,0.286) = 0.2


Temperature 42 D Centigrade
Motor speed (RPM) 31
Degree of membership
Warm
1 4 7..28.. 40 43 46 48
Figure 6.1
Medium
1 4 ... 13.... 31 34 37 40 43 46 48
Figure 6.2
Rule R5
Figure 6.3
LP
Degree of membership
0.142
0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6
Flow rate (ml/Sec)
A
0.25
Min(0.142,0.25) = 0.25


Temperature 42 D Centigrade
Motor speed (RPM) 31
Degree of membership
Warm
1 4 7..28.. 40 43 46 48
Figure 7.1
Brisk
1 4 ...13.. 28..31 34 37 40 43 46 48
Figure 7.2
Rule R6
Figure 7.3
P
Degree of membership
0.142
0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6
Flow rate (ml/Sec)
B
0.286
Min(0.142,0.286) =0.142

When parameters are connected and the minimum of their membership is taken, the composite region formed by the portions A, B, C and D on the output profile is shown below:



Result:
The centre of gravity of this composite region is the crisp output or the desired flow rate value, which in this case is 1.1 ml/Sec!

12.6. Advantages of Fuzzy Logic Systems

- The structure of Fuzzy Logic Systems is easy and understandable
- Fuzzy logic is widely used for commercial and practical purposes
- It helps you to control machines and consumer products
- It may not offer accurate reasoning, but the only acceptable reasoning
- It helps you to deal with the uncertainty in engineering
- Mostly robust as no precise inputs required
- It can be programmed to in the situation when feedback sensor stops working
- It can easily be modified to improve or alter system performance
- inexpensive sensors can be used which helps you to keep the overall system cost and complexity low
- It provides a most effective solution to complex issues

12.7. Disadvantages of Fuzzy Logic Systems

- Fuzzy logic is not always accurate; The results are perceived based on assumption, so it may not be widely accepted.
- Fuzzy systems don't have the capability of machine learning as-well-as neural network type pattern recognition
- Validation and Verification of a fuzzy knowledge-based system needs extensive testing with hardware
- Setting exact, fuzzy rules and, membership functions is a difficult task
- Some fuzzy time logic is confused with probability theory and the terms

12.8. Scikit-fuzzy library in python

12.8.1. Getting started

scikit-fuzzy is an fuzzy logic Python package that works with numpy arrays. The package is imported as skfuzzy:

```
>>> import skfuzzy
```

through the recommended import statement uses an alias:

```
>>> import skfuzzy as fuzz
```

12.8.2 Finding your way around

A list of submodules and functions is found on the API reference webpage. Within scikit-fuzzy, universe variables and fuzzy membership functions are represented by numpy arrays. Generation of membership functions is as simple as:

```
>>> import numpy as np
>>> import skfuzzy as fuzz
>>> x = np.arange(11)
>>> mfx = fuzz.trimf(x, [0, 5, 10])
>>> x
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
>>> mfx
array([ 0. , 0.2, 0.4, 0.6, 0.8, 1. , 0.8, 0.6, 0.4, 0.2, 0. ])
```

While most functions are available in the base namespace, the package is factored with a logical grouping of functions in submodules. These include

fuzz.membership
Fuzzy membership function generation

fuzz.defuzzify
Defuzzification algorithms to return crisp results from fuzzy sets

fuzz.fuzzymath
The core of scikit-fuzzy, containing the majority of the most common fuzzy logic operations.

fuzz.intervals
Interval mathematics. The restricted Dong, Shah, & Wong (DSW) methods for fuzzy set math live here.

fuzz.image
Limited fuzzy logic image processing operations.

fuzz.cluster
Fuzzy c-means clustering.

fuzz.filters
Fuzzy Inference Ruled by Else-action (FIRE) filters in 1D and 2D.

12.8.3. The Tipping Problem - The Hard Way

Note: This method computes everything by hand, step by step. The same problem is solved with the new API in the next section. The 'tipping problem' is commonly used to illustrate the power of fuzzy logic principles to generate complex behavior from a compact, intuitive set of expert rules.

Input variables

A number of variables play into the decision about how much to tip while dining. Consider two of them:

- quality : Quality of the food
- service : Quality of the service

Output variable

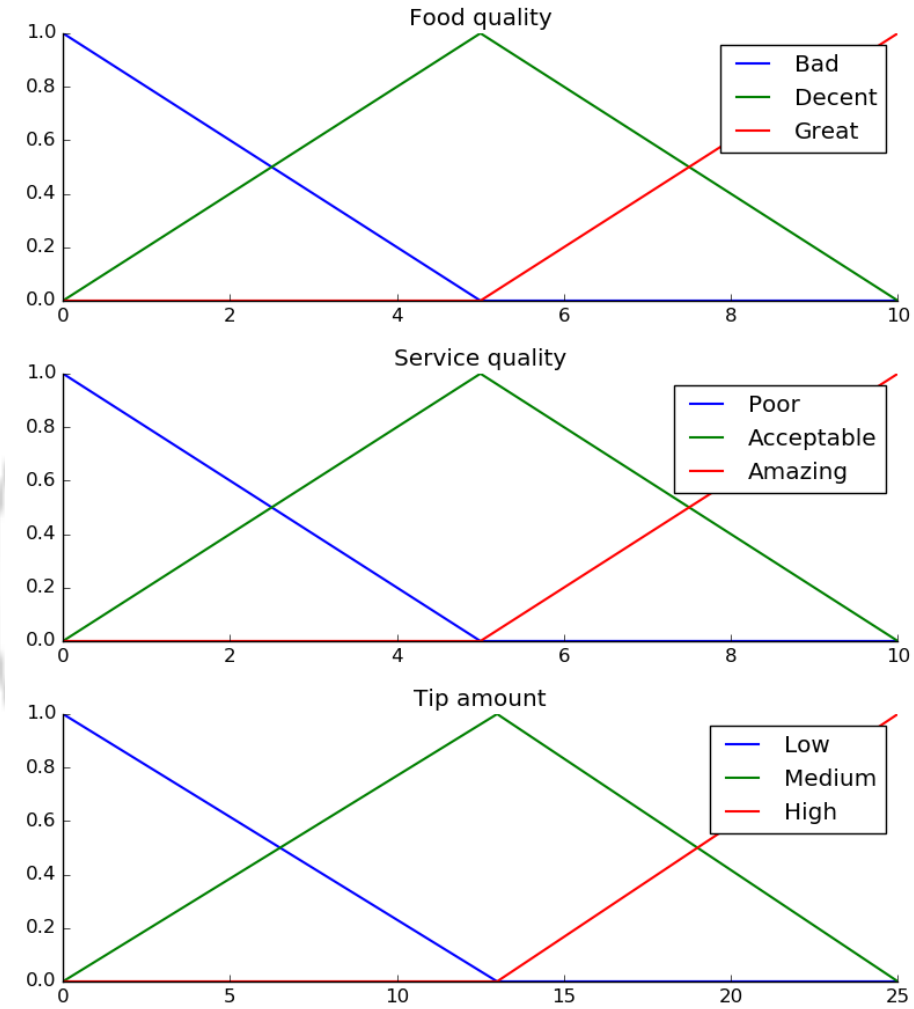
The output variable is simply the tip amount, in percentage points:

- tip : Percent of bill to add as tip

For the purposes of discussion, let's say we need 'high', 'medium', and 'low' membership functions for both input variables and our output variable. These are defined in scikit-fuzzy as follows

```
import numpy as np
import skfuzzy as fuzz
import matplotlib.pyplot as plt
# Generate universe variables
# * Quality and service on subjective ranges [0, 10]
# * Tip has a range of [0, 25] in units of percentage points
x_qual = np.arange(0, 11, 1)
x_serv = np.arange(0, 11, 1)
x_tip = np.arange(0, 26, 1)
# Generate fuzzy membership functions
qual_lo = fuzz.trimf(x_qual, [0, 0, 5])
qual_md = fuzz.trimf(x_qual, [0, 5, 10])
qual_hi = fuzz.trimf(x_qual, [5, 10, 10])
serv_lo = fuzz.trimf(x_serv, [0, 0, 5])
serv_md = fuzz.trimf(x_serv, [0, 5, 10])
serv_hi = fuzz.trimf(x_serv, [5, 10, 10])
tip_lo = fuzz.trimf(x_tip, [0, 0, 13])
tip_md = fuzz.trimf(x_tip, [0, 13, 25])
tip_hi = fuzz.trimf(x_tip, [13, 25, 25])
# Visualize these universes and membership functions
fig, (ax0, ax1, ax2) = plt.subplots(nrows=3, figsize=(8, 9))
ax0.plot(x_qual, qual_lo, 'b', linewidth=1.5, label='Bad')
ax0.plot(x_qual, qual_md, 'g', linewidth=1.5, label='Decent')
ax0.plot(x_qual, qual_hi, 'r', linewidth=1.5, label='Great')
ax0.set_title('Food quality')
ax0.legend()
ax1.plot(x_serv, serv_lo, 'b', linewidth=1.5, label='Poor')
```

```
ax1.plot(x_serv, serv_md, 'g', linewidth=1.5, label='Acceptable')
ax1.plot(x_serv, serv_hi, 'r', linewidth=1.5, label='Amazing')
ax1.set_title('Service quality')
ax1.legend()
ax2.plot(x_tip, tip_lo, 'b', linewidth=1.5, label='Low')
ax2.plot(x_tip, tip_md, 'g', linewidth=1.5, label='Medium')
ax2.plot(x_tip, tip_hi, 'r', linewidth=1.5, label='High')
ax2.set_title('Tip amount')
ax2.legend()
# Turn off top/right axes
for ax in (ax0, ax1, ax2):
      ax.spines['top'].set_visible(False)
      ax.spines['right'].set_visible(False)
      ax.get_xaxis().tick_bottom()
      ax.get_yaxis().tick_left()
plt.tight_layout()
```



Fuzzy rules

Now, to make these triangles useful, we define the fuzzy relationship between input and output variables. For the purposes of our example, consider three simple rules:

1. If the food is bad OR the service is poor, then the tip will be low
2. If the service is acceptable, then the tip will be medium
3. If the food is great OR the service is amazing, then the tip will be high.

Most people would agree on these rules, but the rules are fuzzy. Mapping the imprecise rules into a defined, actionable tip is a challenge. This is the kind of task at which fuzzy logic excels.

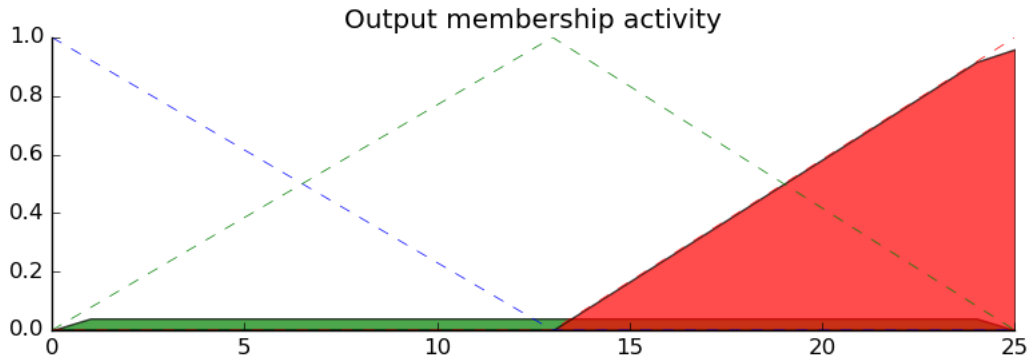
Rule application

What would the tip be in the following circumstance:

- Food quality was 6.5
- Service was 9.8

```
# We need activation of our fuzzy membership functions at these values.
# The exact values 6.5 and 9.8 do not exist on our universes...
# This is what fuzz.interp_membership exists for!
qual_level_lo = fuzz.interp_membership(x_qual, qual_lo, 6.5)
qual_level_md = fuzz.interp_membership(x_qual, qual_md, 6.5)
qual_level_hi = fuzz.interp_membership(x_qual, qual_hi, 6.5)
serv_level_lo = fuzz.interp_membership(x_serv, serv_lo, 9.8)
serv_level_md = fuzz.interp_membership(x_serv, serv_md, 9.8)
serv_level_hi = fuzz.interp_membership(x_serv, serv_hi, 9.8)
# Take our rules and apply them. Rule 1 concerns bad food OR service.
# The OR operator means we take the maximum of these two.
active_rule1 = np.fmax(qual_level_lo, serv_level_lo)
# Now we apply this by clipping the top off the corresponding output
# membership function with `np.fmin`
tip_activation_lo= np.fmin(active_rule1, tip_lo) # removed entirely to 0
# For rule 2 we connect acceptable service to medium tipping
tip_activation_md = np.fmin(serv_level_md, tip_md)
# For rule 3 we connect high service OR high food with high tipping
active_rule3 = np.fmax(qual_level_hi, serv_level_hi)
tip_activation_hi = np.fmin(active_rule3, tip_hi)
tip0 = np.zeros_like(x_tip)
# Visualize this
fig, ax0 = plt.subplots(figsize=(8, 3))
ax0.fill_between(x_tip, tip0, tip_activation_lo, facecolor='b',
alpha=0.7)
ax0.plot(x_tip, tip_lo, 'b', linewidth=0.5, linestyle='--', )
ax0.fill_between(x_tip, tip0, tip_activation_md, facecolor='g',
alpha=0.7)
ax0.plot(x_tip, tip_md, 'g', linewidth=0.5, linestyle='--')
ax0.fill_between(x_tip, tip0,tip_activation_hi, facecolor='r', alpha=0.7)
```

```
ax0.plot(x_tip, tip_hi, 'r', linewidth=0.5, linestyle='--')
ax0.set_title('Output membership activity')
# Turn off top/right axes
for ax in (ax0,):
      ax.spines['top'].set_visible(False)
      ax.spines['right'].set_visible(False)
      ax.get_xaxis().tick_bottom()
      ax.get_yaxis().tick_left()
plt.tight_layout()
```



Rule aggregation

With the activity of each output membership function known, all output membership functions must be combined.

This is typically done using a maximum operator. This step is also known as aggregation.

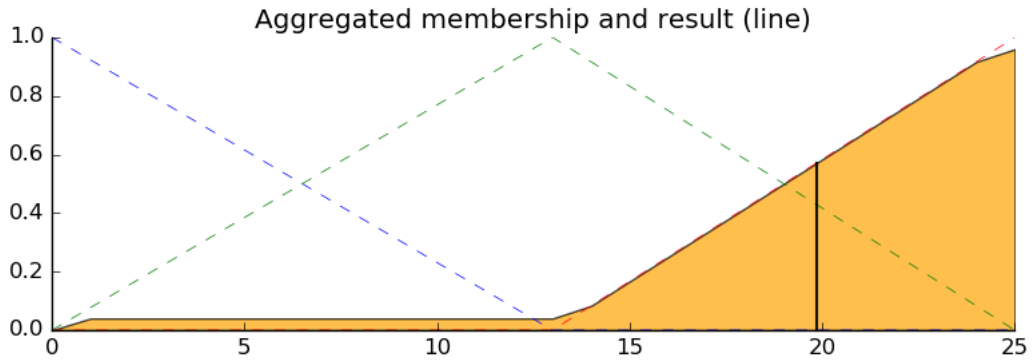
Defuzzification

Finally, to get a real-world answer, we return to crisp logic from the world of fuzzy membership functions. For the purposes of this example the centroid method will be used.

The result is a tip of 20.2%.

```
# Aggregate all three output membership functions together
aggregated = np.fmax(tip_activation_lo,
np.fmax(tip_activation_md, tip_activation_hi))
# Calculate defuzzified result
tip = fuzz.defuzz(x_tip, aggregated, 'centroid')
tip_activation = fuzz.interp_membership(x_tip, aggregated, tip) # for
plot
# Visualize this
fig, ax0 = plt.subplots(figsize=(8, 3))
ax0.plot(x_tip, tip_lo, 'b', linewidth=0.5, linestyle='--', )
ax0.plot(x_tip, tip_md, 'g', linewidth=0.5, linestyle='--')
ax0.plot(x_tip, tip_hi, 'r', linewidth=0.5, linestyle='--')
ax0.fill_between(x_tip, tip0, aggregated, facecolor='Orange', alpha=0.7)
ax0.plot([tip, tip], [0, tip_activation], 'k', linewidth=1.5, alpha=0.9)
```

```
ax0.set_title('Aggregated membership and result (line)')
# Turn off top/right axes
for ax in (ax0,):
      ax.spines['top'].set_visible(False)
      ax.spines['right'].set_visible(False)
      ax.get_xaxis().tick_bottom()
      ax.get_yaxis().tick_left()
plt.tight_layout()
```



Final thoughts

The power of fuzzy systems is allowing complicated, intuitive behavior based on a sparse system of rules with minimal overhead. Note our membership function universes were coarse, only defined at the integers, but fuzz.interp_membership allowed the effective resolution to increase on demand. This system can respond to arbitrarily small changes in inputs, and the processing burden is minimal.

12.8.4. The Tipping Problem – Fuzzy Control Systems

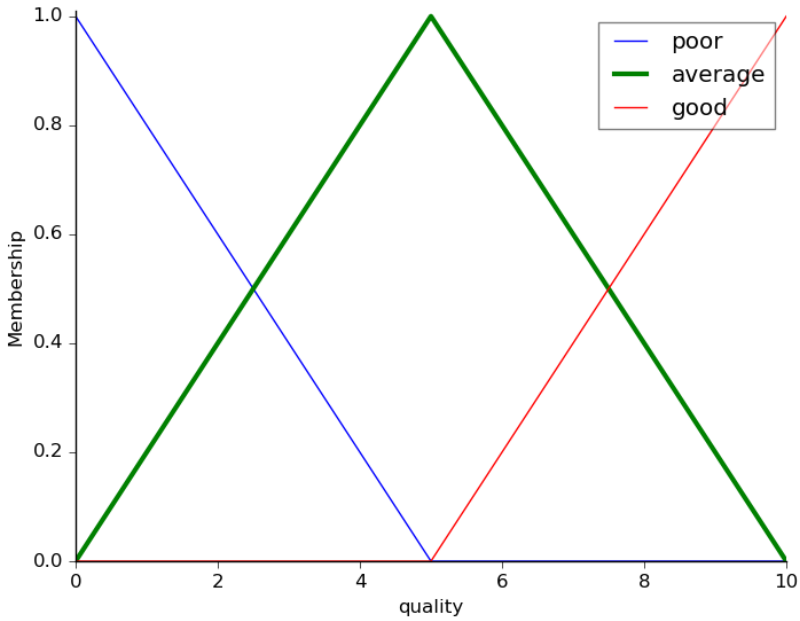
We can use the skfuzzy control system API to model this. First, let's define fuzzy variables

```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl
# New Antecedent/Consequent objects hold universe variables and
# membership functions
quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')
service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')
tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')
# Auto-membership function population is possible with .automf(3,5,or 7)
quality.automf(3)
service.automf(3)
# Custom membership functions can be built interactively with a
# familiar, Pythonic API
tip['low'] = fuzz.trimf(tip.universe, [0, 0, 13])
```

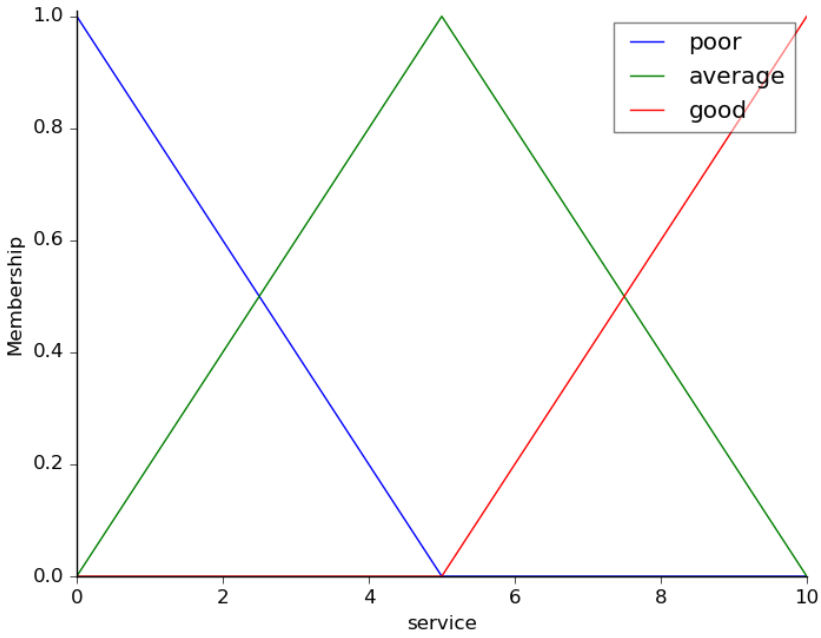
```
tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])
tip['high'] = fuzz.trimf(tip.universe, [13, 25, 25])
```

To help understand what the membership looks like, use the view methods.

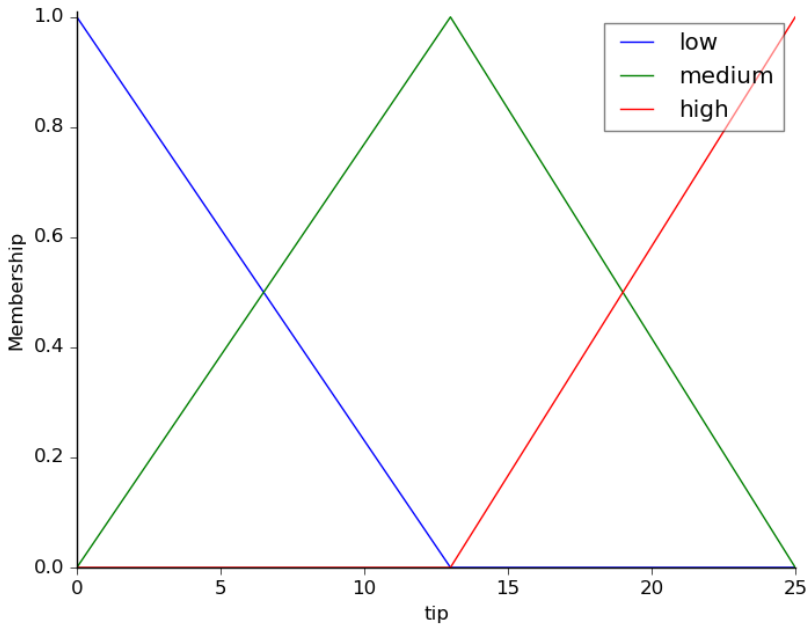
```
# You can see how these look with .view()
quality['average'].view()
```




```
service.view()
```



```
tip.view()
```



Fuzzy rules

Define the fuzzy rules as defined in the earlier section:

```
rule1 = ctrl.Rule(quality['poor'] | service['poor'], tip['low'])
rule2 = ctrl.Rule(service['average'], tip['medium'])
rule3 = ctrl.Rule(service['good'] | quality['good'], tip['high'])
```

Now that we have our rules defined, we can simply create a control system via:

```
tipping_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
```

In order to simulate this control system, we will create a ControlSystemSimulation. Think of this object representing our controller applied to a specific set of cirucmstances. For tipping, this might be tipping Sharon at the local brew-pub. We would create another ControlSystemSimulation when we're trying to apply our tipping_ctrl for Travis at the cafe because the inputs would be different.

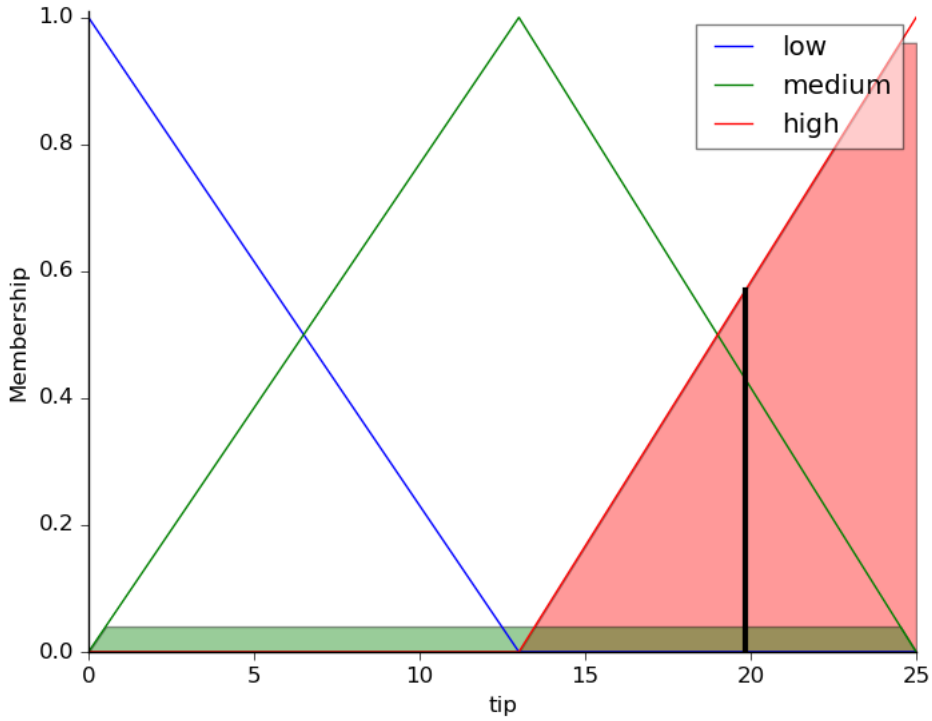
```
tipping = ctrl.ControlSystemSimulation(tipping_ctrl)
```

We can now simulate our control system by simply specifying the inputs and calling the compute method. Suppose we rated the quality 6.5 out of 10 and the service 9.8 of 10.

```
# Pass inputs to ControlSystem using Antecedent labels with Pythonic API
tipping.input['quality'] = 6.5
tipping.input['service'] = 9.8
# Crunch the numbers
tipping.compute()
```

Once computed, we can view the result as well as visualize it.

```
print tipping.output['tip']
tip.view(sim=tipping)
```



The resulting suggested tip is 20.24%.

Final thoughts

The power of fuzzy systems is allowing complicated, intuitive behavior based on a sparse system of rules with minimal overhead. Note our membership function universes were coarse, only defined at the integers, but fuzz.interp_membership allowed the effective resolution to increase on demand. This system can respond to arbitrarily small changes in inputs, and the processing burden is minimal.

12.9. Exercise

Exercise 12.1.

Consider the Room cooler example again.

Ask the instructor to provide each of you individually with a temperature and fan speed.

Determine the water flow rate for the given fan speed and temperature using fuzzy logic and SciKit Fuzzy library.

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology (GIKI)

Faculty of Computer Science & Engineering (FCSE)

CS351L – Introduction to Artificial Intelligence Lab Manual

PREPARED BY DR. SAJID, DR. HASHIM & MR. ARSALAN