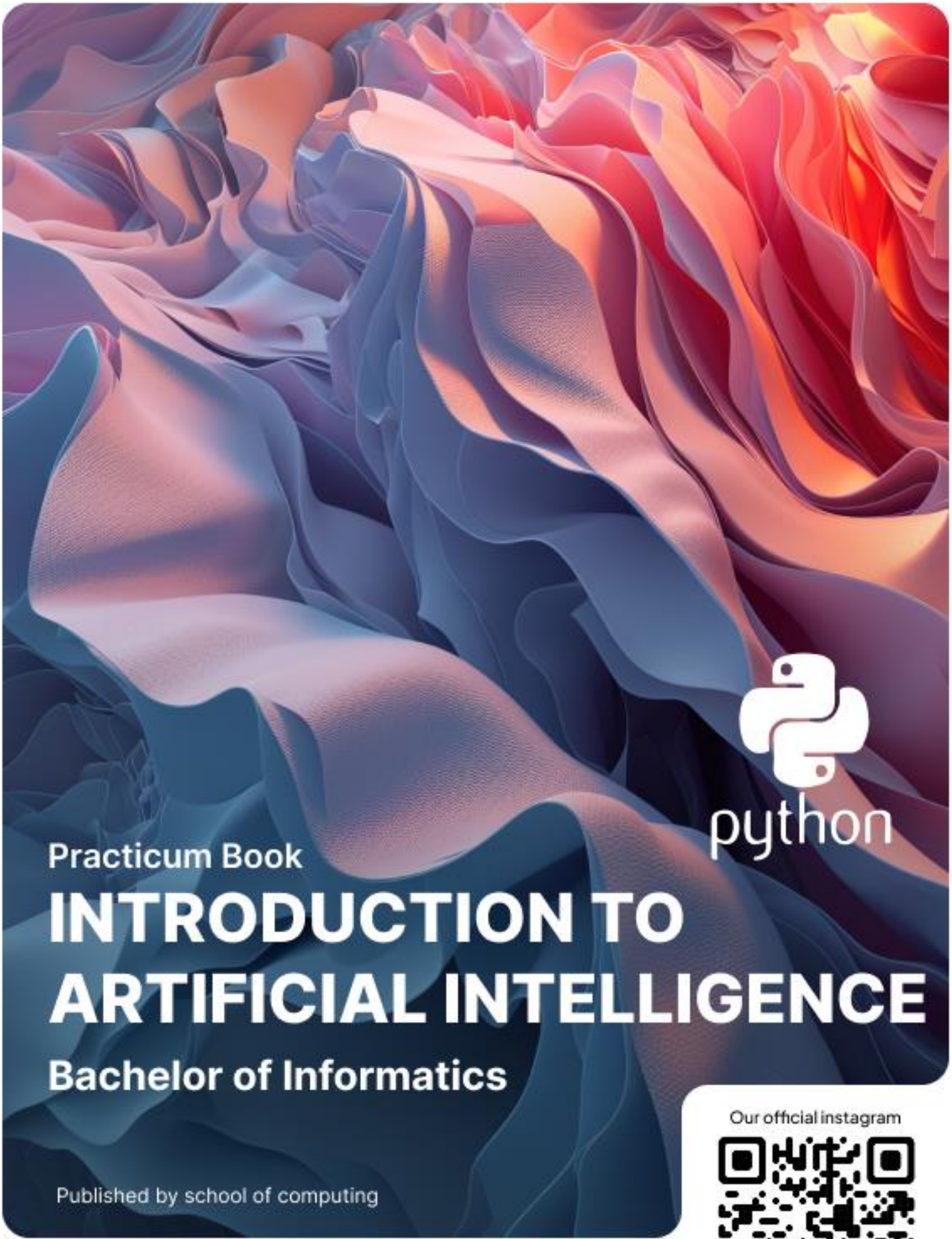


Practicum Book

INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Bachelor of Informatics

Published by school of computing

Our official instagram

@informaticslab_telu

Telkom University

Fakultas Informatika
School of Computing
Telkom University

informatics lab

Page of Approval

The following person:

Name	: SABRINA ADINDA SARI, S.Kom., M.Kom.
Worker ID. No	: 25010014
Coordinator of Practice Lecture	: Introduction to Artificial Intelligence
Study Program	: Bachelors of Informatics

Explain clearly that this module is used for the practicum implementation in the Even Semester of academic year 2025/2026 at the Informatics Laboratory of the Faculty of Informatics Telkom University.

Bandung, 07 February 2026

Approved by,	Acknowledged by,
Coordinator Lecture of Practice of	Chairperson of Undergraduate
Introduction to Artificial Intelligence	Program of Informatic
SABRINA ADINDA SARI, S.Kom., M.Kom.	Dr. MAHMUD DWI SULISTIYO, S.T., M.T.
NIP. 25010014	NIP. 13880017

Informatics Laboratory Practicum Regulations 2025/2026

1. Praktikum diampu oleh dosen kelas dan dibantu oleh asisten laboratorium dan asisten praktikum.
2. Praktikum dilaksanakan di Gedung TULT (Telkom University Landmark Tower) lantai 6 dan 7 sesuai jadwal yang ditentukan.
3. Praktikan wajib membawa modul praktikum, kartu praktikum, dan alat tulis.
4. Praktikan wajib mengecek kehadiran di igracias dan sheet yang dibagikan asisten.
5. Durasi kegiatan praktikum S-1 = 2 jam (100 menit).
6. Jumlah pertemuan praktikum:
    - 16 kali pertemuan
7. Praktikan wajib hadir minimal 75% dari seluruh pertemuan praktikum di lab.
8. Praktikan yang datang terlambat :
    - <= 5 menit : diperbolehkan mengikuti praktikum tanpa tambahan praktikuM
    - >= 30 menit : tidak diperbolehkan mengikuti praktikum
9. Saat praktikum berlangsung, asisten praktikum dan praktikan:
    - Wajib menggunakan seragam sesuai aturan institusi.
    - Wajib mematikan/ mengkondisikan semua alat komunikasi.
    - Dilarang membuka aplikasi yang tidak berhubungan dengan praktikum yang berlangsung.
    - Dilarang mengubah pengaturan *software* maupun *hardware* komputer tanpa ijin.
    - Dilarang membawa makanan maupun minuman di ruang praktikum.
    - Dilarang memberikan jawaban ke praktikan lain.
    - Dilarang menyebarkan soal praktikum.
    - Dilarang membuang sampah di ruangan praktikum.
    - Wajib meletakkan alas kaki dengan rapi pada tempat yang telah disediakan.
10. Setiap praktikan dapat mengikuti praktikum susulan maksimal dua modul untuk satu mata kuliah praktikum.
    - Praktikan yang dapat mengikuti praktikum susulan hanyalah praktikan yang memenuhi syarat sesuai ketentuan institusi, yaitu: sakit (dibuktikan dengan surat keterangan medis), tugas dari institusi (dibuktikan dengan surat dinas atau dispensasi dari institusi), atau mendapat musibah atau kedukaan (menunjukkan surat keterangan dari orangtua/wali mahasiswa.)
    - Persyaratan untuk praktikum susulan diserahkan sesegera mungkin kepada asisten laboratorium untuk keperluan administrasi.
    - Praktikan yang diijinkan menjadi peserta praktikum susulan ditetapkan oleh Lab Informatika dan tidak dapat diganggu gugat.
11. Ketidakhadiran pada kelas praktikum:
    - Nilai Modul = 0
12. Meminta, mendapatkan, dan menyebarluaskan soal dan atau kunci jawaban praktikum:
    - Penyebar soal dan kunci jawaban: Pengajuan sanksi kepada Komisi Disiplin Fakultas
    - Penerima soal dan kunci jawaban: Nilai ‘0’ pada (seluruh assessment) praktikum
13. Lupa menghapus file praktikum:

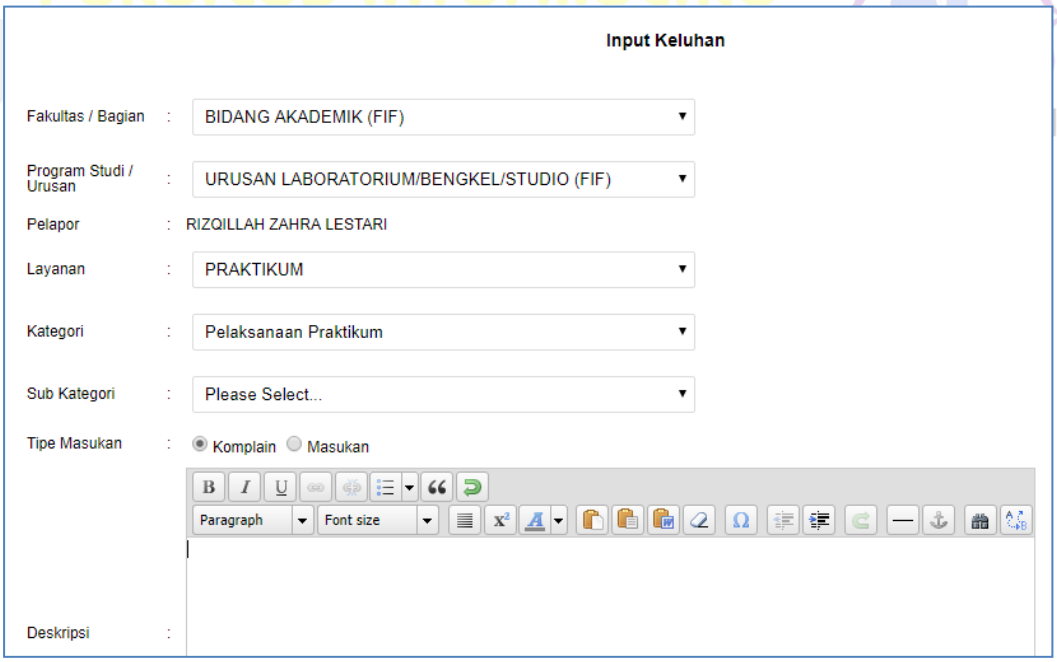
- Pengurangan nilai modul 20%

14. Memicu kegaduhan, sehingga membuat situasi tidak kondusif (jalan-jalan, mengganggu teman, mengobrol, dll), asisten praktikum diwajibkan menegur sebanyak 3x
    - Pengurangan nilai modul 50%
15. Menyalahgunakan fitur lms
    - Siap menerima sanksi lebih lanjut dari IFLAB

IFLAB Practicum Complaint Procedurs Through IGRACIAS

1. Login IGracias
2. Pilih Menu **Masukan dan Komplain**, pilih ***Input* Tiket**

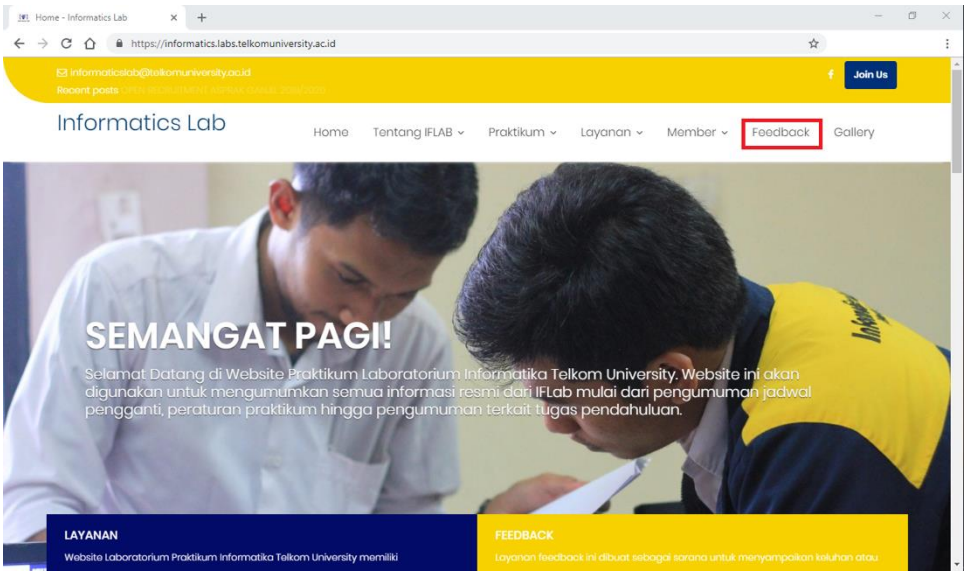
3. Pilih Fakultas/Bagian: **Bidang Akademik (FIF)**
4. Pilih Program Studi/Urusan: **Urusan Laboratorium/Bengkel/Studio (FIF**)
5. Pilih Layanan: **Praktikum**
6. Pilih Kategori: **Pelaksanaan Praktikum**, lalu pilih **Sub Kategori**.
7. Isi **Deskripsi** sesuai komplain yang ingin disampaikan.



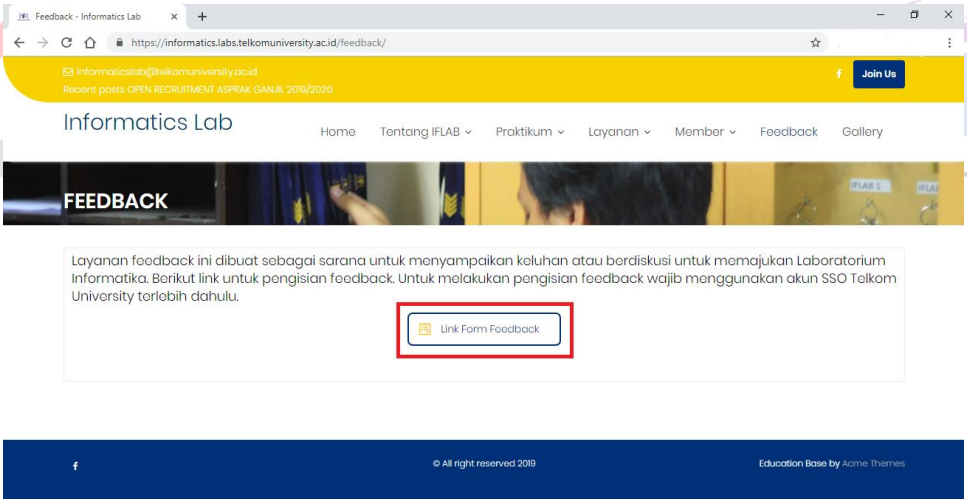
8. Lampirkan *file* jika perlu. Lalu klik Kirim.

IFLAB Practicum Complaint Procedurs Through the Website

1. Buka *website* https://informatics.labs.telkomuniversity.ac.id/ melalui *browser*.
2. Pilih menu ***Feedback*** pada *navigation bar website*.



3. Pilih tombol ***Link Form Feedback***.



4. Lakukan *login* menggunakan akun ***SSO Telkom University*** untuk mengakses *form feedback*.
5. Isi *form* sesuai dengan *feedback* yang ingin diberikan.

Table of Contents

Table of Figures

List of Tables

Modul 1 INTRODUCTION TO PYTHON

Practical Objectives:
1. Students know installation and *working environment* in Python (using Terminal and *Jupyter Notebook*). 2. Students understand how to create programs in Python. 3. Students know the main features available in Python.

1.1. Python with Anaconda

REFERENCE: https://conda.io/projects/conda/en/latest/user-guide/index.html

Anaconda is a Python distribution that is popular in data science and artificial intelligence (AI) development. Anaconda provides environment an isolated platform for running various Python projects, complete with management package that's easy. Anaconda is equipped with package built-in useful for implementing AI algorithms such as NumPy, Scikit-Learn, NetworkX, SymPy, and others.

Anaconda can be downloaded at *link*: https://www.anaconda.com/download

Installation steps are listed below *link*: https://docs.anaconda.com/anaconda/install

1.2. Creating Programs in Python

REFERENCE: https://docs.python.org/3/tutorial/index.html

Python is *high-level programming language* which is popular because of its simple and easy to read syntax. Python is often used in the development of artificial intelligence and data science because Python has *library* which is widely used and easy to use to support data processing and machine learning model development.

1.2.1. Opening Python with Terminal / CMD

REFERENCE: https://docs.python.org/3/tutorial/introduction.html

Python programs can be run through the Terminal by running commands `'python'` on Terminal / *Command Prompt*. Figure *1* hows the Terminal view after opening Python.

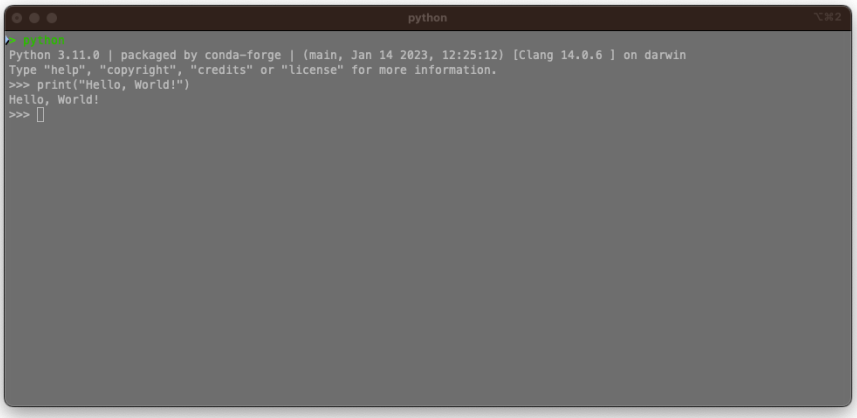


Figure 1. Terminal display after executing the command `'python'`.

1.2.2. Opening Python with Jupyter Notebook

Python programs can also be run from Jupyter Notebook. Jupyter Notebook allows Python to be run with the format Python Notebook. To open a Python program with Jupyter Notebook, do the following steps:

1. Open **Anaconda Prompt** on Windows or **Terminal** on MacOS.
2. On **MacOs**, Jupyter Notebook can be opened by running the command `` `jupyter notebook /Users/[User]/Documents` `` to open *Jupyter Notebook* on folders *Document*. Change `` `[user]` `` by the name *user* which is used. While on **Windows,** Jupyter Notebook can be opened with `` `jupyter notebook C:\Users\[User]\Documents` ``**.**
3. Jupyter Notebook will open automatically on browser with address `` `localhost:8888/tree` ``.
4. Figure *2* shows a view of *Jupyter Notebook*.

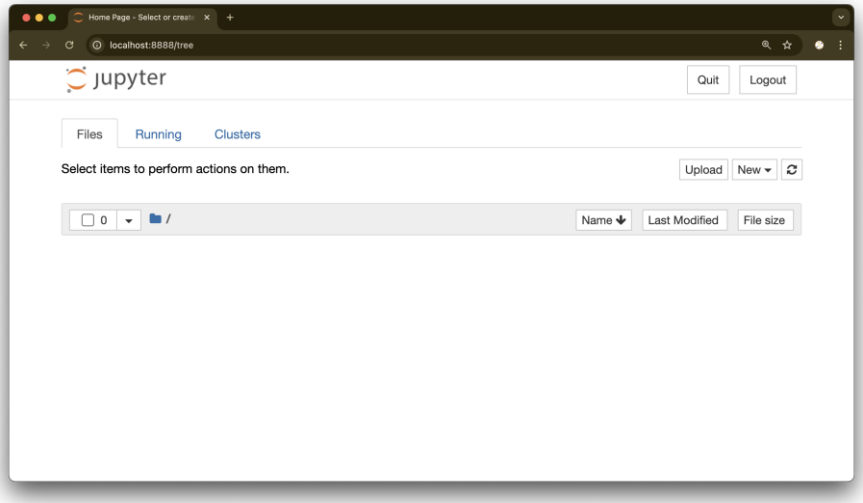


Figure 2. Initial appearance of Jupyter Notebook.

5. To create a new folder, click the button `new`, then select `Folder`. The button for creating a new folder is located as in Figure *3*.

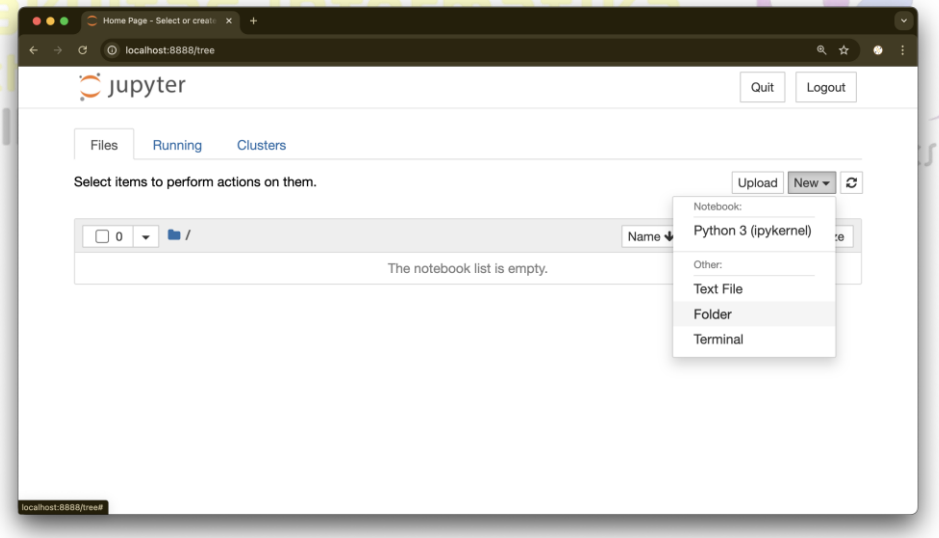


Figure 3. Menu display for creating a new folder in Jupyter Notebook.

6. To change the folder name, check the folder then click the button `Rename`, which is located on the upper left as shown by the Figure *4*(a), then write the name of the folder to be replaced on *pop-up* which appears in Figure *4*(b).

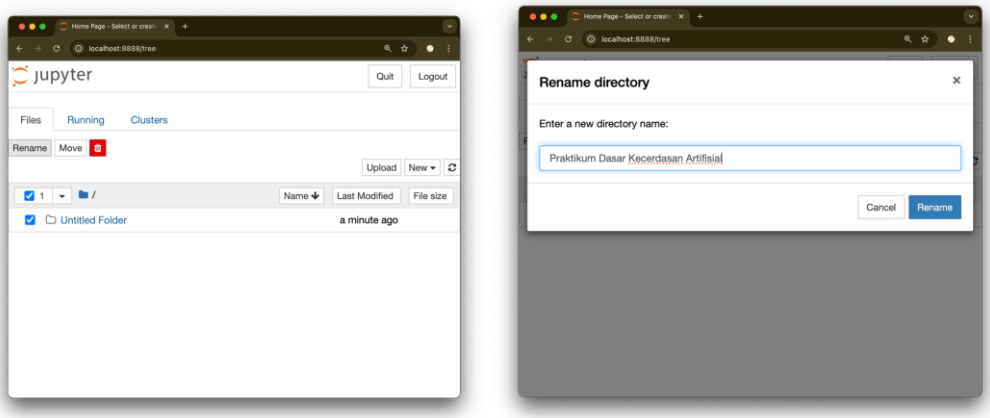


Figure 4. Menu display for changing the folder name. (a) Menu to select the folder to be renamed. (b) Menu to enter a new name for the selected folder.

7. To create a Python Notebook, click the button `new` at the top right and select `Python 3 (ipykernel)` as shown in Figure *5*.

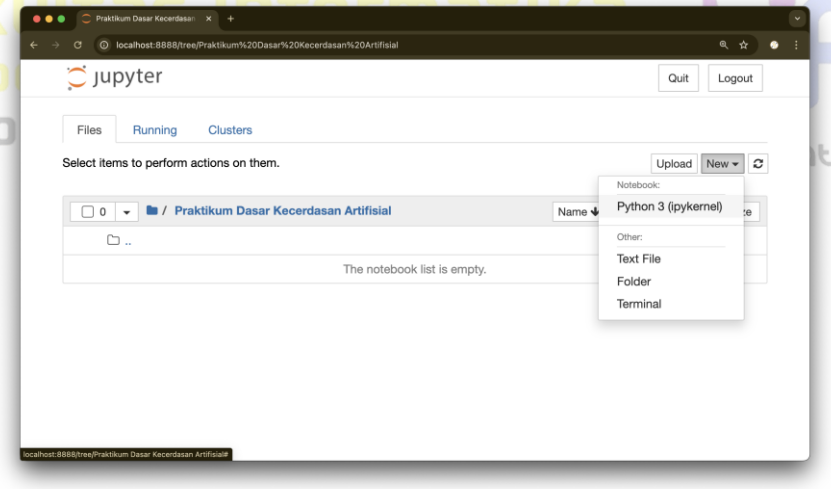


Figure 5. Menu display for creating a new Python Notebook file.

8. If done, the Python Notebook will open and you can write code in the Notebook as shown in the Figure *6*.

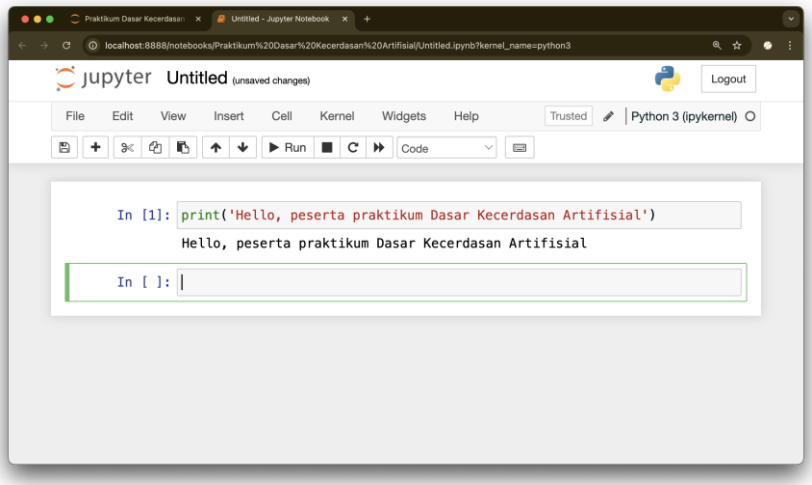


Figure 6. Python Notebook menu display.

The HelloWorld program uses the Python Terminal

```
>>> print("Hello, Fundamentals Artificial Intelligence practicum participants!")
Output:
Hello, Fundamentals Artificial Intelligence practicum participants!
```

The HelloWorld program uses Jupyter Notebook

```
[1]: print("Hello, Fundamentals Artificial Intelligence practicum participants!")
Output:
Hello, Fundamentals Artificial Intelligence practicum participants!
```

1.2.3. Python as a Calculator

Python follows standard mathematical operation precedence rules when executing arithmetic expressions. This rule is also known as **PEMDAS** (*Parentheses*, *Exponents*, *Multiplication and Division*, *Addition and Substraction*). Python operation priorities are listed in Table *1*.

Table 1. Sequence of arithmetic operations performed in Python.

Priority	Operator	Operation Type
1	( )	Parentheses
2	**	Exponents
3	*, /, //, %	Multiplication, Division, Integer Division, Modulus
4	+, -	Addition, Subtraction

Here is an example of using Python as a calculator:

>>>	**2 + 2**
	Output: 4
>>>	**5 * 6**
	Output: 30
>>>	**10 - 5**
	Output: -5
>>>	**(3 + 2) * 5**
	Output: 25
>>>	**5 – 3 * 2**
	Output: -1

The following is an example of a divide and divide operation *integer* in Python:

>>>	**10 / 2** # pembagian
	Output: 5.0
>>>	**10 // 2** # pembagian integer
	Output: 5
>>>	**15 / 4** # pembagian
	Output: 3.75
>>>	**15 // 4** # pembagian integer
	Output: 3

1.2.4. Data Types in Python

Data types in Python are categories or types of data that determine how the data can be used and operated on in the program. Python has several main data types, including:

A. Numerical Data Type

The numeric data type in Python is a data type used to store numeric values. Numerical data types and examples are in the Table *2*.

Table 2. Numerical data types and their descriptions in Python.

Data Type	Example	Information
Integer (`int`)	10, -5, 0	Used to store integers.
Float (`float`)	3.14, -0.001, 2.0	Used to store decimal / real numbers.

Complex (`complex`)	1 + 2i, 3j	Used to store complex numbers, which has a real number and an imaginary part.

B. Data Type *String*

String in Python is a data type used to represent text. String is a sequence of characters defined in quotation marks ('…' / "…"). Here's how to define it string:

[1]:	**str = "Hello, World!"** **print(str)**
	Output: Hello, World!

There are several operations that can be performed on *string* in Python, including:

Merger *Concatenation*)	
[1]:	**str1 = "Hello"** **str2 = "World"** **result = str1 + " " + str2** **print(result)**
	Output: Hello World
Looping (*Repetition*)	
[2]:	**str = "Hello"** **result = str * 3** **print(result)**
	Output: HelloHelloHello
Chracter access	
[3]:	**str = DKA** **first_char = str[0]** # indeks dimulai dari 0 **last_char = str[-1]** # indeks -1 adalah indeks terakhir **print(first_char, last_char)**
	Output: D A
String slicing	
[4]:	**str = "Hello, World!"** **sliced = str[0:5]** # Mengambil karakter ke 0 – 5 **print(sliced)**
	Output: Hello
String length	
[5]:	**str = "Hello, World!"** **print(len(str))**
	Output: 13

Strings in Python also have *method* built-in to manipulate the string itself, like so:

Kapitalisasi string	
[1]:	**str = "Hello, World!"** **print(str.upper())**
	Output: HELLO, WORLD!
Konversi ke huruf kecil	
[2]:	**str = "Hello, World!"** **print(str.lower())**
	Output: hello, world!
Menghapus spasi berlebih	
[3]:	**str = " Hello, World! "** **print(str.strip())**
	Output: Hello, World!
Mengganti bagian string	
[4]:	**str = "Hello, World!"** **replaced = str.replace("World", "Dunia")** # .replace([old], [new]) **print(replaced)**
	Output: Hello, Dunia!

C. Data Type *List*

List is a data structure used to store a group of elements in one variable. List can contain various types of data, such as numbers, string, or even list other. List defined using square brackets `[]`, with elements within separated by commas. The following is an example of a definition list:

[1]:	**list_of_numbers = [1,3,5,10]** **print(list_of_numbers)**
	Output: [1, 3, 5, 10]
[2]:	**list_of_strings = ["dasar", "kecerdasan", "artifisial"]** **print(list_of_strings)**
	Output: ['dasar', 'kecerdasan', 'artifisial']
[3]:	**list_of_items = [1, "hello", ["ini", "list"]]** **print(list_of_items)**
	Output: [1, 'hello', ['ini', 'list']]

Indexes are used to access elements on list. The first index is `0`, and the final index is `-1`.

[1]:	**my_list = [1, 3, 5, "DKA", True]** # Fetching elements based on index **first_elm = my_list[0]** **second_elm = my_list[1]** **last_elm = my_list[-1]**

	second_last_elm = my_list[-2] # Print results **print(first_elm, second_elm, last_elm, second_last_elm)**
	Output: 1 3 True 'DKA'
[2]:	**my_list = [1, 3, 5, "DKA", True]** **elms = my_list[1:4] # Mengambil anggota ke 1 – 4 dalam list** **print(elms)**
	Output: [3, 5, 'DKA']
[3]:	**numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]** **first_five = numbers[:5]** # Take the first 5 members **last_three = numbers[-3:]** # Takes the last 3 members **print(first_five)** **print(last_three)**
	Output: [1, 2, 3, 4, 5] [8, 9, 10]
[4]:	**numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]** **numbers[0] = 24** # Change the value at index to 0 **print(numbers)**
	Output: [24, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Python has methods list comprehension which is a short and elegant way to create list just from list existing ones or iterable others such as range or string. List comprehension makes it possible to write loop in one line in one list. list comprehension can be written with the following syntax:

[**expression** for **item** in **iterable** if **condition**]
`**expression**`: The value or operation to be inserted into the new list. `**item**`: Variable that takes each element from `iterable`. `**iterable**`: Iterable objects such as lists, ranges, strings, etc. `**condition**`: (optional) Condition for including elements in a new list.

Here is a simple example of usage list comprehension:

Membuat *list* dari *range* menggunakan *list comprehension*	
[1]:	**squares = [x**2 for x in range(10)]** **print(squares)**
	Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
***List comprehension* dengan kondisi**	
[2]:	**even_nums = [x for x in range(10) if x % 2 == 0]** **print(even_nums)**
	Output: [0, 2, 4, 6, 8]

List comprehension* dengan operasi pada *string	
[3]:	**fruits = ["apple", "banana", "cherry"]** **uppercased = [fruit.upper() for fruit in fruits]** **print(uppercased)**
	Output: ['APPLE', 'BANANA', 'CHERRY']
Nested list comprehension	
[4]:	**pairs = [(x,y) for x in range(2) for y in range(2)** **print(pairs)**
	Output: [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]

D. Data Type *Dictionary*

Dictionary in Python is a data type used to store pairs key-value. Each `key` in dictionary is for and used to access `value` which is related. Dictionary defined using curly brackets `{}` as follows:

[1]:	**my_dict = {** **"nama": "Fizz",** **"umur": 21,** **"kota": "Bandung",** **"universitas": "Telkom University"** **}** **print(my_dict["nama"])** **print(my_dict["kota"])**
	Output: Fizz Bandung

There are several operations that can be performed on *dictionary*, including:

Add or update values	
[1]:	**my_dict = {** **"nama": "Fizz",** **"umur": 21,** **"kota": "Bandung",** **"universitas": "Telkom University"** **}** # Added new items **my_dict["negara"] = "Indonesia"** **print("Setelah ditambah data negara")** **print(my_dict)** # Update values **print("Setelah nilai umur diperbarui")** **my_dict["umur"] = 23** **print(my_dict)**
	Output: After adding country data

	{
	'name': 'Alice',
	'age': 21,
	'Bandung',
	'university': 'Telkom University',
	'Indonesian country'
	}
	Once the lifetime value is updated
	{
	'name': 'Alice',
	'age': 26,
	'Bandung',
	'university': 'Telkom University',
	'Indonesian country'
	}

Delete an element	
[2]:	**my_dict = {** **"nama": "Fizz",** **"umur": 21,** **"kota": "Bandung",** **"universitas": "Telkom University"** **}** # method 1 **del my_dict["kota"]** # Penghapusan data kota # method 2 # Deleting city data and entering values into variables **kota = my_dict.pop("kota")** **print(my_dict)**

	Output:
	{
	'nama': 'Alice',
	'umur': 26,
	'universitas': 'Telkom University'
	}

Access all *keys* or *values*	
[3]:	**my_dict = {** **"nama": "Fizz",** **"umur": 21,** **"kota": "Bandung",** **"universitas": "Telkom University"** **}** **kunci = my_dict.keys()** **nilai = my_dict.values()** **print("kunci:", kunci)** **print("nilai:", nilai)**
	Output: key: ['name', 'age', 'city', 'university'] values: ['Fizz', 21, 'Bandung', 'Telkom University']

Deep iteration *dictionary*	
[4]:	**my_dict = {** **"nama": "Fizz",** **"umur": 21,** **"kota": "Bandung",** **"universitas": "Telkom University"** **}** **for key, value in my_dict.items():** **print("Value of", key, "is", value)**
	Output: The value of the name is Fizz The value of age is 21 The value of the city is Bandung The value of the university is Telkom University
Checking for presence *key*	
[5]:	**my_dict = {** **"nama": "Fizz",** **"umur": 21,** **"kota": "Bandung",** **"universitas": "Telkom University"** **}** # Checks whether the key 'name' exists in my_dict **if "nama" in my_dict:** **print("Name key found")** **else:** **print("Name key not found")** # Checks whether the key 'country' exists in my_dict **if "negara" in my_dict:** **print("Country key found")** **else:** **print("Country key not found")**
	Output: Name key found Country key not found

E. Type Date *Set*

Set in Python it is a data type used to store a collection of elements that are unique and unordered. That is, every element in *set* must be different and not duplicate. *Set* defined using parentheses curly `{}` or by using `set()` like the example below:

[1]:	# Create an empty set **my_set = set()** # Create sets with multiple elements **my_set = {1, 2, 3, "apple", 3.14}** **print(my_set)**
	Output:

	{1, 2, 3, 3.14, 'apple'}

There are several operations that can be performed on *set*, including:

Adding elements	
[1]:	**my_set = {1, 2, 3, "apple", 3.14}** **my_set.add("orange")** **print(my_set)**
	Output: {1, 2, 3.14, 3, 'apple', 'orange'}
Delete an element	
[2]:	**my_set = {1, 2, 3, "apple", 3.14}** # Will error if apple is not set **my_set.remove("apple")** # There will be no error if banana is not set **my_set.remove("banana")** **print(my_set)**
	Output: {1, 2, 3.14, 3}
Combine *set* (*union*)	
[3]:	**set_1 = {1, 2, 3}** **set_2 = {3, 4, 5}** **union_set = set_1 | set_2** **print(union_set)**
	Output: {1, 2, 3, 4, 5}
Slice *set* (*intersection*)	
[4]:	**set_1 = {1, 2, 3}** **set_2 = {3, 4, 5}** **intersection_set = set_1 & set_2** **print(intersection_set)**
	Output: {3}
Difference *set* (*difference*)	
[5]:	**set_1 = {1, 2, 3}** **set_2 = {3, 4, 5}** **difference_set = set_1 - set_2** **print(difference_set)**
	Output: {1, 2}
Symmetric difference	
[6]:	**set_1 = {1, 2, 3}** **set_2 = {3, 4, 5}** **symmetric_difference_set = set_1 ^ set_2**

	`print(symmetric_difference_set)`
	Output: {1, 2, 4, 5}
Check membership	

[7]:	`set_1 = {1, 2, 3}`
	
	`if 3 in set_1 :`
	`  print("3 is in set_1")`
	`else :`
	`  print("3 is not in set_1")`

	Output: 3 are in set_1
Counting the number of elements	

[8]:	`set_1 = {1, 2, 3}`
	
	`print(len(set_1))`

	Output: 3

F. Data Type *Boolean*

Boolean is a data type in Python that has only two values: `True` And `False`. data type Boolean used in logical operations and conditions to organize program flow, such as in statements `if` And `while`. `True` And `False` also represented by value `1` And `0`. Here is an example of usage Boolean:

Simple Boolean Value	

[1]:	`is_student = True`
	`informatics_student = False`
	
	`print(is_student, informatics_student)`

	Output: True False
Comparison operations	

[2]:	`x = 10`
	`y = 20`
	
	`print(x < y)` # True
	`print(x > y)` # False

	Output: True False
Logic operations	

[3]:	`a = True`
	`b = False`
	
	`print(a and b)` # False
	`print(a or b)` # True
	`print(not a)` # False

	Output:

	False True False
Use *Boolean* in condition	
[4]:	**is_raining = True** **if is_raining:** ** print("Bring an umbrella!")** **else:** ** print("No umbrella needed.")**
	Output: Bring an umbrella!

1.3. *Input And Output On Python*

A. Input On Python

`input()` is a built-in function in Python that is used to receive input from the user. When function `input()` is called, program execution will stop and wait for the user to enter data. Once the user types the data and presses `Enter`, the data will be returned as ***string*** by function `input()`. Here is how to use `input()`:

[1]:	# Asks the user to enter a name **name = input("Enter your name: ")** **print("Hello," name)**
	Output: Enter your name: Fizz Hello, Fizz
[2]:	# Request and convert input from the user **age = int(input("Enter your age: "))** # Casting to integer **print("Next year, your age will be:", age+1)**
	Output: Enter your age: 21 Next year, you will be 22

B. Output On Python

Output in Python refers to the process of displaying data or information to the user, usually via a terminal. Main functions used output in Python is `print()`. This function allows Python to display text, numbers, or the value of a variable. The following is an example of usage `print()`:

Usage simple `print()`	
[1]:	**print("Hello, World!")** **name = "Fizz"** **print("My name is", name)**
	Output: Hello, World!

	`My name is Fizz`
Use of Formats `%`	

`[2]:`	`name = "Fizz"`
	`age = 21`
	
	`print("My name is %s and I'm old %d year." % (name, age))`
	
	`# %s : used to insert strings`
	`# %d : used to insert an integer`
	`# %f : used to insert float`

	Output: My name is Fizz and I am 21 years old.
Using method `str.format()`	

`[3]:`	`name = "Fizz"`
	`age = 21`
	
	`print("My name is {} and I am {}`
	`year.".format(name, age))`

	Output: My name is Fizz and I am 21 years old.
Using F-Strings	

`[4]:`	`name = "Fizz"`
	`age = 21`
	
	`pruns (f"My name is {name} and I am {age} year.")`

	Output: My name is Fizz and I am 21 years old.

1.4. Range Statement

`range` is a Python built-in function that is used to generate a series of numbers incrementally. `range` generally used in *loop* like `for` to iterate through a sequence of numbers. This function is used when you need something repeatedly for a certain number of iterations. The following is an example of usage `range`:

range(start, stop, step)
`start`: (optional) Initial number. *Default* is 0. `stop`: Final figure (not included in results). `step`: (optional) The size of the step between numbers. *Default* is 1.

Here is an example of using `range`:

Using results `range` **as a list**	
`[1]:`	`numbers = list(range(10))` `print(numbers)`
	Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Usage `range` **in** `for`	
`[2]:`	`fruits = ["apple", "orange", "tomato", "papaya", "pear"]`

	for i in range(0, len(fruits), 2) : ** print(fruits[i])**
	Output: apple tomato pear
`range` **with negative steps**	
[3]:	**for i in range(3, 0, -1) :** ** print(i)**
	Output: 3 2 1

1.5. Branching in Python

Branching in Python is a control structure that allows a program to make decisions based on certain conditions. By using branching, a program can execute a specific block of code if a given condition holds `True`, or take an alternative route if these conditions arise `False`.

1.5.1. Branching `if`

`if` used to execute a block of code only if a given condition is valid `True`. Here is the syntax and usage example *statement* `if`:

Syntax `if`	
if condition : ** # Code that is executed if the condition is True**	
Contoh 1	
[1]:	**x = 10** **if x > 5 :** ** print("x is greater than 5")**
	Output: x is greater than 5
Contoh 2	
[2]:	**x = 10** **if x > 3 :** ** print("x is greater than 3")** **if x > 5 :** ** print("x is greater than 5")**
	Output: x is greater than 3 x is greater than 5

1.5.2. Branching `if-else`

`else` used to execute a block of code if all conditions `if` And `elif` above it is worth `False`. `else` must be at the end of the branching block. Following is an implementation and branching example `if-else`:

Syntax `if-else`

```
if condition :
   # Code that is executed if the condition is True
else :
   # Code that is executed if the condition is False
```

Example 1

[1]:
```
x = 10

if x < 5 :
    print("x kurang dari 5")
else :
    print("x is greater than or equal to 5")
```

Output:
```
x is greater than or equal to 5
```

Example 2

[2]:
```
x = 10

if x < 5 :
    print("x is less than 5")
elif x < 7 :
    print("x is less than 7")
else :
    print("x is greater than or equal to 7")
```

Output:
```
x is greater than or equal to 7
```

1.5.3. Branching `if-elif-else`

`elif` tatement (stand for "else if") is used to check another condition if the previous condition is valid `False`. Statement `elif` more than one can be made. Following is the syntax and usage example statement `if-elif-else`:

Syntax `if-elif-else`

```
if condition_1 :
    # Code that is executed if condition_1 is True
elif condition_2 :
    # Code to execute if condition_2 is True
else :
    # Code that is executed if condition_1 and condition_2 are False
```

Example 1

[1]:
```
x = 10

if x < 5 :
    print("x is less than 5")
else :
```

	`print("x is greater than or equal to 5")`
	Output: x is greater than or equal to 5
Example 2	

[2]:	`x = 8`
	
	`if x < 5 :`
	`  print("x is less than 5")`
	`elif x < 7 :`
	`  print("x is less than 7")`
	`elif x < 9 :`
	`  print("x is less than 9")`
	`else :`
	`  print("x is greater than or equal to 7")`

	Output: x is less than 9

1.6. Repetition While

Repetition `while` in Python it is a control structure that allows code to be executed repeatedly as long as a certain condition remains met. This loop continues to run as long as the specified condition is valid `True`, and will stop once the condition is valued `False`. Following is the syntax and example of looping `while`:

Syntax `while`
`while condition :` `  # Block of code to be executed during the True condition`
Example 1

[1]: `count = 1`
 	
`while count <= 3 :`
`  print(count, end=' ')`
`  count += 1`

Output: 1 2 3
Example 2

[2]: `number = 10`
 	
`while number > 0:`
`  number -= 1`
`  if number == 5 :`
`  print("Stopped at 5")`
`  break # Forcibly stops the loop`

Output: Stop at 5

1.7. Repetition For

Repetition `for` in Python it is a control structure used to iterate or repeat the execution of a block of code based on the sequence of elements of an iterable, such as *list*, *tuple*, *string*, or *range*. Following is the syntax and example of looping `for`:

Syntax `for`

```
for elemen in iterable :
    # Block of code to be executed during the True condition
```

Iterating over the list

[1]:
```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits :
    print(fruit)
```

Output:
```
'apple'
'banana'
'cherry'
```

Iterating over strings

[2]:
```
for char in "DKA" :
    print(char)
```

Output:
```
D
K
A
```

Usage `range()` **in** `for`

[3]:
```
for i in range(5) :
    print(i)
```

Output:
```
0
1
2
3
4
```

1.8. Function

Functions in Python are blocks of code designed to perform a specific task and can be called at any time in the program. The use of functions makes it possible to group frequently used code or complex code into smaller, more manageable units.

To define functions in Python, keywords are used `def`, followed by the function name, parentheses `()` which can contain parameters, and colons `:`. The block code included in the function must be entered with indentation. Here is the syntax and example of the function in Python:

Syntax fungsi

```
def function_name(parameter1, parameter2, …) :
    # Function code block
```

	`return hasil # (optional) Returns the result value`
Function without parameter	

[1]:	`def hello_world():`
	`  print("Hello, World!")`
	`# call the function`
	`hello_world()`

	Output: Hello, World!
Function with parameter	

[2]:	`def greet(name):`
	`  print(f"Halo, {name}!")`
	
	`greet("Fizz")`

	Output: Halo, Fizz!
Function with *return value*	

[3]:	`def add(a, b):`
	`  return a + b`
	
	`result = add(5,3)`
	`print(result)`

	Output: 8

1.8.1. Parameters and Arguments in Functions

Parameters are variables defined in parentheses when declaring a function. Arguments are the actual values passed to the parameters when the function is called. The following is an example of using parameters and arguments in a function:

Use of parameters and arguments	

`def subtract(x, y): # x and y are parameters`
`  return x - y`
 	
`result = subtract(10, 3) # 10 and 3 are arguments`
`print(result)`	

Function with default parameters Default parameters are parameters that have a default value, which is used if no arguments are provided when the function is called.	

[1]:	`# name is the positional argument, message is the keyword argument`
	`def greet(name, message="Welcome!"):`
	`  print(f"{message}, {name}")`
	
	`greet("Fizz") # Use the value for the default message`
	`greet("Buzz", "Halo") # Override the default value for message`

	Output: Welcome, Fizz Halo, Buzz
Functions with args parameters The args parameter is used to accept an unlimited number of positional arguments.	
[2]:	`def sum(*args):` `  return sum(args)`

	`print(sum(1, 2, 3, 4, 5))`
	Output: 15
Function with kwargs parameters The kwargs parameter is used to receive a number *keyword argument* which is unlimited.	

[3]:	`def print_info(**kwargs):`
	`  for key, value in kwargs.items():`
	`  print(f"{key}: {value}")`
	
	`print_info(nama="Fizz", umur=21, universitas="Telkom University")`

	Output: nama: Fizz umur: 21 universitas: Telkom University

1.8.2. Anonymous Function

Python also supports anonymous functions with keywords `lambda`, which is a small function defined without a name. *Lambda function* usually used for simple operations. Here is an implementation and example of an anonymous function:

[1]:	`# Lambda function to add two numbers`
	`add = lambda x, y: x + y`
	`print(add(3,4))`

	Output: 7

1.9. Modular Programming

Modular programming is a programming approach in which a program is divided into small parts called **module**. Each module is a separate piece of code that handles a specific function in the program. By separating code into smaller, organized modules, programs become easier to understand, manage, and develop. Here is how to implement modular programming in Python:

File `matematika.py`	

[1]:	`def tambah(a, b):`
	`  return a + b`
	
	`def kali(a, b):`
	`  return a * b`

File `main.py`	

[2]:	`import matematika # Import all functions in mathematika.py`
	
	`# Using functions from math.py`
	`hasil_tambah = matematika.tambah(5, 3)`
	`hasil_kali = matematika.kali(4, 2)`
	
	`print("Hasil tambah:", hasil_tambah)`

	`print("Hasil kali:", hasil_kali)`
	Output: 8 8

1.10. Python Programming Practice Questions

1. Create a function (*function*) isLeap in Python, which is a function to determine whether a number of years is a leap year or not. *Input* of this function is *integer* (year number), the output of this function is *boolean* (`True` when it's leap, `False` if not leap). Next call the function and check the results.

2. Create a function and a procedure to print numbers *Fibonacci* using Python! For *input* `n`, then this function returns a number *Fibonacci* nth, while the procedure will print a sequence of numbers *Fibonacci* starting from the 1st term to the nth term. Next, call the functions and procedures and check the results.

Modul 2 INTRODUCTION LIBRARY NUMPY

Practical Objectives:
1. Students understand how to manage and process computing array with library NumPy. 2. Students know the main features available in library Numpy.

REFERNCE: https://numpy.org/doc/2.0/

NumPy is library fundamentals in the Python programming language used to perform numerical computations with high efficiency. Library it provides support for array multidimensional, which allows data processing more quickly and efficiently compared to list Regular Python. NumPy is also equipped with various mathematical functions and linear algebra operations that are very useful in data analysis, image processing, as well as artificial intelligence modeling and machine learning. NumPy's ability to perform vector and matrix operations optimally makes it an essential tool for developers in managing and analyzing large-scale data.

2.1. Method *Import* NumPy

Numpy should already be present in your Anaconda installation. To use the NumPy module, it must be done import first. Here is the standard way to import library Numpy:

```
import numpy as np
```

2.2. Definition Array And *Array* NumPy

Array is a data structure that stores a number of elements belonging to the same data type sequentially in memory. In Python, array can be declared using list with syntax like `x = [1, 2, 3]`, where elements - elements array placed in square brackets `[ ]` and separated by commas. Array can also be declared using library NumPy, declaration differences array familiar with array NumPy can be seen in Table *3*.

Table 3. Differences between arrays in Python and NumPy Array.

Aspect	Ordinary Arrays	Array NumPy
Declaration	`x = [1, 2, 3]`	`import numpy as np` `x = np.array([1, 2, 3])`
Data Type	Heterogeneous (can store different types of data)	Homogeneous (all elements must have the same data type)
Dimensions	1 dimension (can accommodate	Supports multiple dimensions

	nested lists to store more dimensions)	(1D, 2D, 3D, etc) out of the box
Operation Mathematics	Does not support direct operation (`[1, 2, 3] + [4, 5, 6]` will combine list)	Supports live operation (`np.array([1, 2, 3]) + np.array([4, 5, 6])` produce `[5, 7, 9]`)
Memory Usage	Less efficient (due to data type flexibility)	More efficient (with fixed memory allocation and single data type)
Functionality	Basic, especially for non-numerical operations	Supports math functions, linear algebra, statistics, etc
Slicing	Support slicing base (`x[0:2]` will produce `[1, 2]`)	Support slicing with additional features such as `x[:, 1:3]` for multidimensional arrays
Broadcasting	Does not support	Supports intermediate operations array with different sizes as long as they are compatible
Addition of Elements	Easy, use `append()` or `+`	It is necessary to use methods such as `np.append()` or necessary make array new

2.3. Create Programs with Library Numpy

The following is an implementation of use array in Python:

Merger 2 array different ones use signs `+`	
[1]:	**x = [1, 2, 3, 4]** **y = [5, 6, 7, 8]** **z = x + y** **print(z)**
	Output: [1, 2, 3, 4, 5, 6, 7, 8]
Addition of elements (append) on array using syntax `append`	
[2]:	**x = [1, 2, 3, 4]** **print("Initial condition of array x :", x)** **x.append(5, 6, 7, 8)** **print("Final condition of array x :", x)**
	Output: Initial condition of array x : [1, 2, 3, 4] Final condition of array x : [1, 2, 3, 4, 5, 6, 7, 8]

The following is an implementation of use array using NumPy. For the record **make**

sure to execute the line `import numpy as np` every time you start a new session (run time) or script Python that requires use library NumPy.

The arithmetic operation of addition on elements array NumPy uses signs `+`	
[1]:	**import numpy as np** **x = np.array([1, 2, 3, 4])** **y = np.array([5, 6, 7, 8])** **z = x + y** **print(z)** **print(z.tolist())** # Use the `tolist()` syntax to display each comma-delimited element (,)
	Output: [6 8 10 12] [6, 8, 10, 12]
Merger 2 array use different syntax `concatenate`	
[2]:	**x = np.array([1, 2, 3, 4])** **y = np.array([5, 6, 7, 8])** **z = np.concatenate((x, y))** **print(c)**
	Output: [1 2 3 4 5 6 7 8]

2.4. Attribute *Array* NumPy

Internal attributes array NumPy is a built-in property that provides important information regarding structure, size, data type, and shape array. These attributes allow you to easily access and manipulate the fundamental characteristics of array without having to perform complex manual operations.

2.4.1. Attribute ndim

The `ndim` attribute indicates the number of dimensions or levels in the array. This is one of the basic attributes that help understand the data structure the program is operating on. For example, a 1D array has an `ndim` of 1, a 2D array has an `ndim` of 2, and so on. The dimension structure of a NumPy array is described in Table *4*.

Table 4. Dimensional structure of NumPy arrays.

Amount Dimensions	Example Structure Array	Description
1D	`[1, 2, 3]`	Array one-dimensional or

		vector, contains elements in one row.
2D	`[[1, 2, 3], [4, 5, 6]]`	Array two-dimensional or matrix, containing elements in rows and columns.
3D	`[[[1, 2], [3, 4]], [[5, 6], [7, 8]]]`	Array three-dimensional, contains data in cube form with depth, rows and columns.
4D	`[[[[1], [2]], [[3], [4]]], [[[5], [6]], [[7], [8]]]]`	Array four dimensions, usually used for data with multiple channels (eg batch Figure in the model deep learning).

The following is an example of using the syntax `ndim`:

Example 1

```
[1]: import numpy as np

     x = np.array([85, 90, 78, 92, 88])

     print("Array dimensions x:", x.ndim)
```

```
Output:
Array dimensions x: 1
```

Example 2

```
[2]: x = np.array([[30, 32, 31], [29, 33, 30], [28, 34, 29]])

     print("Array dimensions x:", x.ndim)
```

```
Output:
Array dimensions x: 2
```

2.4.2. Attribute shape

Attribute `shape` shows dimensions or sizes array in form tuple. It provides information about the number of elements in each axis array. Its use is to ensure dimensions array appropriate before mathematical operations, data manipulation, or further processing, such as inputting data into an artificial intelligence model. The following is an example of using attributes `shape`. The following is an implementation and example of the shape attribute in array NumPy:

Example 1

```
[1]: import numpy as np

     x = np.array([[1, 2, 3], [4, 5, 6]])
     print("Shape x:", x.shape)
```

	Output: Shape x: (2, 3)
Explanation: **Attribute `shape` returns (2, 3), which means array has 2 rows and 3 columns.**	
Example 2	
[2]:	**x = np.array([[1, 2, 3], [4, 5, 6]])** **y = np.array([[[7, 8, 9], [10, 11, 12]], [[13, 14, 15], [16, 17, 18]]])** # Check whether x and y have the same shape **if x.shape == y.shape:** **print("Arrays x and y have the same shape.")** **else:** **print("Arrays x and y have different shapes.")** **print("Shape x:", x.shape) # Output: (2, 3)** **print("Shape y:", y.shape) # Output: (2, 2, 3)**
	Output: Arrays x and y have different shapes. Shape x: (2, 3) Shape y: (2, 2, 3)
Explanation: **`x` is array 2D with the shape (2, 3), which means 2 rows and 3 columns.** **`y` is array 3D with the shape (2, 2, 3), which means it has 2 matrices, each containing 2 rows and 3 columns.** **The first matrix is `[[7, 8, 9], [10, 11, 12]]`** **The second matrix is `[[13, 14, 15], [16, 17, 18]]`**	

2.4.3. Attribute size

Attribute `size` shows the total number of elements present in array, without regard to its dimensions or shape. The main use of this attribute is to find out the overall size array. Attribute `size` helps ensure that the number of elements in array as expected, especially when performing operations such as `reshape` or allocate memory for very large data. The following is an implementation and example of the size attribute in NumPy:

Example 1	
[1]:	**import numpy as np** **x = np.array([[1, 2, 3], [4, 5, 6]])** **print("Size x:", x.size)**
	Output: Size x: 6
Explanation: **Attribute `size` returns 6, indicating that there are 6 elements in total array `x` This.**	
Example 2	

[2]:	**x = np.array([1, 2, 3, 4, 5, 6])** **y = np.array([[5, 6], [7, 8], [9, 10]])** # Compares the sizes of both arrays **if x.size == y.size:** **print("Both arrays have the same number of elements.")** **else:** **print("Both arrays have different number of elements.")**
	Output: Both arrays have the same number of elements.
Explanation: **Attribute** `size` **helps in comparing whether two array have the same number of elements.** **In this example,** `x` **has 6 elements and y has 6 elements, so the result is "Both arrays have the same number of elements."**	

2.4.4. Attribute dtype

Attribute `dtype` indicates the data type of the inner elements array, like integer, float, or string. Its use is to ensure the elements array has data types that match processing needs, affects mathematical operations, memory allocation, and prevents data type errors when working with array big. Following is an example and implementation to check the data type of an inner element array:

Example 1 (Checking the data type of a array)	
[1]:	**import numpy as np** # Array with integer data type **x = np.array([1, 2, 3])** **print("Data type x:", x.dtype)** # Array with float data type **y = np.array([1.1, 2.2, 3.3])** **print("Data type y:", y.dtype)**
	Output: **Data type** x: int64 **Data type** y: float64
Example 2 (Ensure data type when creating array)	
[2]:	# Create an array with string data type **x = np.array(["apple", "banana", "cherry"], dtype=np.str_)** **print("Data type x:", x.dtype)** # Create an array with 64-bit float data type **y = np.array([1.1, 2.2, 3.3], dtype=np.float64)** **print("Data type y:", y.dtype)**
	Output: **Data type** x: <U6 **Data type** y: float64
Explanation:	

Specifies the dtype data type when creating an array to ensure elements array x is string. Here, <U6 indicates string Unicode with a maximum length of 6 characters.

2.5. Declaration Array with NumPy

Declaration array with NumPy it is possible to create array multidimensional with various initializations to suit data analysis needs. With functions like `np.zeros()`, `np.ones()`, `np.empty()`, `np.arange()`, `np.linspace()`, `np.full()`, `np.random.random()`, `np.random.randint()`, and `np.fromfunction()`, can easily create array with default values, a specific order, or evenly distributed values. The main use of this method is to efficiently prepare data before further processing. The following are several ways to declare array with NumPy:

Syntax `np.zeros()`	
[1]:	**x = np.zeros((2, 3))** **print(x)**
	Output: [[0. 0. 0.] [0. 0. 0.]]
Explanation: `np.zeros()` **make array with the form (2, 3) and fills all elements with zero values**	
Syntax `np.ones()`	
[2]:	**x = np.ones((3, 4))** **print(x)**
	Output: [[1. 1. 1. 1.] [1. 1. 1. 1.] [1. 1. 1. 1.]]
Explanation: `np.ones()` **make array with the form (3, 4) and fills all elements with the value one.**	
Syntax `np.empty()`	
[3]:	**x = np.empty((2, 2))** **print(x)**
	Output: [[7.34 2.4] [6.3 2.162]]
Explanation: `np.empty()` **make array with the form (2, 2) but does not initialize the values. These values will be random data that previously existed in memory. This is faster than** `np.zeros()` **or** `np.ones()`**.**	
Syntax `np.arange()`	
[4]:	**x = np.arange(0, 10, 2)** **print(x)**
	Output:

	`[0 2 4 6 8]`
Explanation: `np.arange()` **make array with a sequence of values ranging from 0 to less than 10 with an interval of 2. This is useful for generating a range of values with a certain interval.**	
Syntax `np.linspace()`	
`[5]:`	**`x = np.linspace(0, 1, 5)`** **`print(x)`**
	Output: `[0. 0.25 0.5 0.75 1. ]`
Explanation: `np.linspace()` **make array with 5 values evenly distributed from 0 to 1. This is useful for creating data with equal intervals within a specified range.**	
Syntax `np.full()`	
`[6]:`	**`x = np.full((2, 2), 7)`** **`print(x)`**

	Output:
	`[[7 7]`
	` [7 7]]`

Syntax `np.random.random()`	
`[7]:`	**`x = np.random.random((2, 3))`** **`print(x)`**

	Output
	`[[0.123 0.456 0.789]`
	` [0.012 0.345 0.678]]`

Explanation: `np.random.random()` **make array with random values uniformly distributed between 0 and 1. This is useful for creating random data for simulation or testing.**	
Syntax `np.random.randint()`	
`[8]:`	**`x = np.random.randint(0, 10, size=(3, 4))`** **`print(x)`**

	Output:
	`[[2 5 7 1]`
	` [8 9 0 3]`
	` [4 2 6 1]]`

Explanation: `np.random.randint()` **make array with random values in the specified range (0 to 10) of the form (3, 4). It is often used to generate random data in a range integer certain.**	
Syntax `np.fromfunction()`	
`[9]:`	**`def addition(number1, number2):`** **`  return number1 + number2`** **`x = np.fromfunction(addition, (3, 3), dtype=int)`** **`print(x)`**

	Output:
	`[[0 1 2]`
	` [1 2 3]`

	[2 3 4]]
Explanation: `np.fromfunction()` **reates an array of the form (3, 3) based on the function applied to the array index. Here, the sum function is used to fill the values in the array. So the index [1][2] will be filled with the value 3. This is useful for producing arrays based on certain calculations or formulas.**	

2.6. Addition of Array Elements

Adding elements in a NumPy array is an important process that is often used in various applications, especially in data processing and analysis in artificial intelligence. Although NumPy arrays have a fixed size, there are several methods that allow you to add new elements by creating a new array that includes the additional elements. Here is an implementation and example for adding elements to an array in NumPy:

Syntax `np.append()`	
[1]:	`# append to 1D array` **`a = np.array([1, 2, 3])`** **`a = np.append(a, [4, 5])`** **`print("1D array after append:\n", a)`** **`# append pada array 2D`** **`x = np.array([[1, 2], [3, 4]])`** **`x = np.append(x, [[5, 6]], axis=0)`** **`print("2D array after append:\n", x)`**
	Output: 1D array after append: [1 2 3 4 5] 2D array after append: [[1 2] [3 4] [5 6]]
Syntax `np.concatenate()`	

[2]:	`# concatenate on 1D array`
	`a = np.array([1, 2])`
	`b = np.array([3, 4])`
	
	`c = np.concatenate((a, b))`
	
	`print("1D array after concatenate:\n", c)`
	
	`# concatenate pada array 2D`
	`x = np.array([[1, 2], [3, 4]])`
	`y = np.array([[5, 6]])`
	
	`z = np.concatenate((x, y), axis=0)`

```
print("2D array after concatenate:\n", z)
```

```
Output:
1D array after concatenate:
[1 2 3 4]
2D array after concatenate:
[[1 2]
 [3 4]
 [5 6]]
```

Syntax `np.insert()`

[3]:
```
# insert into 1D array
a = np.array([1, 2, 3])

a = np.insert(a, 1, 4)  # Insert '4' in index 1

print("1D array after insert:\n", a)

# insert into 2D arrayinsert into 2D array
x = np.array([[1, 2], [3, 4]])

x = np.insert(x, 0, [5, 6], axis=0) # Insert line [5, 6] index 0

print("2D array after insert:\n", x)
```

```
Output:
1D array after insert:
[1 4 2 3]
2D array after insert:
[[5 6]
 [1 2]
 [3 4]]
```

Syntax `np.stack()`

[4]:
```
# stack on 1D array
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

c = np.stack((a, b))

print("1D array after stack:\n", c)
print("Shape of array c:", c.shape)


# stack on 2D array
x = np.array([[1, 2], [3, 4]])
y = np.array([[5, 6], [7, 8]])

z = np.stack((x, y))

print("2D array after stack:\n", z)
print("Shape from array z:", z.shape)
```

```
Output:
1D array after stack:
[[1 2 3]
 [4 5 6]]
Shape of array c: (2, 3)
2D array after stack:
[[[1 2]
```

	[3 4]] [[5 6] [7 8]]] Shape of array z: (2, 2, 2)

2.7. Element Removal Array

Removal of elements from array NumPy is an important step in data cleaning and manipulation. Element deletion allows you to delete elements based on certain indices, conditions, or criteria. The use of this method helps in managing irrelevant or unwanted data, as well as preparing data for further analysis or modeling. The following is an example and implementation of deleting elements on array NumPy:

Syntax `np.delete()` **to delete the element on array 1D**	
[1]:	**x = np.array([1, 2, 3, 4, 5])** # Delete the element at index 2 **x = np.delete(x, 2)** **print(x)**
	Output: [1 2 4 5]
Syntax `np.delete()` **to delete the row on array 2D**	
[2]:	**x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])** # Delete the row at index 1 (second row) **x = np.delete(x, 1, axis=0)** **print(x)**
	Output: [[1 2 3] [7 8 9]]
Syntax `np.delete()` **to delete the column on array 2D**	
[3]:	**x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])** # Delete the column at index 1 (second column) **x = np.delete(x, 1, axis=1)** **print(x)**
	Output: [[1 3] [4 6] [7 9]]
Syntax `np.where()`	
[4]:	**x = np.array([1, 2, 3, 4, 5])** # Replaces elements greater than 3 with NaN **x = np.where(x > 3, np.nan, x)** # Removes NaN elements from the array **x = x[~np.isnan(x)]**

	print(x)
	Output: [1. 2. 3.]

2.8. Element Ordering Array

Sequencing array NumPy is an important operation in data analysis and artificial intelligence modeling. By sorting elements in a specific order, you can make data analysis and processing easier. NumPy provides functions like `np.sort()`, `array.sort()`, and `np.argsort()` to sort data directly, in-place, or get an element order index. The following is an implementation and example of sorting elements in array NumPy:

Syntax `np.sort()`	
[1]:	**x = np.array([3, 1, 2])** **y = np.sort(x)** **print("Sorted array: ", y)**
	Output: Sorted array: [1 2 3]
Syntax `array.sort()`	
[2]:	**x = np.array([3, 1, 2])** **x.sort()** # array elements are sorted directly within the original **print("Sorted array: ", x)**
	Output: Sorted array: [1 2 3]
Syntax `np.sort()` **for Array Multidimensional**	
[3]:	**x = np.array([[3, 1, 2], [9, 7, 8]])** **y = np.sort(x, axis=0)** # Sort along a column **print("Array that has been sorted along columns: \n", y)** **z = np.sort(x)** # Sorts along rows **print("Array that has been sorted along the lines: \n", z)**
	Output: Array that has been sorted along columns: [[3 1 2] [9 7 8]] Array that has been sorted along the lines: [[1 2 3] [7 8 9]]

2.9. Indexing and Slicing Array

Indexing And slicing is an important technique in data processing using NumPy. Indexing allows access to specific elements within array based on index, whereas slicing allows

taking a subset of array using index ranges. Both make it easier to manipulate data, filter and access relevant information, especially useful in data analysis and artificial intelligence modeling. Here is an implementation and example indexing And slicing on array NumPy:

Simple Indexing	
[1]:	**x = np.array([10, 20, 30, 40, 50])** **print(x[2])**
	Output: 30
Explanation: **Indexing simple is used to access a single element from array based on the index. In this example, `x[2]` accessing the 3rd element of array `x`, which is the value 30**	
Slicing	
[2]:	**x = np.array([10, 20, 30, 40, 50])** # Slicing to get elements from index 1 to 3 **subset = x[1:4]** **print(subset)**
	Output: [20 30 40]
Explanation: **Slicing used to take a subset of array with index range. In this example, `x[1:4]` accesses elements starting from index 1 to 3 (end index not included), yields array `[20, 30, 40]`**	
Boolean *Indexing*	
[3]:	**x = np.array([10, 20, 30, 40, 50])** # Boolean mask to select elements greater than 25 **mask = x > 25** **print("Boolean result of masking:", mask)** **subset = x[mask]** **print("Result elemen:", subset)**
	Output: Boolean result of masking: [False False True True True] Result elemen: [30 40 50]
Explanation: **Explanation: Boolean indexing uses array boolean (mask) to select elements that meet certain conditions. Here, `x > 25` generate masks `[False, False, True, True, True]`, which is then used to select elements greater than 25.**	
Fancy Indexing	
[4]:	**x = np.array([10, 20, 30, 40, 50])** **# Array of index** **index = [1, 3]** **subset = x[index]** **print(subset)**
	Output:

	`[20 40]`
Explanation: **Fancy indexing accessing array elements using array index. In this example, `x[index]` selects elements at indices 1 and 3 from array `x`, namely 20 and 40.**	
Multi-Dimensional Indexing	
`[5]:`	**`x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])`** `# Access elements in the 1st row and 2nd column` **`element = x[1, 2]`** **`print(element)`**
	`Output:` `6`
Explanation: **Multi-dimensional indexing used to access inner elements array multidimensional. Here, `x[1, 2]` access the elements that are in the 1st row and 2nd column of array 2D, namely 6.**	
Ellipsis **`...`**	
`[6]:`	**`x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])`** `# Access all elements in the 1st row` **`subset = x[1, ...]`** **`print(subset)`**
	`Output:` `[4, 5, 6]`
Explanation: **`x[1, ...]` access all elements in the second row of array 2D x, yield array `[4, 5, 6]`. Usage Ellipsis (`...`) is here equivalent to `x[1, :]` and doesn't give a difference because array has only two dimensions.**	
Syntax`np.ix_()`	
`[7]:`	**`x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])`** `# Row and column indices` **`rows = [0, 2]`** **`cols = [1, 2]`** **`subset = arr[np.ix_(rows, cols)]`** **`print(subset)`**

	`Output`
	`[[2 3]`
	`[8 9]]`

Explanation: `np.random.random()` **make array with randomly distributed values uniform between 0 and 1. This is useful for generating random data for simulation or testing**	

2.10. Copying (Copy) Array

Copying a NumPy array allows data manipulation without affecting the original array.There are two types of copying: shallow copy And deep copy. Shallow copy (like using`view()` or slicing) creates a copy that shares data with array original, so changesto

one will affect the others. On the contrary, deep copy (use `copy()`) creates anindependent copy, so changes to the copy have no effect array original. Choose a typecopy The right one is important for managing data effectively in a variety of operations.Here is an implementation and example of copying array on Numpy:

Syntax `view()`

[1]:
```
x = np.array([1, 2, 3, 4, 5])

# Create a shallow copy using view()
y = x.view()

# Changes the first element of the shallow copy
y[0] = 99

print("Original array:", x)
print("Shallow copy:", y)
```

```
Output:
Original array: [99  2  3  4  5]
Shallow copy: [99  2  3  4  5]
```

Explanation:
In this example, array `and` is a shallow copy of array `x`, made using `view()`. When the first element is in array `and` changed, the changes are also reflected in array `x`, because they both share the same data

Syntax `copy()`

[2]:
```
x = np.array([1, 2, 3, 4, 5])

# Create a deep copy using copy()
y = x.copy()

# Changes the first element of the deep copy
y[0] = 99

print("Original array:", x)
print("Deep copy:", y)
```

```
Output:
Original array: [1 2 3 4 5]
Deep copy: [99  2  3  4  5]
```

Explanation:
Here, array `and` is a deep copy of array `x`, made using `copy()`. When the first element is in array `and` changed, array original `x` unaffected, because deep copy creates independent copies that do not share data with array original.

2.11. Basic Operations Array

Basic operations on array NumPy includes calculations of the maximum, minimum,total, and average of elements array. These functions, such as `max()`, `min()`,`sum()`, And `mean()`, often used in data analysis and artificial intelligencemodeling to evaluate and understand the main characteristics of dataset.

With these basicoperations, you can quickly gain insight into the distribution and scale of the data beingprocessed, making subsequent data processing steps easier. Following is animplementation and example of basic operations on array NumPy:

Syntax `np.max()`	

[1]:	`x  = np.array([1, 5, 3, 8, 2])`
	
	`max_value = np.max(x)`
	
	`print("Maximum value:", max_value)`

	Output: Maximum value: 8
Syntax `np.min()`	

[2]:	`x  = np.array([1, 5, 3, 8, 2])`
	
	`min_value = np.min(x)`
	
	`print("Minimum value:", min_value)`

	Output: Minimum value: 1
Syntax `np. sum()`	

[3]:	`x  = np.array([1, 5, 3, 8, 2])`
	
	`sum_value = np.sum(x)`
	
	`print("Sum value:", sum_value)`

	Output: Sum value: 19
Syntax `np. mean()`	

[4]:	`x  = np.array([1, 5, 3, 8, 2])`
	
	`mean_value = np.mean(x)`
	
	`print("Average:", mean_value)`

	Output: Average: 3.8

2.12. Transpose and Reshape Array

Reshape And transpose are two important operations in manipulation array. Reshapechange shape or dimensions array without changing the data, allowing customization ofthe data for specific operations or machine learning models. Transpose changedimensions array, such as converting rows to columns, and are often used in linearoperations or when switching data orientation. Both provide flexibility in managing datastructures. Here is an implementation and example transpose And reshape on arrayNumPy:

Syntax `transpose()`	
[1]:	**x = np.array([[1, 2, 3], [4, 5, 6]])** **y = x.transpose()** **print(y)**
	Output: [[1 4] [2 5] [3 6]]
Syntax `np.transpose()`	
[2]:	**x = np.array([[1, 2, 3], [4, 5, 6]])** **y = np.transpose(x)** **print(y)**
	Output: [[1 4] [2 5] [3 6]]
Syntax `.T` **for transpose**	
[3]:	**x = np.array([[1, 2, 3], [4, 5, 6]])** **y = x.T** **print(y)**
	Output: [[1 4] [2 5] [3 6]]
Syntax `reshape()`	
[4]:	**x = np.array([1, 2, 3, 4, 5, 6])** # Converts to a 2D array with 2 rows and 3 columns **y = x.reshape((2, 3))** **print(y)**
	Output: [[1 2 3] [4 5 6]]
Syntax `np.reshape()`	
[5]:	**x = np.array([1, 2, 3, 4, 5, 6])** # Converts to a 2D array with 2 rows and 3 columns **y = np.reshape(x, (2, 3))** **print(y)**
	Output: [[1 2 3]

```
[4 5 6]]
```

2.13. Numpy Programming Practice Questions

1. Create a Python program using library Numpy that can store data array of students (Name, NIM, Grade, Year of Entry). Type name and NIM string, The value is of type number real, while EntryYear is of type integer. Your program must be able to accept student data input interactively (from keyboard), displays all data that has been input, as well as displays the highest value, lowest value and average value. Your program must also be able to search for students based on NIM/Name, then display complete student data on the results of the search..

2. Create a Python program using library NumPy which can store inventory data for goods in a warehouse. Data that needs to be stored includes Item Name (string), Item Code (string), Amount (integer), and PricePerUnit (number real). Your program must be able to accept item data input interactively (from keyboard), displays all data that has been entered, and displays the total inventory value (Quantity x Price Per Unit for each item). Apart from that, your program must also be able to search for items based on Item Code or Item Name, and display complete data on the items resulting from the search

Modul 3 INTRODUCTION *LIBRARY* NETWORKX

Practical Objectives:
1. Students understand how to manage graph data structures *library* NetworkX. 2. Students know the main features available in *library* NetworkX.

MAIN REFERENCE: https://networkx.org/documentation/stable/tutorial.html
ADDITIONAL REFERENCES:
https://networkx.org/documentation/stable/reference/index.html

NetworkX is *library* Python is used to create, manipulate, and study the structure and function of graphs. NetworkX supports operations on undirected graphs, directed graphs, weighted graphs, etc.

3.1. Method *Import* NetworkX

NetworkX should already be present in your Anaconda installation. To use the NetworkX module, it must be done *import* first. Here is the standard way to import the NetworkX library:

```
import network as nx
```


3.2. Creating an Empty Graph on NetworkX

On NetworkX, there are two basic graph types that will be studied. The first is `**Graph**`, which produces an undirected graph (*undirected graph*), where each *edge* has no direction and is symmetrical. If there is *edge* between two *node*, then second *node* They are connected to each other without regard to direction. Second is `**DiGraph**` which produces a directed graph, so the relationship between two node doesn't have to be symmetrical. This means that if there is *edge* from *node* And when *node* B, indirectly there must be *edge* from *node* B is *node* A. The following is how an empty graph is defined in NetworkX:

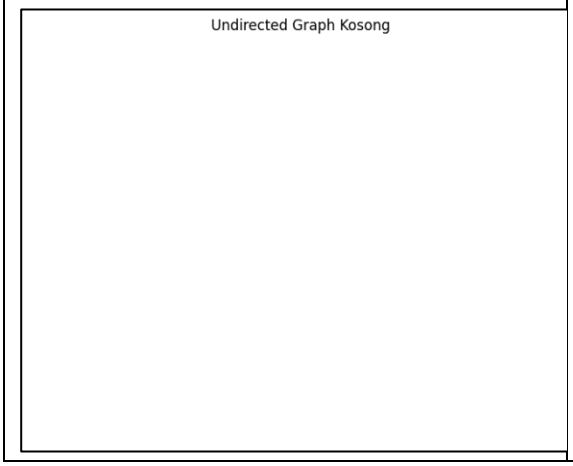
```
# Import the library used
import matplotlib.pyplot as plt
import networkx as nx

# Node positions used in this module
pos = {'A': [ 0.93888004, -0.0449161 ],
 'B': [-0.73226196, -0.53345128],
 'C': [0.29334637, 1.        ],
 'D': [-0.79991428,  0.56134392],
 'E': [ 0.29994983, -0.98297654]}
```

```
# Support function for printing graphs
def show_graph(G, pos=None, title='') :
  if pos == None :
    pos = nx.spring_layout(G)

  nx.draw(
      G,
      pos,
      with_labels=True,
      node_color='red',
      node_size=2000,
      font_color="white",
      font_weight="bold",
      width=5
  )

  edge_labels = nx.get_edge_attributes(G, 'weight')
  nx.draw_networkx_edge_labels(
      G,
      pos,
      edge_labels=edge_labels,
      font_color='blue',
      font_weight="bold",
      font_size=12,
  )

  plt.margins(0.2)
  plt.title(title)
  plt.show()
```

Definition `Graph` empty

```
G_undirected = nx.Graph()

# Print the graph
show_graph(G_undirected,
title="Undirected Graph Kosong")
```

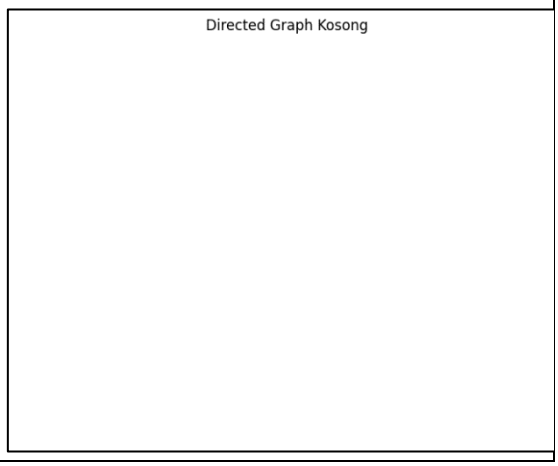
Output:



Definition ``DiGraph'' empty

```
G_directed = nx.DiGraph()

# Print the graph
show_graph(G_directed,
title="Directed Graph Kosong")
```

Output:



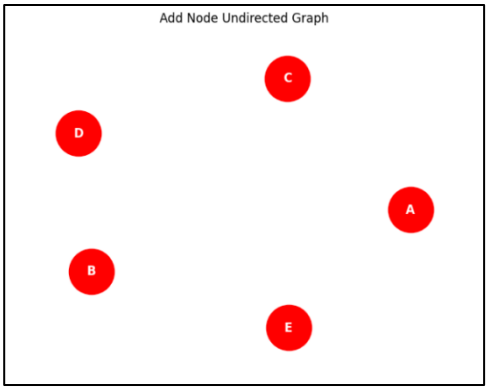
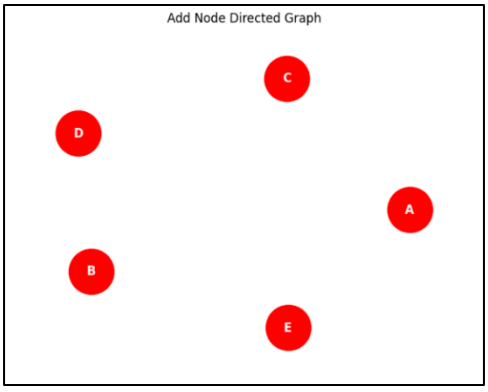
From the results above, it can be seen that the empty graph definition was successful because an empty image was printed with a title.

3.3. Add *Node* On the graph

There are two ways to add *node* on NetworkX, namely with `add_node()` to add one *node* and with `add_nodes_from()` to add some *node* at a time. Position *node* when printed it may vary each time the code is run. Here is the implementation to add *node* on the graph:

Add *node* on *undirected graph*	**Add *node* on *directed graph***

`# Cara add_node()`
`G_undirected.add_node('A')`
`G_undirected.add_node('B')`
`G_undirected.add_node('C')`
`G_undirected.add_node('D')`
`G_undirected.add_node('E')`
 	
`# Cara add_nodes_from()`
`node_data = ['A', 'B', 'C', 'D', 'E']`
`G_undirected.add_nodes_from(node_data)`
 	
`# Record node positions to ensure consistency`
`pos = nx.spring_layout(G_undirected)`
 	
`# Print the graph`
`show_graf(G_undirected, pos=pos, title="Add Node Undirected Graph")`
 	
`# View the number of nodes in the graph`
`print("Jumlah node:", G_undirected.number_of_nodes())`
	`# Cara add_node()`
	`G_directed.add_node('A')`
	`G_directed.add_node('B')`
	`G_directed.add_node('C')`
	`G_directed.add_node('D')`
	`G_directed.add_node('E')`
	
	`# Cara add_nodes_from()`
	`node_data = ['A', 'B', 'C', 'D', 'E']`
	`G_directed.add_nodes_from(node_data)`
	
	`# Records the position of the order node consistent`
	`pos = nx.spring_layout(G_directed)`
	
	`# Print the graph`
	`show_graf(G_directed, pos=pos, title="Add Node Directed Graph")`
	
	`# View the number of nodes in the graph`
	`print("Jumlah node:", G_directed.number_of_nodes())`

`Output:`
 
 	
`Number of nodes: 5`
	`Output:`
	 
	
	`Number of nodes: 5`

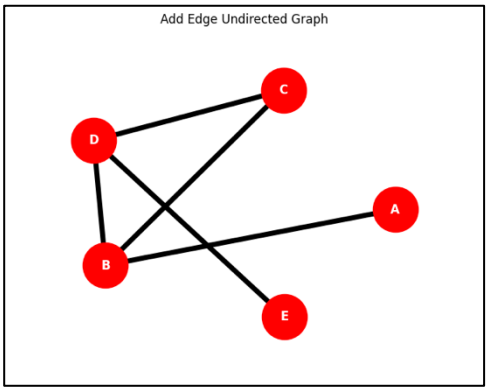
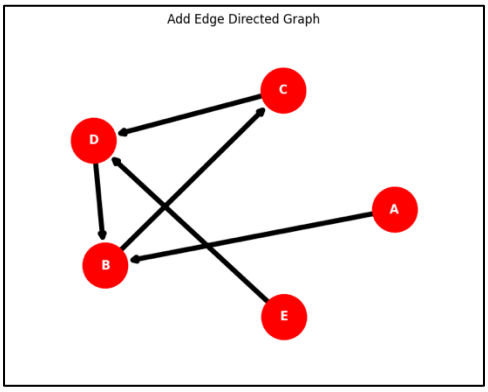
From the two results above, it can be seen that the two graphs still look the same. This is because of differences *undirected graph* And *directed graph* located on *edge*.

3.4. Add *Edge* On the Graph

To add *edge* in NetworkX, method `add_edge()` used to add one *edge*, And `add_edges_from()` used to add some *edge* at a time. *Edge* between two *node* can be added by specifying *node* beginning and end. If *node* mentioned is not yet in the graph, then *node* will be added automatically. On *directed graph*, *edge* has direction from *node* first to *node* second. Here is the implementation to add *edge* on the graph at NetworkX:

Add *edge* on *undirected graph*	**Add *edge* on *directed graph***

`# Cara add_edge()`
`G_undirected.add_edge('A', 'B')`
`G_undirected.add_edge('B', 'C')`
`G_undirected.add_edge('C', 'D')`
`G_undirected.add_edge('D', 'B')`
`G_undirected.add_edge('E', 'D')`
 	
`# Cara add_edges_from()`
`edge_data = [('A', 'B'), ('B', 'C'),`
`('C', 'D'), ('D', 'B'), ('E', 'D')]`
`G_undirected.add_edges_from(edge_data)`
 	
`# Print the graph`
`show_graf(G_undirected, pos=pos,`
`title="Add Edge Undirected Graph")`
 	
`# View the number of edges in the graph`
`print("Jumlah edge:",`
`G_undirected.number_of_edges())`
	`# Cara add_edge()`
	`G_directed.add_edge('A', 'B')`
	`G_directed.add_edge('B', 'C')`
	`G_directed.add_edge('C', 'D')`
	`G_directed.add_edge('D', 'B')`
	`G_directed.add_edge('E', 'D')`
	
	`# Cara add_edges_from()`
	`edge_data = [('A', 'B'), ('B', 'C'),`
	`('C', 'D'), ('D', 'B'), ('E', 'D')]`
	`G_directed.add_edges_from(edge_data)`
	
	`# Print the graph`
	`show_graf(G_directed, pos=pos,`
	`title="Add Edge Directed Graph")`
	
	`# View the number of edges in the graph`
	`print("Jumlah edge:",`
	`G_directed.number_of_edges())`

Output:  Number of edges: 5	Output:  Number of edges: 5

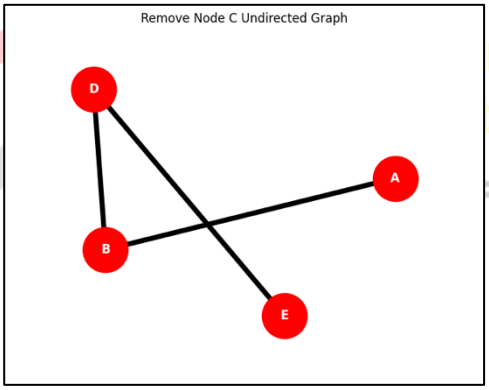
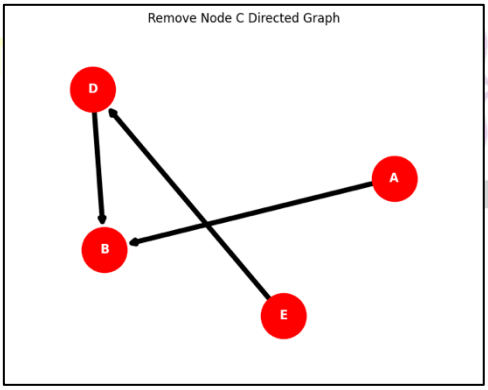
Difference between *undirected graph* And *directed graph* visible after addition *edge* on the graph. Order of addition *edge* on *directed graph* not the same, where `add_edge('A', 'B')` different from `add_edge('B', 'A')`.

3.5. Delete *Node* And *Edge*

On NetworkX, *node* And *edge* can be removed from the graph using the method `remove_node()` And `remove_edge()`. On `remove_node()`, *node* selected will be removed from the graph, Along with all *edge* which is connected on *node* the. Meanwhile on `remove_edge()`, *edge* selected will be removed from the graph, but *node* which is connected by *edge* it will still be there. Following is the deletion implementation *node* And *edge* on the graph at NetworkX:

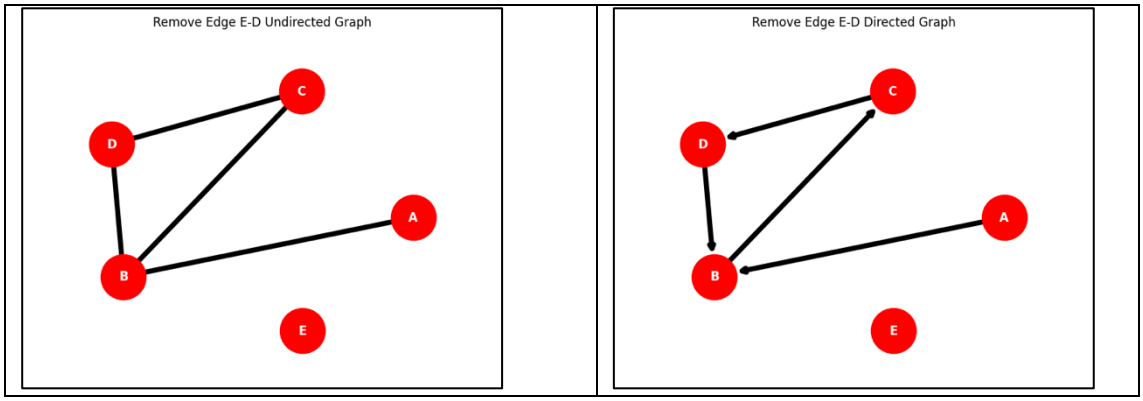
<table>
<tr><td colspan="2">Elimination node on the graph</td></tr>
<tr><td>Elimination node on undirected graph</td><td>Elimination node on directed graph</td></tr>
<tr><td><pre># Deletion of node 'C'
G_undirected_remove.remove_node('C')

Print the graph
show_graph(G_undirected_remove,
pos=pos, title='Remove Node C
Undirected Graph')</pre></td><td><pre># Deletion of node 'C'
G_directed_remove.remove_node('C')

Print the graph
show_graph(G_directed_remove,
pos=pos, title='Remove Node C
Directed Graph')</pre></td></tr>
<tr><td>Output:
 
</td><td>Output:
 
</td></tr>
<tr><td colspan="2">Elimination edge on the graph</td></tr>
<tr><td>Elimination edge on undirected graph</td><td>Elimination edge pada directed graph</td></tr>
<tr><td><pre># Delete edges E - D
G_undirected_remove.remove_edge('E',
'D')

Print the graph
show_graph(G_undirected_remove,
pos=pos, title='Remove Edge E-D
Undirected Graph')</pre></td><td><pre># Delete edges E - D
G_directed_remove.remove_edge('E',
'D')

Print the graph
show_graph(G_directed_remove,
pos=pos, title='Remove Edge E-D
Directed Graph')</pre></td></tr>
<tr><td>Output:</td><td>Output:</td></tr>
</table>



3.6. Elements On A Graph

The elements in the graph in NetworkX are the main components that form the structure of the graph and make it possible to analyze the graph created. The following is an explanation of the various graph elements and the methods used to access them:

`.nodes()` *Method this is used to return a list of all node in the graph*
print("Node pada G_undirected:", G_undirected.nodes()) **print("Node pada G_directed:", G_directed.nodes())**
Output: Node pada G_undirected: ['A', 'B', 'C', 'D', 'E'] Node pada G_directed: ['A', 'B', 'C', 'D', 'E']
`.edges()` *Method this is used to return a list of all edges in the graph*
print("Edge pada G_undirected:", G_undirected.edges()) **print("Edge pada G_directed:", G_directed.edges())**
Output: Edge pada G_undirected: [('A', 'B'), ('B', 'C'), ('B', 'D'), ('C', 'D'), ('D', 'E')] Edge pada G_directed: [('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'B'), ('E', 'D')]
`.adj()` *Method* which is used to access the neighbor list or *adjacency* from each *node*. *Method* It provides direct access to the nodes connected to a particular node.

`.adj()` **on *undirected graph***	`.adj()` **on *directed graph***
print("Tetangga G_undirected") **for node in G_undirected.nodes():** **# Neighbors of a given node** **adj = list(G_undirected.adj[node])** **print(f"Tetangga {node}: {adj}")**	**print("Tetangga G_directed")** **for node in G_directed.nodes():** **# Neighbors of a given node** **adj = list(G_directed.adj[node])** **print(f"Tetangga {node}: {adj}")**
Output: G_undirected Neighbor Neighbor A: ['B'] Neighbor B: ['A', 'C', 'D'] Neighbor C: ['B', 'D'] Neighbor D: ['C', 'B', 'E']	Output: G_undirected Neighbor Neighbor A: ['B'] Neighbor B: ['C'] Neighbor C: ['D'] Neighbor D: ['B']

Neighbor E: ['D']	Neighbor E: ['D']
`.degree()` *Method* it is used to return the degree of each node, i.e. the total number *edge* connected to that node. On *undirected graph*, this is the total number of connections, while on *directed graph* this is the amount *incoming* And *outgoing edges*.	
`.adj()` **on *undirected graph***	`.adj()` **on *directed graph***
print("Derajat node G_undirected") **for node, degree in G_undirected.degree():** **print(f"Derajat {node}: {degree}")**	**print("Derajat node G_directed")** **for node, degree in G_directed.degree():**
Output: G_undirected Neighbor Neighbor A: ['B'] Neighbor B: ['A', 'C', 'D'] Neighbor C: ['B', 'D'] Neighbor D: ['C', 'B', 'E'] Neighbor E: ['D']	Output: G_directed neighbor Neighbor A: ['B'] Neighbor B: ['C'] Neighbor C: ['D'] Neighbor D: ['B'] Neighbor E: ['D']
`.in_degree()` *Method* This can only be used on *directed graph* to analyze *in-degree* of each node, namely the number *edge* that enters that node.	
print("In-degree node G_directed") **for node, degree in G_directed.in_degree():** **print(f"In-degree {node}: {degree}")**	
Output: In-degree node G_directed In-degree A: 0 In-degree B: 3 In-degree C: 2 In-degree D: 3 In-degree E: 1	
`.out_degree()` *Method* This can only be used on *directed graph* to analyze out-*degree* from each node, namely the amount *edge* that comes out of that node.	
print("Out-degree node G_directed") **for node, degree in G_directed.out_degree():** **print(f"Out-degree {node}: {degree}")**	
Output: Out-degree node G_directed Out-degree A: 1 Out-degree B: 1 Out-degree C: 1 Out-degree D: 1 Out-degree E: 1	

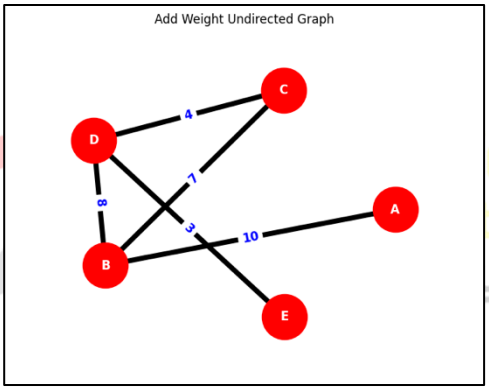
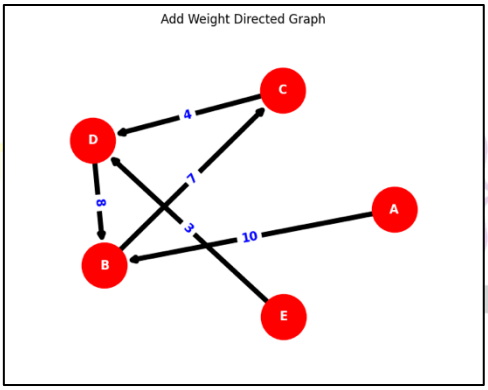
3.7. Add *Weight* On *Graph*

Weight or weights can be added on *edge* in a graph to represent the value of the relationship between two *node*. There are two main methods for adding *weight* on *edge*. First by using `.add_edge()`, which is used to add one *edge* on the graph and can be added as attributes using parameters`weight`. Second, to add `weight` can using method `.add_weighted_edges_from()` which is used to add some

edge at the same time with *weight* which has been specified. Here is the implementation and example implementation for adding *weight* on *graph*:

Add *weight* on *undirected graph*	**Add *weight* on *directed graph***

`# Cara add_edge()`
`G_undirected.add_edge('A', 'B', weight=10)`
`G_undirected.add_edge('B', 'C', weight=7)`
`G_undirected.add_edge('C', 'D', weight=4)`
`G_undirected.add_edge('D', 'B', weight=8)`
`G_undirected.add_edge('E', 'D', weight=3)`
 	
`# Cara add_weighted_edges_from()`
`edge_weight_data = [('A', 'B', 10), ('B', 'C', 7), ('C', 'D', 4), ('D', 'B', 8), ('E', 'D', 3)]`
 	
`G_undirected.add_weighted_edges_from( edge_weight_data)`
 	
`# Print the graph`
`show_graf(G_undirected, pos=pos, title="Add Weight Undirected Graph")`
	`# Cara add_edge()`
	`G_directed.add_edge('A', 'B', weight=10)`
	`G_directed.add_edge('B', 'C', weight=7)`
	`G_directed.add_edge('C', 'D', weight=4)`
	`G_directed.add_edge('D', 'B', weight=8)`
	`G_directed.add_edge('E', 'D', weight=3)`
	
	`# Cara add_weighted_edges_from()`
	`edge_weight_data = [('A', 'B', 10), ('B', 'C', 7), ('C', 'D', 4), ('D', 'B', 8), ('E', 'D', 3)]`
	
	`G_directed.add_weighted_edges_from( edge_weight_data)`
	
	`# Print the graph`
	`show_graf(G_undirected, pos=pos, title="Add Weight Directed Graph")`

Output: 	Output: 

3.8. Graph Analysis

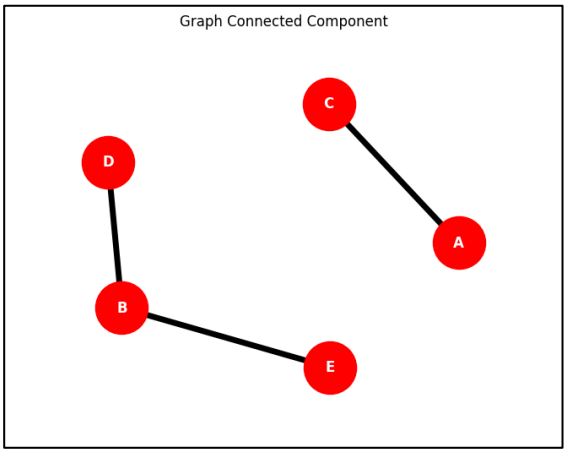
In NetworkX, various functions for analyzing graphs are provided to help understand the structure and characteristics of the created graph. The following is an explanation and example implementation of each function:

`.connected_components()` This function is used to find all *connected components* in *undirected graph* which means every *node* within a group can be reached by *node* others in the same group.

`import networkx as nx`
 	
`G = nx.Graph() # Create a new graph`
`# Add edges (as well as nodes) to the graph`
`G.add_edges_from([('B', 'D'), ('B', 'E'), ('C', 'A')])`
 	
`# Create a group of connected components`
`components = list(nx.connected_components(G))`
`print(components)`

```
# Print the graph
show_graph(G, title="Graph Connected Component")
```

```
Output:
[{'D', 'E', 'B'}, {'A', 'C'}]
```

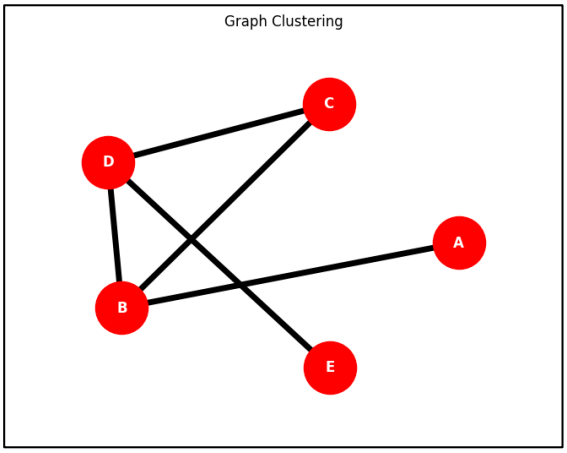


`.clustering()`

This function measures *clustering coefficient* from each *node* in the graph. *Clustering coefficient* is a local measure that describes how inclined *node-node* neighbors of a *node* each other connected to each other.

```
import networkx as nx

G = nx.Graph() # Create a new graph
# Add edges (as well as nodes) to the graph
edge_data = [('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'B'), ('E', 'D')]
G.add_edges_from(edge_data)

# Calculate the clustering coefficient for each node
clustering_coefficients = nx.clustering(G)
print(clustering_coefficients)

# Print the graph
show_graph(G, title="Graph Clustering")
```

```
Output:
{'A': 0, 'B': 0.333, 'C': 1.0, 'D': 0.333, 'E': 0}
```

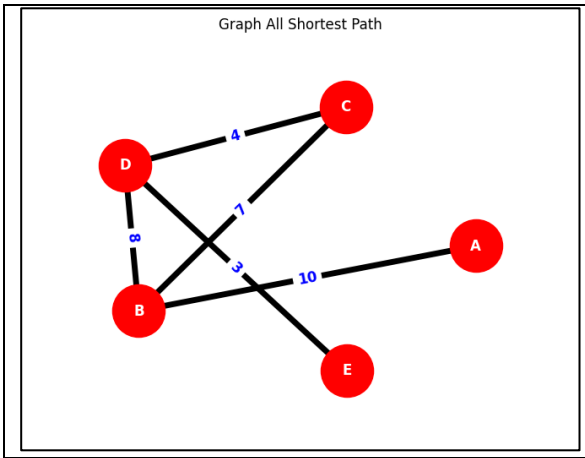


`.all_shortest_path()`

This function is used to find all the shortest paths between two *node* in the graph. This function searches for all routes that have a value *edge* smallest of the two *node* certain.

```
import networkx as nx

G = nx.Graph() # Create a new graph
# Add edges and weights (as well as nodes) to the graph
edge_weight_data = [('A', 'B', 10), ('B', 'C', 7), ('C', 'D', 4), ('D', 'B', 8), ('E', 'D', 3)]
G.add_weighted_edges_from(edge_weight_data)

# Create a list of shortest paths between two nodes for all nodes
shortest_paths = dict(nx.all_pairs_shortest_path(G))
print(shortest_paths)

# Print the graph
show_graph(G, title="Graph All shortest path")
```

```
Output:
{
  'A': {
    'A': ['A'],
    'B': ['A', 'B'],
    'C': ['A', 'B', 'C'],
    'D': ['A', 'B', 'D'],
    'E': ['A', 'B', 'D', 'E']
  },
  'B': {
    'B': ['B'],
    'A': ['B', 'A'],
    'C': ['B', 'C'],
    'D': ['B', 'D'],
    'E': ['B', 'D', 'E']
  },
  'C': {
    'C': ['C'],
    'B': ['C', 'B'],
    'D': ['C', 'D'],
    'A': ['C', 'B', 'A'],
    'E': ['C', 'D', 'E']
  },
  'D': {
    'D': ['D'],
    'C': ['D', 'C'],
    'B': ['D', 'B'],
    'E': ['D', 'E'],
    'A': ['D', 'B', 'A']
  },
  'E': {
    'E': ['E'],
    'D': ['E', 'D'],
    'C': ['E', 'D', 'C'],
    'B': ['E', 'D', 'B'],
    'A': ['E', 'D', 'B', 'A']
  }
}
```



3.9. Displaying Graphs with Matplotlib

Matplotlib acts as *library* The main visualization used by NetworkX to plot graphs. With Matplotlib, graph displays can be customized in detail, including color, size, layout, labels, and more, to create more informative and engaging visualizations.

3.9.1. Layout *Node*

Layout *node* in the graph in the plot will be displayed randomly and will change each time the code block for displaying the graph is executed. So that the position is consistent, then position *node* can be stored in a variable which is usually named `pos`.

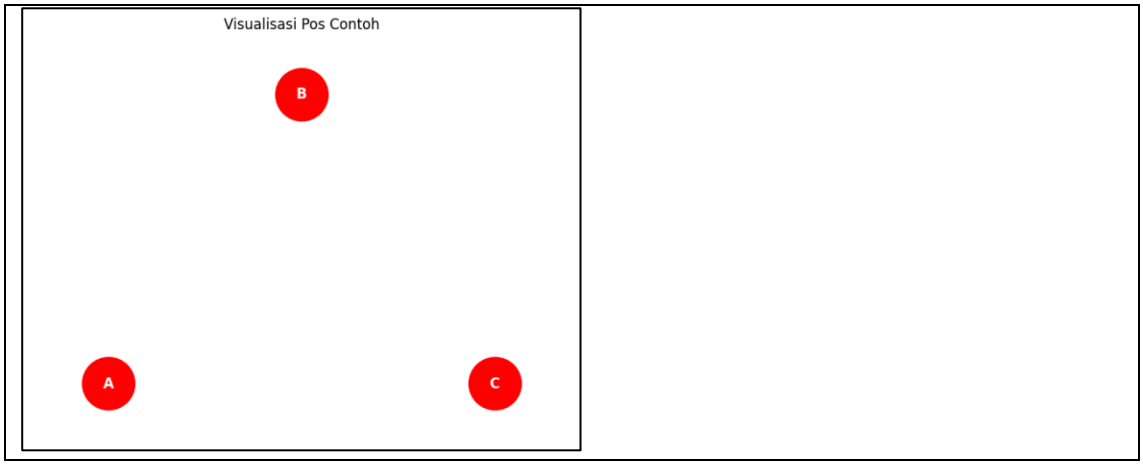
There are several rules so that NetworkX and Matplotlib can read given coordinates. Here are the rules `pos`:

- Data type `pos` must be a *dictionary*.
- `key` in *dictionary* are the names *node* in a graph, for example `A`, `B`, `C`.
- The `value` of the associated key is a tuple or list containing coordinates `(x, y)`, which determines the position *node* on the graph.

The following is an example of defining a variable `pos` for *node* `A`, `B`, `C`:

```
pos = {
    'A': (0,0), # Position for node A
    'B': (1,1), # Position for node B
    'C': (2,0)  # Position for node C
}
```

```
Visualization of `post` Coordinates:
```



3.9.2. Function *Layout* On the graph

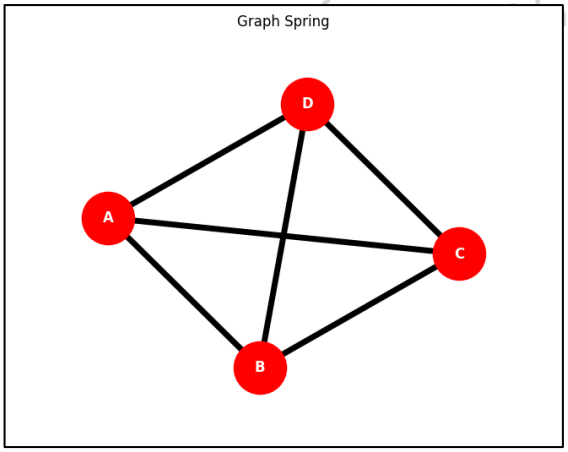
NetworkX provides a variety of functions *layout* automatically to set the layout *node* in 2-dimensional space by considering graph structure, relationships between *node*, and visual readability. With function *layout*, quite complex graphs can be easily arranged, especially when the graph becomes very complex *node*. Following are some of the functions *layout* frequently used in NetworkX:

`spring_layout()`

This function uses a model *spring,* where as if *node* connected by springs. *Node* will be placed tend to be closer to each other, while *node* those that are not connected will be placed further away.

```
pos = nx.spring_layout(G)
```

Contoh spring_layout():



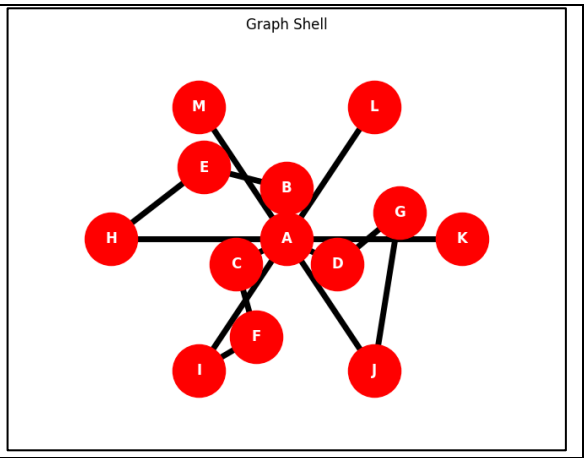
`shell_layout()`

This function puts *node* in several layers, Each lapisa is a circle that surrounds the center, and *node* certain grouped in *shell* different. The `nlist` parameter defines the group *node* which will be placed in a certain layer.

```
pos = nx.shell_layout(
  G_shell,
  nlist=[
    ['A'], # Lapis 1
    ['B', 'C', 'D'], # Lapis 2
    ['E', 'F', 'G'], # Lapis 3
```

Contoh shell_layout():

```
    ['H', 'I', 'J', 'K', 'L', 'M']
  ]
)
```

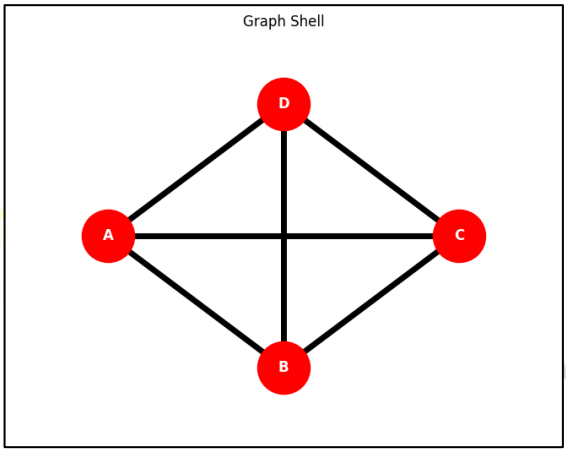


`circular_layout()`

This function puts all *node* in a graph on a circle with the same distance from the center. The position of each *node* in the circle are determined evenly so as to form a symmetrical arrangement.

```
pos = nx.circular_layout(G)
```

Contoh circular_layout():

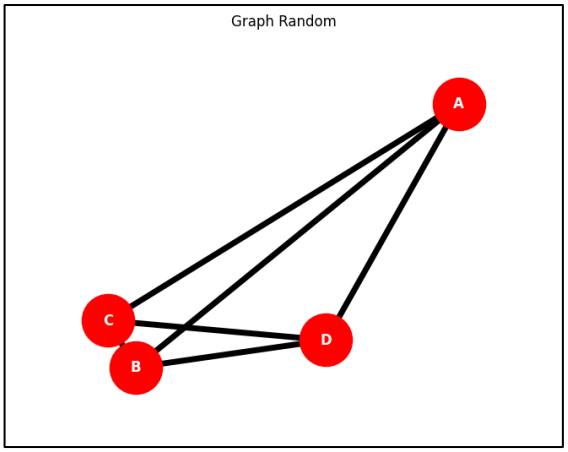


`random_layout()`

This function puts *node* at random. Each node is assigned a randomly generated coordinate position, without regard to relationships or graph structure.

```
pos = nx.random_layout(G)
```

Contoh random_layout():



3.9.3. Function *show_graph* For Graph Visualization

Function `show_graph` is a support function designed to visualize graphs built with NetworkX. This function provides an easy and flexible way to display graphs with various options. All graphs visualized in this practical module are displayed using functions `show_graph` is a support function designed to visualize graphs built with NetworkX. This function provides an easy and flexible way to display graphs with various options. All graphs visualized in this practical module are displayed using functions:

- **Parameter `G`**
  `G` is the graph to be visualized. This graph can be: *undirected graph* (`.Graph()`) or *directed graph* (`.DiGraph()`).
- **Parameter `pos`**
  `post` is *dictionary* which determines the position of each *node* in the graph. If the `pos` parameter is not provided at call time, this function will automatically generate the layout *node* using `.spring_layout()`.
- **Parameter `title`**
  `title` is *string* optional used to add a title to the graph plot. If no title is given, the graph will be displayed without a title.

Here is the function `show_graf` along with an explanation of the parameters:

```
import matplotlib.pyplot as plt
import networkx as nx

# Support function for printing graphs
def show_graph(G, pos=None, title='') :
  # Create a post if a post is not given
  if pos is None:
      pos = nx.spring_layout(G)

  # Function to draw nodes
  nx.draw(
      G,                  # Graf NetworkX
      pos,                # Node position
      with_labels=True,   # Displays the node name
      node_color='red',   # Node color
      node_size=2000,     # Node size
      font_color="white", # Node label font color
      font_weight="bold", # Node label font thickness
      width=5             # Edge line thickness
  )

  # Takes edge labels if there is weight
  edge_labels = nx.get_edge_attributes(G, 'weight')
  # Function to draw nodes
  nx.draw_networkx_edge_labels(
      G,
```

```
        pos,
        edge_labels=edge_labels, # Data weight
        font_color='blue',       # Edge label font color
        font_weight="bold",      # Edge label font color
        font_size=12,            # Edge label font size
    )

    plt.margins(0.2)  # Provides margin to the plot
    plt.title(title)  # Displays the graph title if given
    plt.show()        # Display graphs using matplotlib
```

3.10. Create Graphs Automatically

In NetworkX, graphs can be created automatically using built-in functions that generate different types of graphs based on certain characteristics and structures. NetworkX provides various ways to construct graphs that are well known in graph theory. There are several ways and methods to create graphs automatically.

3.10.1. *Complete Graph*

Complete graph (graf lengkap) is a graph where every pair *node* connected to each other exactly *edge* as shown in Figure 7. Example Figure Complete Graph of a Complete Graph. This means that the graph is complete with `n` *node* will have `n(n-1)2/2` *edge*.

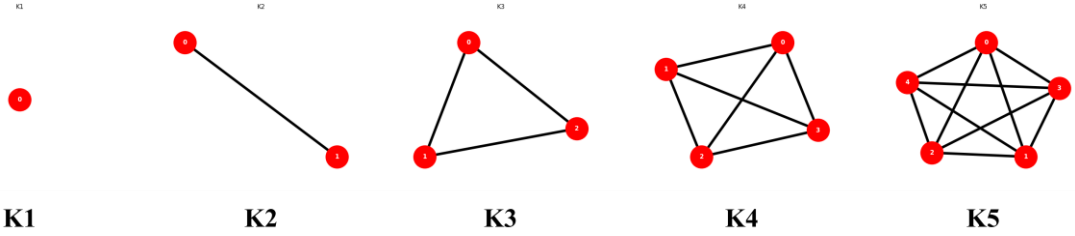
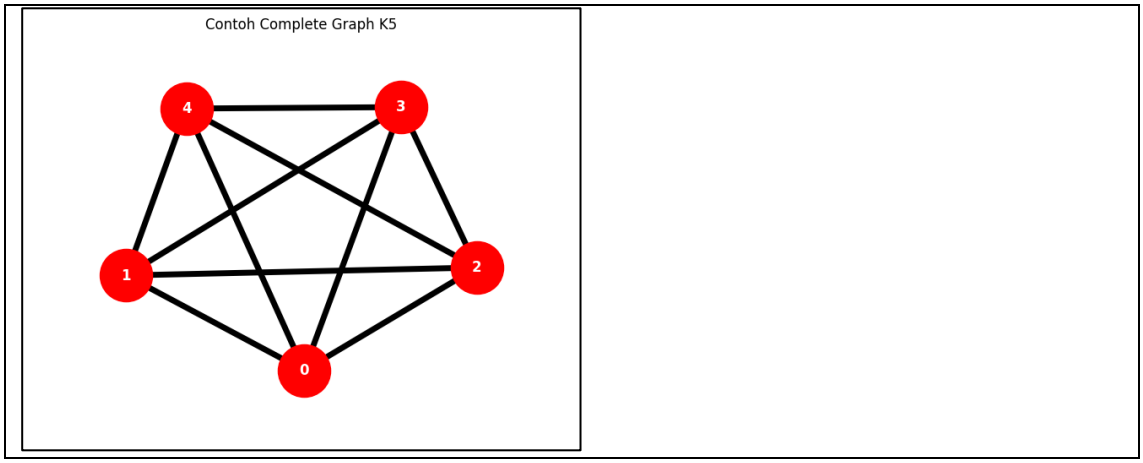


Figure 7. Example Figure *Complete Graph*.

The following is an implementation and creation example *complete graph* on NetworkX:

```
# Create a complete graph with 5 nodes
G = nx.complete_graph(5)

# Print the graph
show_graph(G, title="Contoh Complete Graph K5")
```

Output:



3.10.2. *Complete Bipartite Graph*

Complete bipartite graph (complete bipartite graph) shown in Figure 8 is a graph where *node* divided into two separate sets, and each *node* in a set connected with each *node* in other sets, but there are none *edge* in the same set.

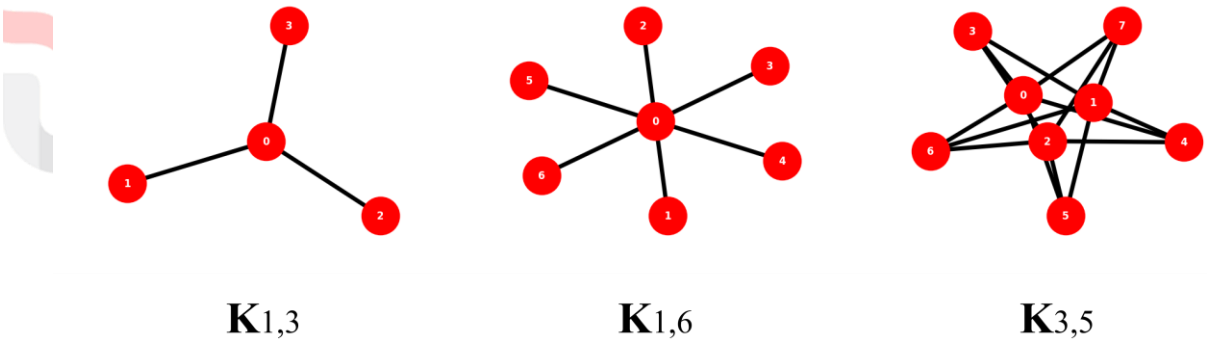
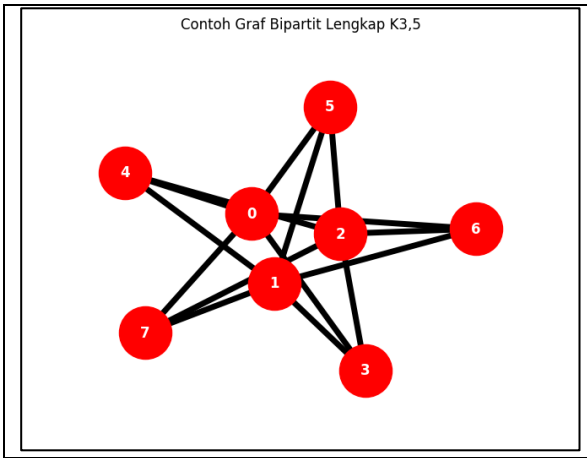


Figure 8. Example image of a Bipartite Graph.

The following is an implementation and creation example *complete bipartite graph* on NetworkX:

```
# Create a complete bipartite graph K3,5
G = nx.complete_bipartite_graph(3,5)

# Print the graph
show_graph(G, title="Contoh Complete Graph K5")
```

Output:

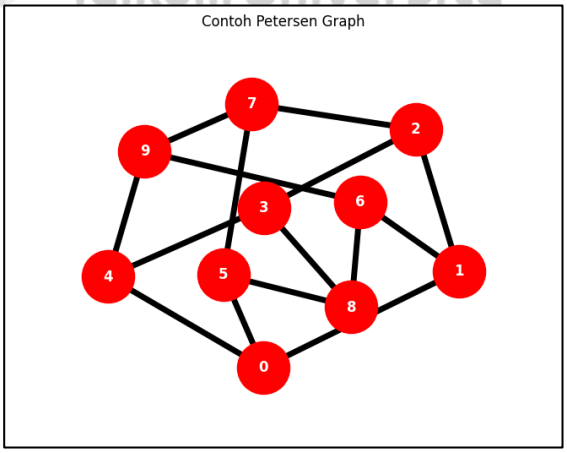


3.10.3. *Petersen Graph*

Graf *Petersen* is a graph consisting of 10 *node* and 15 *edge.* This graph is often used as a study in various graph theory problems, including those related to graph coloring and graphs *Hamilton*. The following is an implementation and creation example *Petersen Graph* on NetworkX:

```
# Create a complete bipartite graph K3,5
G = nx.petersen_graph()

# Print the graph
show_graph(G, title="Contoh Petersen Graph")
```

Output:



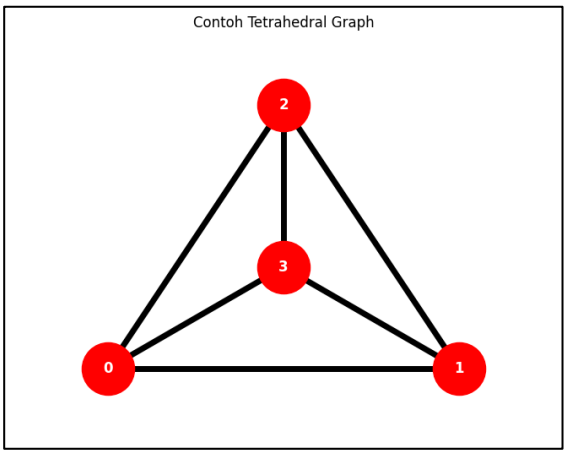
3.10.4. *Tetrahedral Graph*

Graf *tetrahedral* is a graph that represents the structure of a *tetrahedron*, which consists of 4 *node* and 6 *edge*. *Tetrahedral Graph* is a complete graph with 4 *node* (K4). The following is an implementation and example of creation *tetrahedral graph* on NetworkX:

```
# Create a complete bipartite graph K3,5
G = nx.tetrahedral_graph()
```

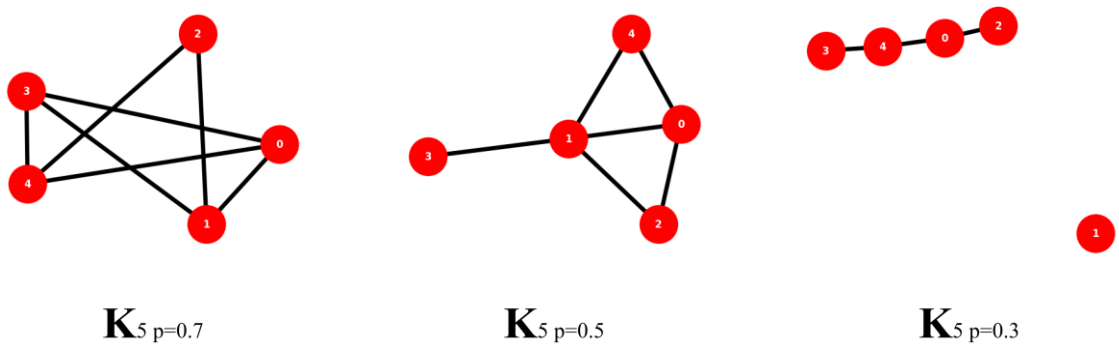
```
# Print the graph
show_graph(G, title="Contoh Tetrahedral Graph")
```

Output:



3.10.5. *Edros-Renyi Graph*

Edros-Renyi Graph is a type of random graph where every *edge* between two *node* has a fixed probability `p` to exist or not exist as in Pictire 9. This graph is usually used to study the properties of random networks, such as degree distribution, connected components, and connectedness threshold.

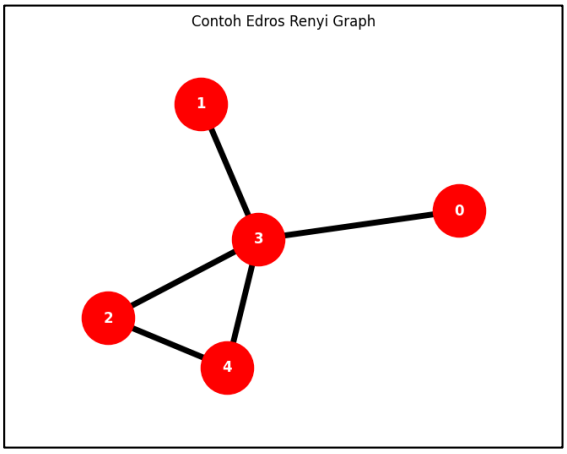


Pictire 9. Example Figure *Edros-Renyi Graph*.

The following is an implementation and example of creation *Edros-Renyi Graph* on NetworkX:

```
# Create an Edros-Renyi K5 graph with p=0.5
G = nx.edros_renyi_graph(5, 0.5)

# Print the graph
show_graph(G, title="Contoh Edros Renyi Graph")
```

Output:

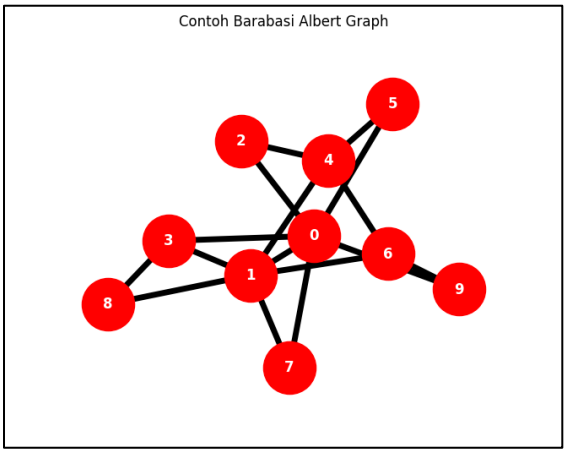


3.10.6. *Barabasi-Albert Graph*

Graf *Barabasi-Albert* is a random graph constructed using a mechanism *preferential attachment*, where *node* new tends to connect to *node* who already has many connections. This graph is used to model social networks, the internet, and biological networks. The following is an implementation and example of graph creation *Barabasi-Albert*:


```
# Create a Barabasi-Albert graph with 10 nodes and 2 edges
G = nx.barabasi_albert_graph(10,2)

# Print the graph
show_graph(G, title="Contoh Barabasi-Albert Graph")
```

Output:

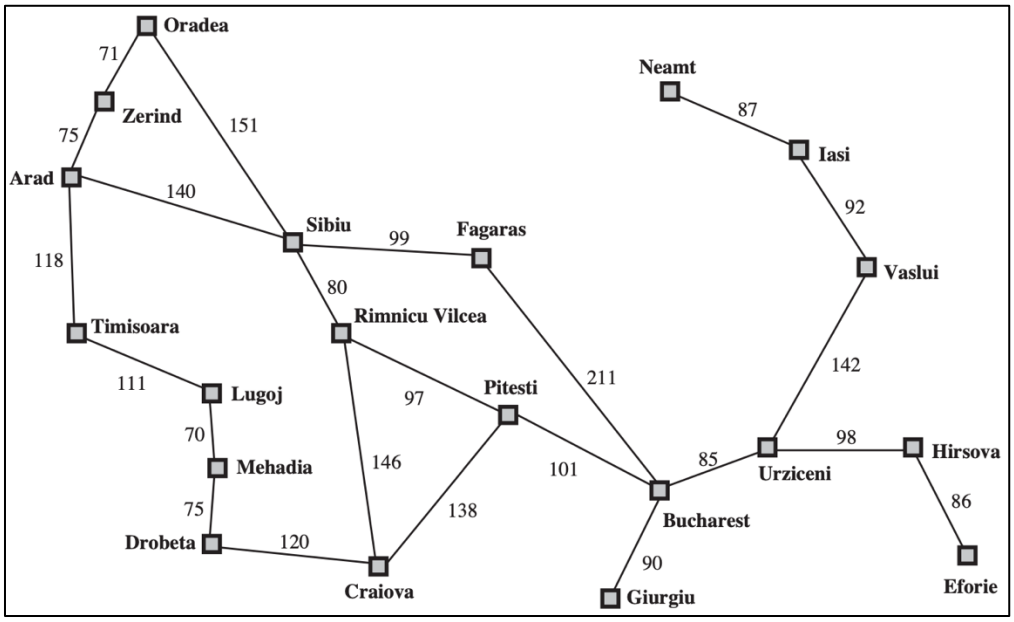


3.11. NetworkX Programming Practice Questions

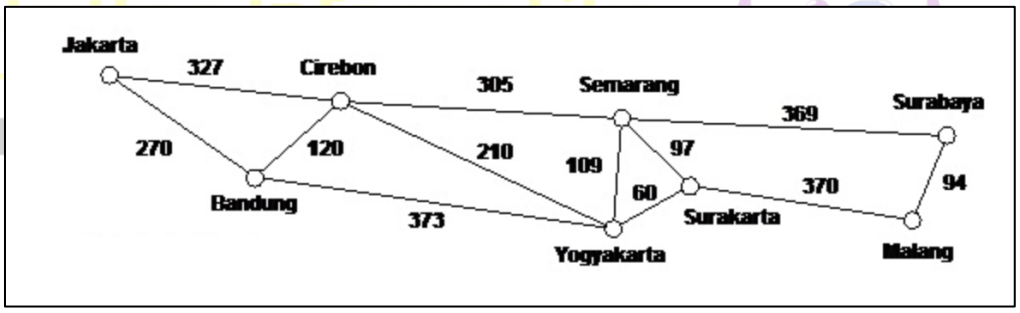
1. Create a weighted graph (*weighted graf*) from the following 2 graph images using the NetworkX library. The node name must match the name of the city listed on it

Figure. The type of graph used is *undirected graph*. Then, using the shortest path function in NetworkX, find all the shortest path pairs from the graph!

a. Map graph of major cities in Europe:



b. Map graph of major cities in Indonesia (Java island):



Modul 4 INTRODUCTION *LIBRARY* MATPLOTLIB

Practical Objectives:
1. Students understand how to create and display graphs using library Matplotlib. 2. Students know the main features available in library Matplotlib.

MAIN REFERENCES:
https://matplotlib.org/stable/plot_types/index.html
https://matplotlib.org/stable/users/explain/quick_start.html
ADDITIONAL REFERENCES:
https://matplotlib.org/stable/index.html
https://matplotlib.org/stable/tutorials/index.html

Matplotlib is *library* Python is used to create data visualizations in graphical form. *Library* it provides various tools to create graphs like *linechart, barchart, histogram, scatter plot,* and many more.

4.1. Method *Import* Matplotlib

Matplotlib should already be present in your Anaconda installation. To use the Matplotlib module, it must be done *import* first. Here is the standard way to *import* library Matplotlib:

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

4.2. Elements *Figure* On Matplotlib

Figure is a basic object in Matplotlib that represents the entire area or canvas on which the plot or graph will be drawn. Each time Matplotlib visualizes a graph, the graph is placed inside a *figure*. Here is an implementation of the main features of *figure*:

Area Plotting

Figure placed in a plotting area created with `` `plt.figure()` `` where all the visual elements (plot, text, legend and annotations) are placed. This is a large canvas that can contain one or more subplots.

```
[1]: # Create an empty figure
     fig = plt.figure()
     fig
```

```
     Output:
     <Figure size 640x480 with 0 Axes>
```

Set Size and Resolution

Size and resolution of figure can be adjusted using parameters such as `figsize` to set dimensions and `dpi` to set the image resolution.

[2]:
```
# Create a figure with a size of 8x6 inches and a resolution of 100 DPI
fig = plt.figure(figsize=(8,6), dpi=100)
fig
```

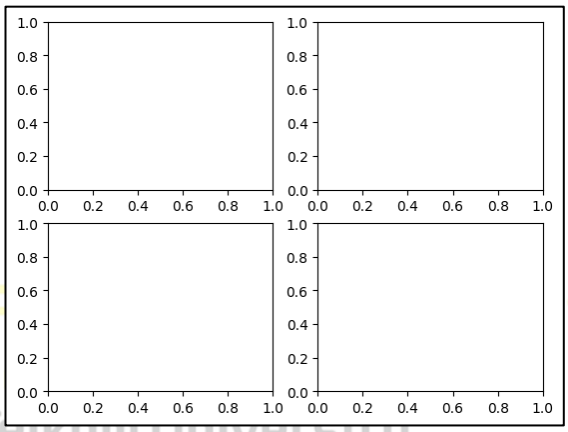
Output:
```
<Figure size 800x600 with 0 Axes>
```

Subplot

Figure can be divided into several *subplot*, which are various individual plots within *the same. Each subplot* can be used to display different graphs.

[3]:
```
# Create a plot with 2x2 subplots
fig, ax = plt.subplots(2,2)
fig, ax
```

Output:

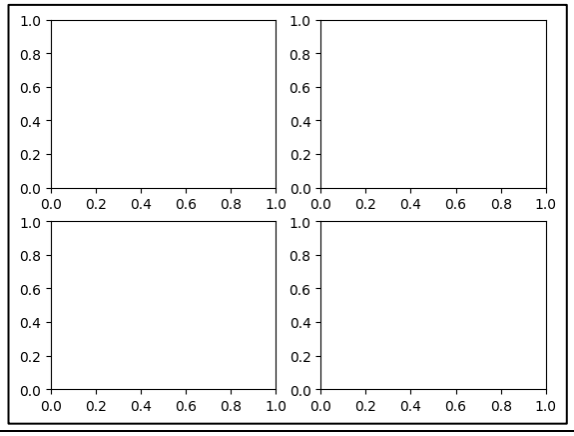


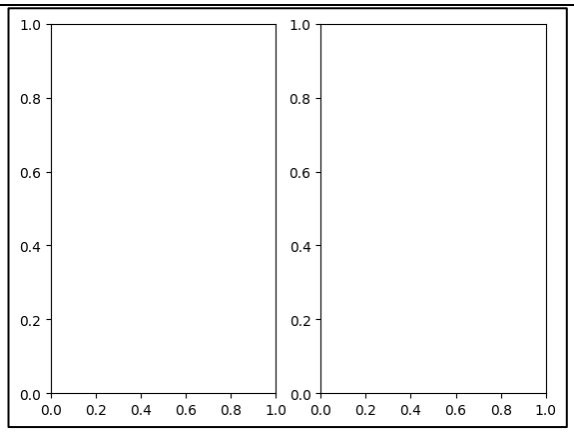
Displaying Figures

Once all elements are added to *figure*, *figure* can be displayed on layers using `plt.show()`

[4]:
```
# First figure
fig, ax = plt.subplots(2,2)
plt.show()

# Second figure
fig, ax = plt.subplots(2,2)
plt.show()
```

Output:

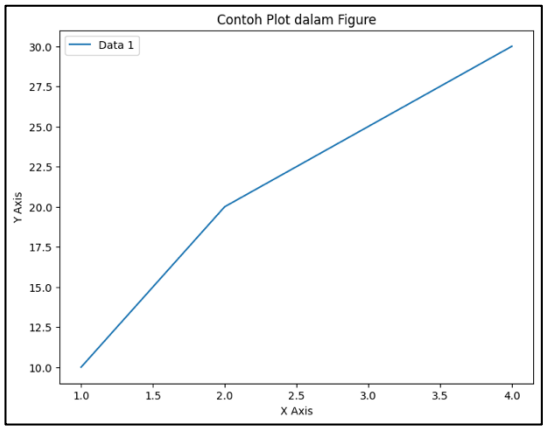




Example of Using Figures in Matplotlib

```
[5]: # Initialize Figure with size 8x6
     fig = plt.figure(figsize=(8,6))

     # Create simple plots
     ax.plot([1,2,3,4], [10,20,25,30], label='Data1')

     # Add labels and titles
     ax.set_xlabel("X Axis")
     ax.set_ylabel("Y Axis")
     ax.set_title("Example of Plot in Figure")

     # Displays the legend
     ax.legend()

     # Displays figures
     plt.show()
```

Output:



4.3. Labeling Plot

Plot labeling is the process of giving a graph a title, axis labels, and a legend to explain the elements displayed. Labeling helps readers understand the content of the graph and the data it represents. Here are some functions to carry out plot labeling easily:

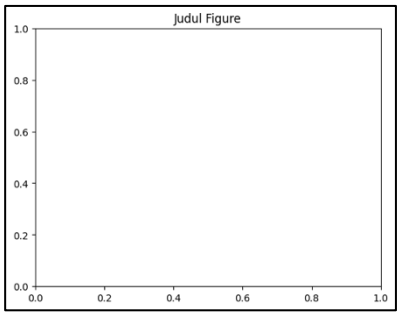
Chart Title

The graph title is used to provide a general idea of what the graph represents.

[1]:

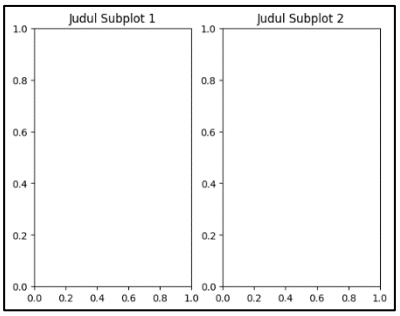
```
# Create figures
fig = plt.figure()
# Add a title to the figure
plt.title("Figure Title")
# Displays figures
plt.show()
```

Output:




```
# Figure with 2 subplots
fig, ax = plt.subplots(1,2)
# Added subplot title 1
ax[0].set_title("Judul 1")
# Added subplot title 2
ax[1].set_title("Judul 2")
# Displays figures
plt.show()
```

Output:



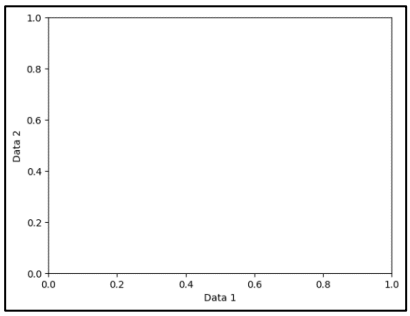
Labeling the Plot

The x-axis and y-axis labels are usually used to indicate the type of data represented on each axis.

[2]:

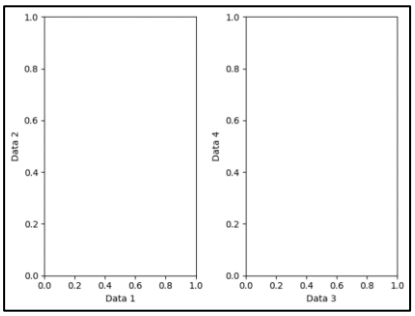
```
# Create figures
fig = plt.figure()
# Add labels to the plot
plt.xlabel("Data 1")
plt.ylabel("Data 2")
# Displays figures
plt.show()
```

Output:




```
# Figure with 2 subplots
fig, ax = plt.subplots(2,2)
# Add subplot labels
ax[0].set_xlabel("Data 1")
ax[0].set_ylabel("Data 2")
ax[1].set_xlabel("Data 3")
ax[1].set_ylabel("Data 4")

# Displays figures
plt.show()
```

Output:



Provides Legend to the Plot

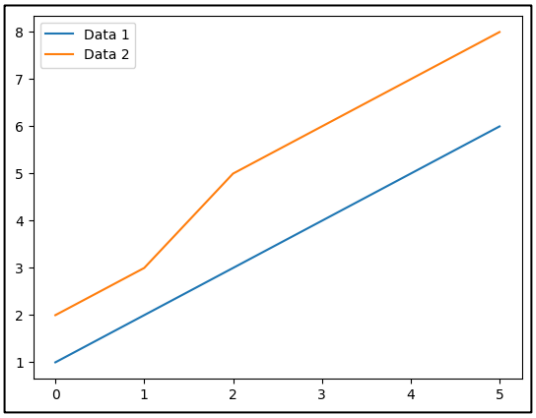
Legends are used to identify each element or data drawn on a plot. Legends help differentiate between various data sets.

[3]:

```
# Create figures
fig = plt.figure()
# Create a linechart in the figure
plt.plot([1, 2, 3, 4, 5, 6], label="Data 1")
plt.plot([2, 3, 5, 6, 7, 8], label="Data 2")
# Displays the legend
plt.legend()
```

```
# Displays figures
plt.show()
```

Output:



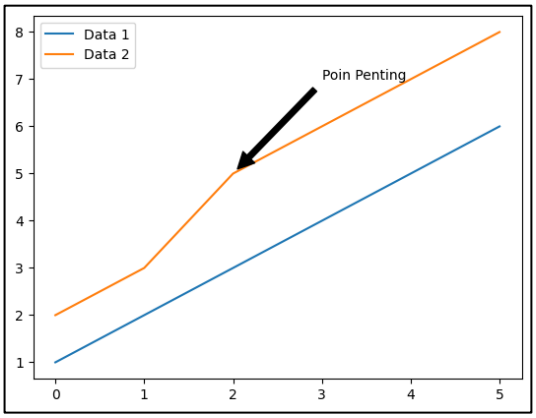
Annotating Labels

Annotations are a way to mark specific points on a plot, usually with text that refers to important points or features in the graph.

[4]:

```
# Create figures
fig = plt.figure()
# Create a linechart
plt.plot([1, 2, 3, 4, 5, 6], label="Data 1")
plt.plot([2, 3, 5, 6, 7, 8], label="Data 2")
# Create annotations on data
plt.annotate("Important Points", xy=(2, 5), xytext=(3, 7),
             arrowprops=dict(facecolor='black', shrink=0.05))
# Displays a legend based on labels in the data
plt.legend()
# Displays figures
plt.show()
```

Output:



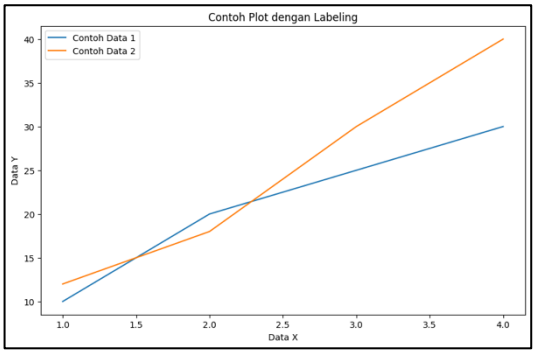
Contoh Penggunaan Labeling dalam Plot

[5]:

```
# Data
x = [1, 2, 3, 4]
y1 = [10, 20, 25, 30]
y2 = [12, 18 , 30, 40]

# Create plots
fig = plt.figure(figsize=(10,6))
plt.plot(x, y1, label="Contoh Data 1")
```

```
plt.plot(x, y2, label="Contoh Data 2")

# Add title and labels
plt.title("Contoh Plot dengan Labeling")
plt.xlabel("Data X")
plt.ylabel("Data Y")

# Add legend
plt.legend()

# Displays figures
plt.show()
```

Output:



4.4. Plot *Pairwise* in Matplotlib

Plot *pairwise* refers to a way to draw graphs that involve pairs of data (x, y) represented in various types of graphs such as *line plot*, *scatter* plot, *bar* plot, *fill between*, dan stack plot. Each plot type has its own uses and characteristics in data visualization. The following is an example and implementation of a pairwise plot in Matplotlib:

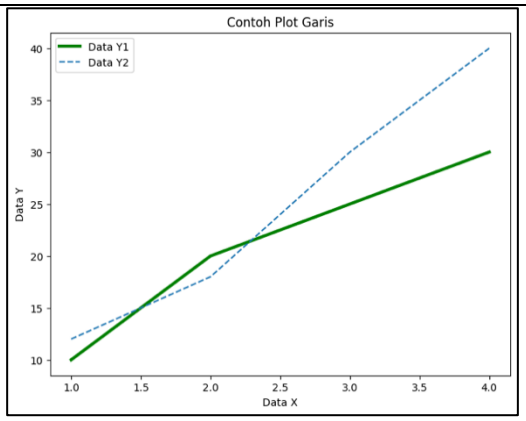
Line Plot (`plt.plot()`)

A line plot is the most basic type of graph, used to illustrate the relationship between two sets of data (x, y) by connecting data points using lines. It is often used to show trends or changes over time.

[1]:

```
# Data
x = [1, 2, 3, 4]
y1 = [10, 20, 25, 30]
y2 = [12, 18, 30, 40]

# Create figures
fig = plt.figure(figsize=(8,6))
# Create a linechart with plt.plot()
plt.plot(x, y1, color="green", linewidth=3, label="Data Y1")
plt.plot(x, y2, linestyle='--', label="Data Y2")
# Labeling plots
plt.title("Example of Line Plot")
plt.xlabel("Data X")
plt.ylabel("Data Y")
# Displays figures
plt.legend()
plt.show()
```

Output:



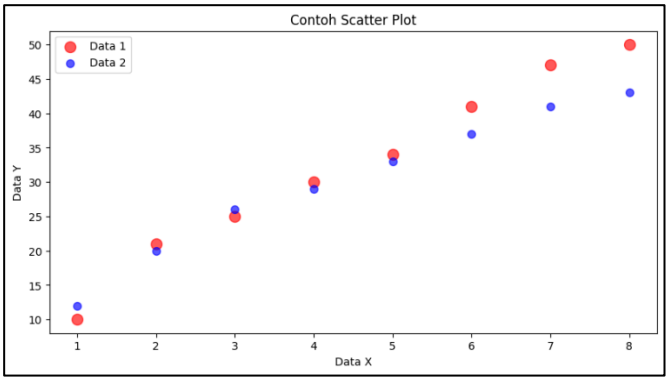
***Scatter* Plot (`plt.scatter()`)**

Scatter plot used to describe the distribution and relationship between two variables. Each point on the *scatter* plot represents an (x, y) pair, and there are no lines connecting these points. *Scatter* plots are useful for seeing patterns, correlation relationships, or outliers in data.

[2]:

```
# Data
x = [1, 2, 3, 4, 5, 6, 7 ,8]
y1 = [10, 21, 25, 30, 34, 41, 47, 50]
y2 = [12, 20, 26, 29, 33, 37, 41, 43]

# Create figures
fig = plt.figure(figsize=(10,5))
# Create a scatter plot with plt.scatter()
plt.scatter(x, y1, s=100, c='red', alpha=0.65, label='Data 1')
plt.scatter(x, y2, s=50, c='blue', alpha=0.65, label='Data 2')
# Labeling plots
plt.title("Example of Scatter Plot")
plt.xlabel("Data X")
plt.ylabel("Data Y")
# Displays figures
plt.legend()
plt.show()
```

Output:



***Bar* Plot (`plt.bar()`)**

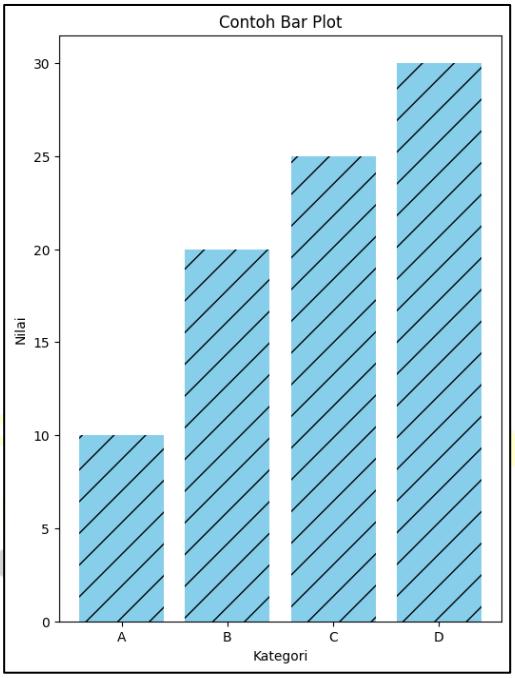
Bar plot used to represent categorical data or to compare multiple data values. The length of the bar shows the size of the data value.

[3]:

```
# Data
category = ['A', 'B', 'C', 'D'] # Label each category
value = [10, 20, 25, 30] # Value each category
```

```
# Create Figures
fig = plt.figure(figsize=(6,8))
# Create a bar plot with plt.bar()
plt.bar(category, value, hatch='/', color='skyblue')
# Labeling plot
plt.title("Example of Bar Plot")
plt.xlabel("Category")
plt.ylabel("Value")
# Displays figures
plt.show()
```

Output:



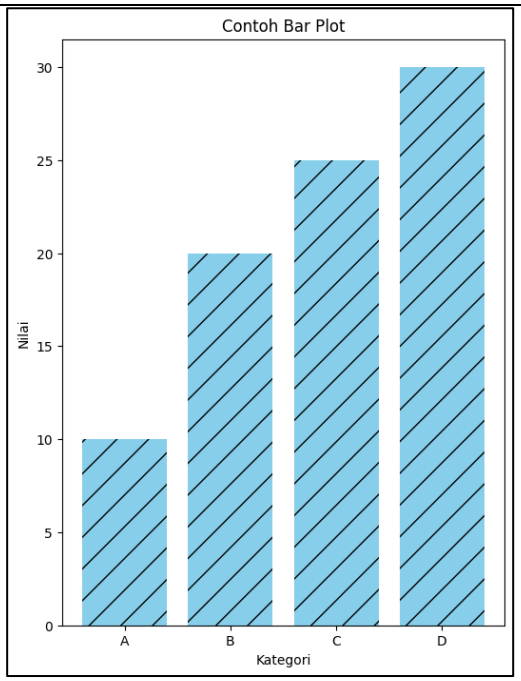
***Bar* Plot (`plt.bar()`)**

Bar plot used to represent categorical data or to compare multiple data values. The length of the bar shows the size of the data value.

[4]:

```
# Data
category = ['A', 'B', 'C', 'D'] # Label each category
nilai = [10, 20, 25, 30] # Value each category

# Create Figures
fig = plt.figure(figsize=(6,8))
# Create a bar plot with plt.bar()
plt.bar(category, value, hatch='/', color='skyblue')
# Labeling plot
plt.title("Example of Bar Plot")
plt.xlabel("Category")
plt.ylabel("Value")
# Displays figures
plt.show()
```

Output:



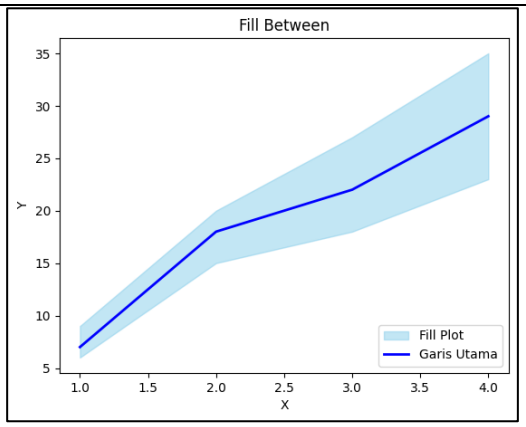
Fill Between **(`plt.feel_between()`)**

Plot *feel between* used to fill the area between two curves and the X-axis. These plots are often used to show the area under a curve or to highlight a specific range of data.

[5]:

```
# Data
x = [1, 2, 3, 4]
y1 = [9, 20, 27, 35]
y_main = [7, 18, 22, 29] # Main line data
y2 = [6, 15, 18, 23]

# Create figures
fig = plt.plot(figsize=(10,6))
# Create a fill_between plot
plt.fill_between(x, y1, y2, color='skyblue', alpha=0.5, label='Fill
Plot')
# Create a main line plot
plt.plot(x, y_main, color='blue', linewidth=2, label='Main Line')
# Labeling plots
plt.title("Fill Between")
plt.xlabel("X")
plt.ylabel("Y")
# Displays figures
plt.legend(loc='lower right')
plt.show()
```

Output:



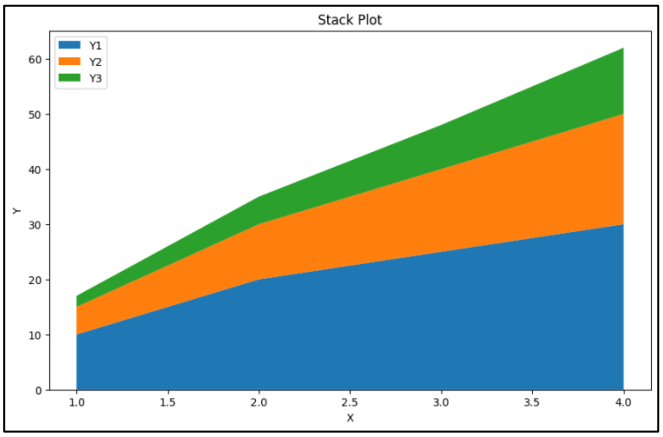
***Stack* Plot (`plt.stackplot()`)**

Stack plot used to display multiple data sets stacked on top of each other. This helps see how individual components contribute to the total cumulatively.

[6]:

```
# Data
x = [1, 2, 3, 4]
y1 = [10, 20, 25, 30]
y2 = [5, 10, 15, 20]
y3 = [2, 5, 8, 12]

# Create figures
fig = plt.figure(figsize=(10,6))
# Create a stack plot with .stackplot()
plt.stackplot(x, y1, y2, y3, labels=['Y1', 'Y2', 'Y3'])
# Labeling plots
plt.title("Stack Plot")
plt.xlabel("X")
plt.ylabel("Y")
# Displays figures
plt.legend(loc='upper left')
plt.show()
```

Output:



4.5. Plot Statistical Distributions In Matplotlib

Statistical distribution plots are used to illustrate how data is spread out, providing an overview of the distribution, median, outliers, and variation in a dataset. The following is an explanation of several types of statistical distribution plots commonly used in Matplotlib:

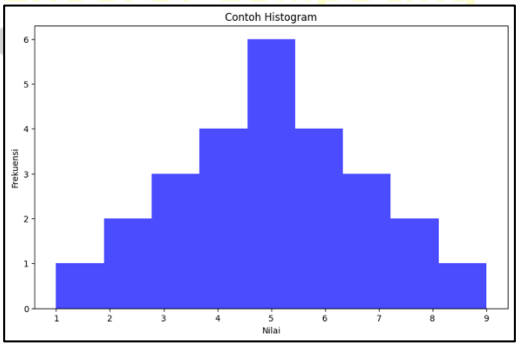
Histogram **(`plt.hist()`)**

Histogram is a type of plot used to represent the frequency distribution of a dataset. Data is divided into bins or intervals, and the height of each bar represents the amount of data that falls within that interval. *Histograms* are useful for understanding the overall distribution of data, including identifying *skewness*, *outliers*, and modes.

[1]:

```
# Data
data = [1,9,2,2,8,8,7,7,7,3,3,3,4,4,4,4,6,6,6,6,5,5,5,5,5,5]

# Create figures
fig = plt.figure(figsize=(10,6))
# Create a histogram with .hist()
plt.hist(data, bins=9, color="blue", alpha=0.7)
# Labeling plots
plt.title("Example of Histogram")
plt.xlabel("Value")
plt.ylabel("Frequency")
# Displays figures
plt.show()
```

Output:



Box **Plot (`plt.boxplot()`)**

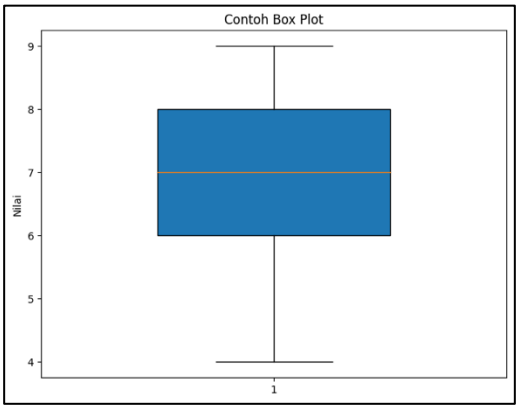
Box plot is a plot that displays the distribution of data based on five main numbers: minimum, Q1, median, Q3, and maximum. *Box* plots are useful for analyzing data distribution, finding *outliers*, and comparing distributions between several groups of data.

[2]:

```
# Data
data = [7, 5, 8, 9, 6, 7, 5, 8, 6, 7, 6, 8, 9, 4, 7, 5, 9, 8,
6, 7]

# Create figures
fig = plt.figure(figsize=(8,6))
# Create a box plot with .boxplot()
plt.boxplot(data, patch_artist=True, widths=0.50)
# Labeling Plots
plt.title("Example of Box Plot")
```

```
plt.ylabel("Value")
# Displays figures
plt.show()
```

Output:



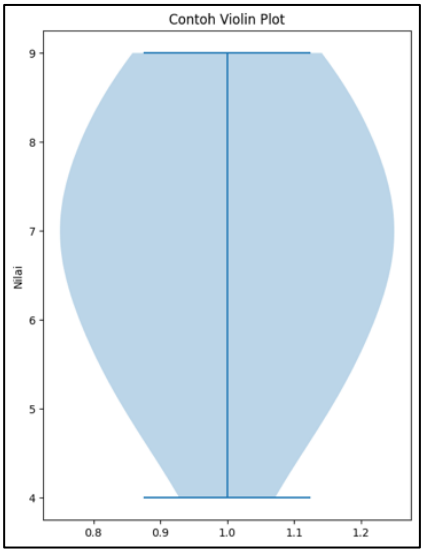
***Violin* Plot (`plt.violinplot()`)**

Violin plot is a combination of *box* plot and *kernel density* plot. It not only conveys the information provided by the *box* plot, but also provides an idea of the probability distribution of the data at various levels of values. The violin plot shape shows the density of the data. This plot is useful for understanding the distribution of data, especially the case of data that has more than one distribution peak.

[3]:

```
# Data
data = [7, 5, 8, 9, 6, 7, 5, 8, 6, 7, 6, 8, 9, 4, 7, 5, 9, 8, 6, 7]

# Create figures
fig = plt.figure(figsize=(6,8))
# Create a violin plot with .violinplot()
plt.violinplot(data)
# Labeling Plots
plt.title("Example of Violin Plot")
plt.ylabel("Value")
# Displays figures
plt.show()
```

Output:



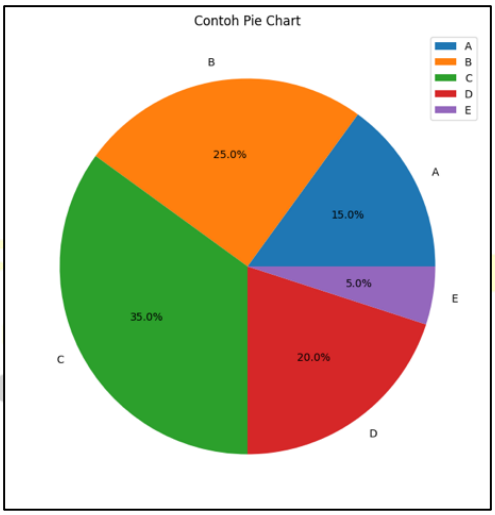
***Pie Chart* (`plt.pie()`)**

Pie chart is a circular plot used to show the proportion or percentage of all data. Each slice in a *pie* chart represents a proportion of data in a particular category. *Pie* charts are useful for visualizing the proportion of data in a category to the overall data in the context of categorical data.

[4]:

```
# Data
label = ['A', 'B', 'C', 'D', 'E']
value = [15, 25, 35, 20, 5]

# Create figures
fig = plt.figure(figsize=(8,8))
# Create a pie chart with .pie()
plt.pie(value, labels=label, autopct='%1.1f%%')
# Labeling plots
plt.title("Example of Pie Chart")
# Displays figures
plt.legend()
plt.show()
```

Output:



4.6. Matplotlib Library Practice Questions

1. Display a graph of the maximum, minimum and average scores for each group of entering student cohorts from the data contained in Table 5!

Table 5. Student score data for Matplotlib library practice questions.

Name	Programming Algorithms	Discrete Mathematics	Basis of Artificial Intelligence	Class Year
Budi	81.24	73.66	95.89	2023
Siti	98.52	84.86	88.7	2024
Asep	91.96	71.03	79.93	2022
Dewi	87.96	97.28	71.91	2024
Andi	74.68	77.76	79.33	2023

Lestari	74.68	89.88	79.76	2022
Joko	71.74	79.35	91.89	2022
Rina	95.99	85.6	89.13	2022
Fajar	88.03	86.4	96.62	2024
Indah	91.24	75.55	84.17	2023
Agus	70.62	99.09	73.59	2022
Fitri	99.1	93.25	91.4	2022
Rudi	94.97	98.18	92.82	2022
Maya	76.37	96.84	86.84	2024
Hadi	75.45	87.94	93.13	2024
Sri	75.5	97.66	84.81	2023
Wawan	79.13	72.65	85.68	2024
Yuni	85.74	75.88	82.83	2022
Bambang	82.96	71.36	70.76	2023
Ayu	78.74	79.76	73.24	2022

2. Display graph images from practice questions in the **Error! Reference source not found.** to the screen using the Matplotlib library! Please set the position of the abscissa and ordinates (X and Y) for each node/city yourself, so that the display produced by Matplotlib resembles the original graph image.

Modul 5 IMPLEMENTATION *BREADTH-FIRST SEARCH*

Practical Objectives:
1. Students understand and are able to implement algorithms *Breadth-First Search* in Python by using *library* NetworkX.

Reference:
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.traversal.breadth_first_search.bfs_edges.html#networkx.algorithms.traversal.breadth_first_search.bfs_edges

Breadth-First Search **(BFS)** is a search or traversal algorithm in a graph structure or *tree*. BFS works by exploring all neighbors of a *node* before moving to the next level. BFS is very useful for finding the shortest path.

NetworkX, BFS can be easily implemented using built-in functions. NetworkX uses `` `nx.bfs_edges()` `` to restore the order *node* explored by BFS. The BFS algorithm makes it possible to explore graphs systematically and obtain the resulting exploration sequence.

5.1. Implementation of BFS on NetworkX

5.1.1. Implementation of BFS On *Tree*

BFS implementation *tree* starting from the root (*root*) and explore all *node* at the same level before moving to the next level. BFS will explore nodes at depth 0 (*root*), then at node depth 1, and so on.

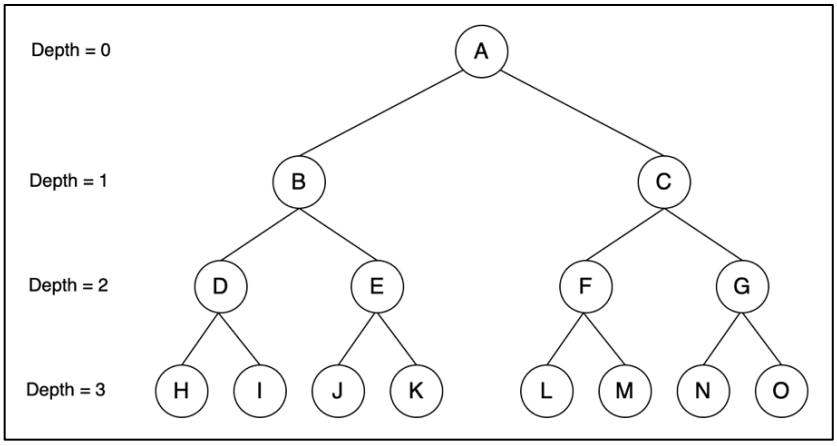


Figure 10. Example of Tree Implementation for BFS.

Here is the implementation code for BFS on *tree* for Figure *10*:

Import *library* which is used

[1]:
```
import network as ns # to create a graph
import matplotlib.pyplot as plt # for graphical plot
```

Function to print plots

[2]:
```
# Support function for printing graphs
def show_graph(G, pos=None, title='') :
  # Create a post if a post is not given
  if pos is None:
      pos = nx.spring_layout(G)

  # Function to draw node
  nx.draw(
      G,                  # Graf NetworkX
      pos,                # Node position
      with_labels=True,   # Displays node name
      node_color='red',   # Node color
      node_size=2000,     # Node size
      font_color="white", # Node label font color
      font_weight="bold", # Node label font thickness
      width=5             # Edge line thickness
  )

  # Takes edge labels if there is weight
  edge_labels = nx.get_edge_attributes(G, 'weight')
  # Function to draw nodes
  nx.draw_networkx_edge_labels(
      G,
      pos,
      edge_labels=edge_labels, # Data weight
      font_color='blue',       # Edge label font color
      font_weight="bold",      # Edge label font weight
      font_size=12,            # Edge label font size
  )

  plt.margins(0.2)  # Provides margin to the plot
  plt.title(title)  # Displays the graph title if given
```

	`plt.show()` # Display graphs using matplotlib
Position of nodes `pos` to match the example image	

[3]:	`pos = {`
	`'A': (0, 3),`
	`'B': (-10, 2), 'C': (10, 2),`
	`'D': (-15, 1), 'E': (-5, 1), 'F': (5, 1), 'G': (15, 1),`
	`'H': (-18, 0), 'I': (-12, 0), 'J': (-8, 0), 'K': (-2, 0),`
	`'L': (2, 0), 'M': (8, 0), 'N': (12, 0), 'O': (18, 0)`
	`}`

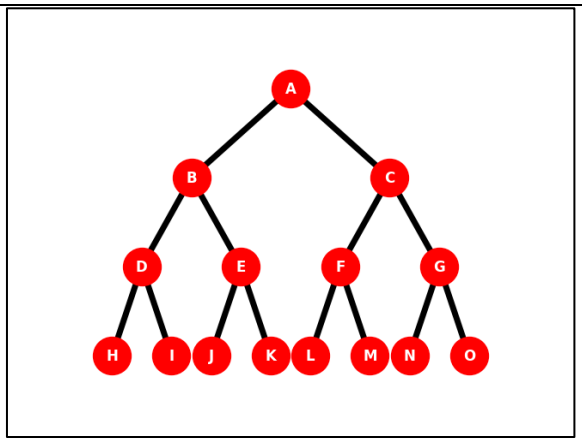
Create a tree with NetworkX	
[4]:	`# Initialize empty graph` `G = nx.Graph()`

[5]:	`# Definition of nodes and edges according to the image`
	`edges = [`
	`('A', 'B'), ('A', 'C'),`
	`('B', 'D'), ('B', 'E'),`
	`('C', 'F'), ('C', 'G'),`
	`('D', 'H'), ('D', 'I'),`
	`('E', 'J'), ('E', 'K'),`
	`('F', 'L'), ('F', 'M'),`
	`('G', 'N'), ('G', 'O')`
	`]`
	
	`# Add edges to the graph`
	`G.add_edges_from(edges)`

Evaluate elements in a graph	

[6]:	`print("Elements in Graph:")`
	`print("List of Nodes in the Graph:", G.nodes())`
	`print("List of Edges in Graph:", G.edges())`
	`print("Number of Nodes in the Graph", G.number_of_nodes())`
	`print("Number of Edges in Graph", G.number_of_edges())`

	Output: Elements on the Graph: List of Nodes in the Graph: ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O'] List of edges in the graph: [('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E'), ('C', 'F'), ('C', 'G'), ('D', 'H'), ('D', 'I'), ('E', 'J'), ('E', 'K'), ('F', 'L'), ('F', 'M'), ('G', 'N'), ('G', 'O')] Number of Nodes in Graph 15 Number of Edges on Graph 14
Graph visualization	
[7]:	`# Prints the graph` `show_graph(G, pos=pos)`
	Output:



Order *Breadth-First Search*

[8]:
```
# Create a loop for the Breadth First Search sequence
print("Sequence of visited nodes:")
i = 1
for nodes in nx.bfs_edges(G, source='A'):
  print(f"{i}. {nodes[0]} - {nodes[1]}")
  i  += 1
```

```
Output:
Order of visited nodes:
1. A - B
2. A - C
3. B - D
4. B - E
5. C - F
6. C - G
7. D - H
8. D - I
9. E - J
10. E - K
11. F - L
12. F - M
13. G - N
14. G - O
```

[9]:
```
# loop for Breadth First Search sequence with depth 2
print("Sequence of visited nodes:")
i = 1
for nodes in nx.bfs_edges(G, source='A', depth_limit=2):
  print(f"{i}. {nodes[0]} - {nodes[1]}")
  i  += 1
```

```
Output:
Order of visited nodes:
1. A - B
2. A - C
3. B - D
4. B - E
5. C - F
6. C - G
```

5.1.2. Implementation of BFS On *Directed Graph*

BFS on *directed graph* starting from a starting node (*source node*) and explore all the neighbors that can be reached directly from *node* the. Once all neighbors on the first level have been explored, BFS proceeds to neighbors from all over *node* at this level, the next level, and so on. On *directed graph*, BFS just follows the direction *edge* while exploring the graph. BFS on *directed graph* can be used to determine the shortest path from one *node* the *node* everything else in the graph is unweighted.

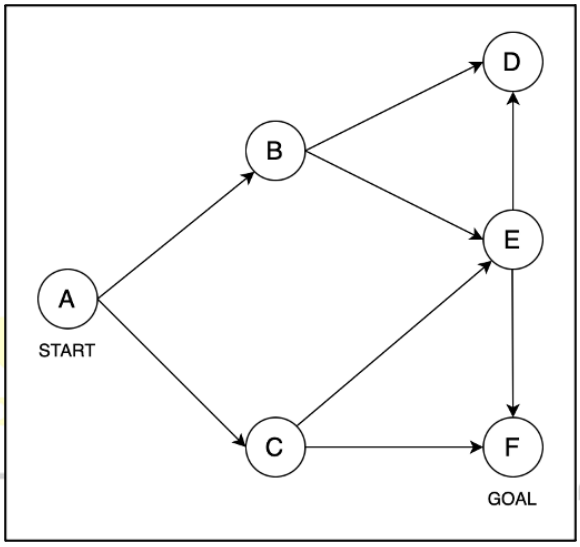


Figure 11. Example of directed graph for DFS implementation.

Here is the code for implementing BFS on *directed graph* for Figure *11*:

Import *library* which is used

```
[1]: import network as ns # to create a graph
     import matplotlib.pyplot as plt # for graphical plot
```

Function to print plots

```
[2]: # Support function for printing graphs
     def show_graph(G, pos=None, title='', labels=labels) :
       # Create a post if a post is not given
       if pos is None:
           pos = nx.spring_layout(G)

       # Function to draw node
       nx.draw(
           G,                  # Graf NetworkX
           pos,                # Node position
           with_labels=True,   # Displays node name
           labels=labels       # Displays the label of each node
           node_color='red',   # Node color
```

```
        node_size=2000,      # Node size
        font_color="white", # Node label font color
        font_weight="bold", # Node label font thickness
        width=5             # Edge line thickness
    )

    # Takes edge labels if there is weight
    edge_labels = nx.get_edge_attributes(G, 'weight')
    # Function to draw nodes
    nx.draw_networkx_edge_labels(
        G,
        pos,
        edge_labels=edge_labels, # Data weight
        font_color='blue',       # Edge label font color
        font_weight="bold",      # Edge label font weight
        font_size=12,            # Edge label font size
    )

    plt.margins(0.2)  # Provides margin to the plot
    plt.title(title)  # Displays the graph title if given
    plt.show()        # Display graphs using matplotlib
```

Position *node* `pos` and labels *start* And *goal* to match the example image

[3]:

```
pos = {
    'A': (0, 1),
    'B': (1, 2),
    'C': (1, 0),
    'D': (2, 3),
    'E': (2, 1.5),
    'F': (2, 0)
}

# Records start and goal nodes
labels = {node: node for node in G.nodes()}
labels['A'] = 'A\nStart'
labels['F'] = 'F\nGoal'
```

Make *directed graph* with NetworkX

[4]:

```
# Initialize an empty directed graph
G = nx.DiGraph()
```

[5]:

```
# Definition of nodes and edges according to the image
edges = [
    ('A', 'B'), ('A', 'C'),
    ('B', 'D'), ('B', 'E'),
    ('C', 'F'), ('C', 'G'),
    ('D', 'H'), ('D', 'I'),
    ('E', 'J'), ('E', 'K'),
    ('F', 'L'), ('F', 'M'),
    ('G', 'N'), ('G', 'O')
]

# Add edges to the graph
G.add_edges_from(edges)
```

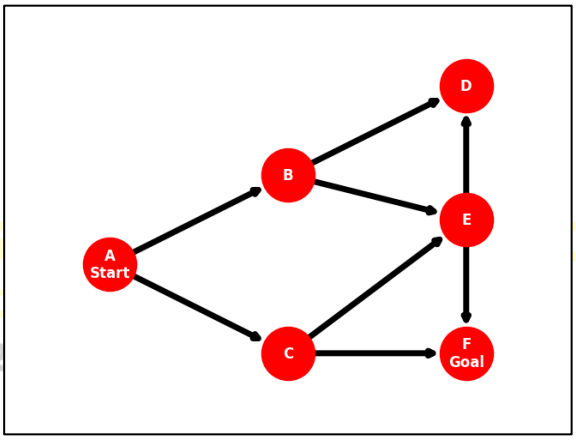
Add labels *start* And *goal*

[6]:

```
# Records start and goal nodes
labels = {node: node for node in G.nodes()}
labels['A'] = 'A\nStart'
```

	`labels['F'] = 'F\nGoal'`
Evaluate elements in a graph	

[7]:	`print("Elements in Graph:")`
	`print("List of Nodes in the Graph:", G.nodes())`
	`print("List of Edges in Graph:", G.edges())`
	`print("Number of Nodes in the Graph", G.number_of_nodes())`
	`print("Number of Edges in Graph", G.number_of_edges())`

	Output: Elements on the Graph: List of Nodes in the Graph: ['A', 'B', 'C', 'D', 'E', 'F'] List of edges in the graph: [('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E'), ('C', 'E'), ('C', 'F'), ('E', 'D'), ('E', 'F')] Number of Nodes in the Graph 6 Number of Edges on Graph 8
Graph visualization	
[8]:	`# Prints the graph` `show_graph(G, pos=pos, labels=labels)`
	Output: 
Order *Breadth-First Search*	

[9]:	`# Create a loop for the Breadth First Search sequence`
	`print("Sequence of visited nodes:")`
	`i = 1`
	`for nodes in nx.bfs_edges(G, source='A'):`
	`  print(f"{i}. {nodes[0]} - {nodes[1]}")`
	`  i  += 1`

	Output: Order of visited nodes: 1. A - B 2. A - C 3. B - D 4. B - E 5. C - F

[10]:	`# loop for Breadth First Search sequence with depth`
	`print("Sequence of visited nodes:")`
	`i = 1`
	`for nodes in nx.bfs_edges(G, source='A', depth_limit=1):`
	`  print(f"{i}. {nodes[0]} - {nodes[1]}")`
	`  i  += 1`

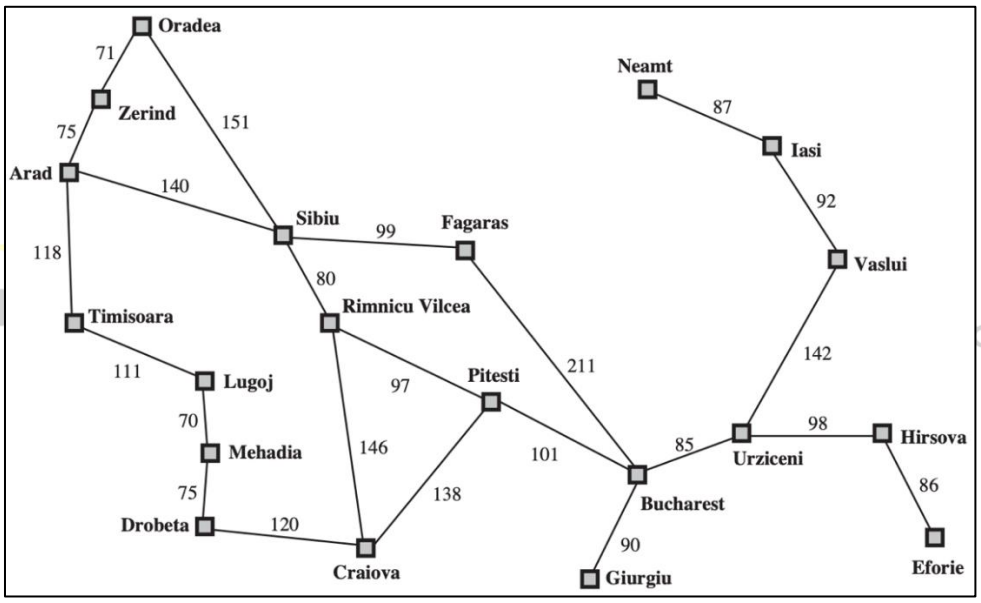
	Output: Order of visited nodes:

	1. A - B 2. A - C

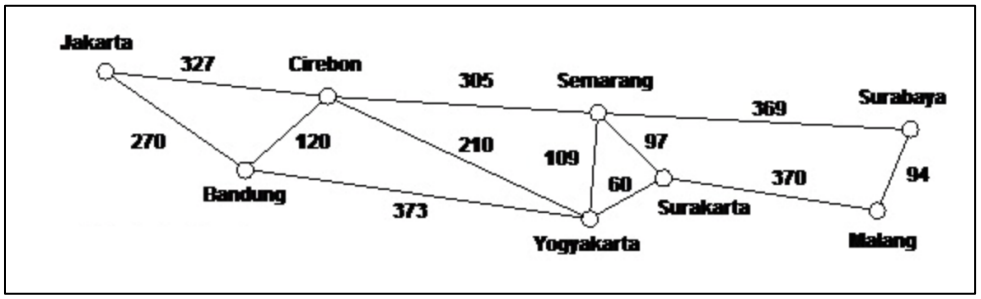
5.2. Exercise Questions

1. Create a weighted graph (*weighted graf*) from the following 2 graph images using the NetworkX library. Then do the search *Breadth-First Search* use *library* NetworkX! Then check the results manually! For European graphs, the city of origin (*source*) is Arad. For the Java Island graph, the city of origin (*source*) is Bandung. Check whether weight information is used or not in this Breadth First Search algorithm!

   a. Map graph of major cities in Europe:



   b. Map graph of major cities in Indonesia (Java island):



Modul 6 IMPLEMENTATION DEPTH-FIRST SEARCH

Practical Objectives:
1. Students understand and are able to implement algorithms *Depth-First Search* in Python by using *library* NetworkX.

Reference:
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.traversal.depth_first_search.dfs_edges.html#networkx.algorithms.traversal.depth_first_search.dfs_edges

Depth-First Search (DFS*)* is a search or traversal algorithm that can be used on graph structures or tree. DFS works by exploring one branch or path as deeply as possible before returning and exploring another path. DFS is used to search for all possible paths to solve a problem that involves deep search.

In NetworkX, DFS can be easily implemented using functions `` `nx.dfs_edges()` `` to restore the order node explored by DFS. The DFS algorithm makes it possible to explore graphs systematically and obtain a resulting exploration sequence based on a deep search on each node before moving to node next.

6.1. DFS Implementation On Tree

DFS starts from *node* root (*root*) and follow the path until you reach the leaf (*leaf*) or the last point, then return to *node* beforehand to explore other unvisited paths.

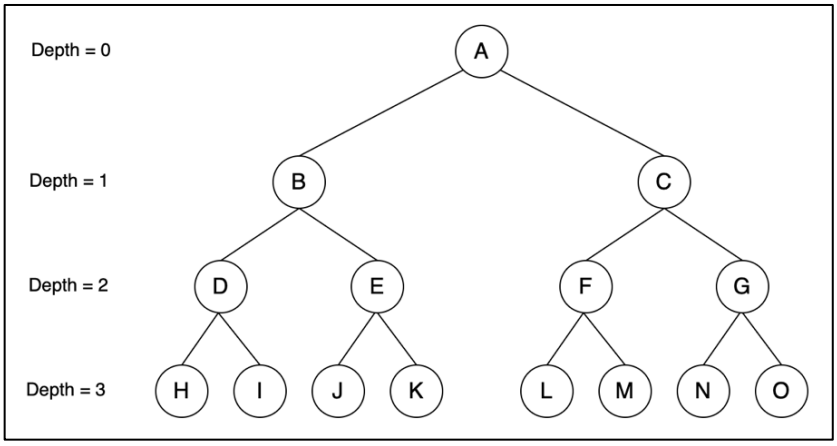


Figure 12. Example tree for DFS.

Here is the implementation code for DFS on tree for Figure *12*:

Import *library* which is used

[1]:
```
import network as ns # to create a graph
import matplotlib.pyplot as plt # for graphical plot
```

Function to print the plot

[2]:
```
# Support function for printing graphs
def show_graph(G, pos=None, title='') :
  # Create a post if a post is not given
  if pos is None:
      pos = nx.spring_layout(G)

  # Function to draw nodes
  nx.draw(
      G,                  # Graf NetworkX
      pos,                # Node position
      with_labels=True,   # Displays the code name
      node_color='red',   # Node color
      node_size=2000,     # Node size
      font_color="white", # Node label font color
      font_weight="bold", # Node label font thickness
      width=5             # Edge line thickness
  )

  # Takes edge labels if there is weight
  edge_labels = nx.get_edge_attributes(G, 'weight')
  # Function to draw nodes
  nx.draw_networkx_edge_labels(
      G,
      pos,
      edge_labels=edge_labels, # Data weight
      font_color='blue',       # Edge label font color
      font_weight="bold",      # Edge label font weight
      font_size=12,            # Edge label font size
  )

  plt.margins(0.2)  # Provides margin to the plot
  plt.title(title)  # Displays the graph title if given
```

	`plt.show()` # Display graphs using matplotlib
Position of nodes `pos` to match the example image	

[3]:	`pos = {`
	`'A': (0, 3),`
	`'B': (-10, 2), 'C': (10, 2),`
	`'D': (-15, 1), 'E': (-5, 1), 'F': (5, 1), 'G': (15, 1),`
	`'H': (-18, 0), 'I': (-12, 0), 'J': (-8, 0), 'K': (-2, 0),`
	`'L': (2, 0), 'M': (8, 0), 'N': (12, 0), 'O': (18, 0)`
	`}`

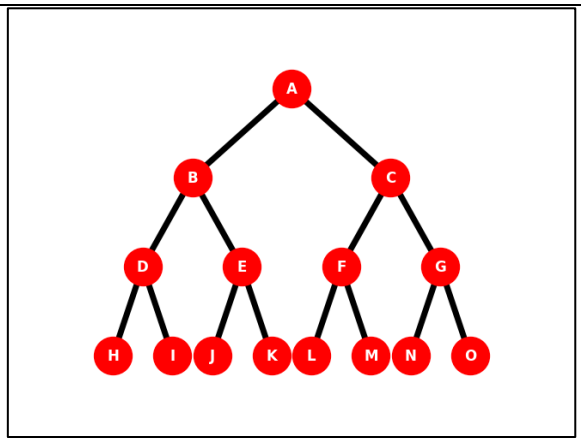
Make *tree* with NetworkX	
[4]:	`# Initialize empty graph` `G = nx.Graph()`

[5]:	`# Definition of nodes and edges according to the image`
	`edges = [`
	`('A', 'B'), ('A', 'C'),`
	`('B', 'D'), ('B', 'E'),`
	`('C', 'F'), ('C', 'G'),`
	`('D', 'H'), ('D', 'I'),`
	`('E', 'J'), ('E', 'K'),`
	`('F', 'L'), ('F', 'M'),`
	`('G', 'N'), ('G', 'O')`
	`]`
	
	`# Add edges to the graph`
	`G.add_edges_from(edges)`

Evaluate elements in a graph	

[5]:	`print("Elements in Graph:")`
	`print("List of Nodes in the Graph:", G.nodes())`
	`print("List of Edges in Graph:", G.edges())`
	`print("Number of Nodes in the Graph", G.number_of_nodes())`
	`print("Number of Edges in Graph", G.number_of_edges())`

	Output: Elements on the Graph: Daftar Node pada Graf: ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O'] Daftar Edge pada Graf: [('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E'), ('C', 'F'), ('C', 'G'), ('D', 'H'), ('D', 'I'), ('E', 'J'), ('E', 'K'), ('F', 'L'), ('F', 'M'), ('G', 'N'), ('G', 'O')] Jumlah Node pada Graf 15 Number of Edges on Graph 14
Graph visualization	
[6]:	`# Prints the graph` `show_graph(G, pos=pos)`
	Output:



Order *Depth-First Search*

[7]:
```
# Create a loop for the Depth First Search sequence
print("Sequence of visited nodes:")

i = 1
for nodes in nx.dfs_edges(G, source='A'):
  print(f"{i}. {nodes[0]} - {nodes[1]}")
  i  += 1
```

```
Output:
Order of visited nodes:
1. A - B
2. B - D
3. D - H
4. D - I
5. B - E
6. E - J
7. E - K
8. A - C
9. C - F
10. F - L
11. F - M
12. C - G
13. G - N
14. G - O
```

[8]:
```
# loop for Depth First Search sequence with depth 2
print("Sequence of visited nodes:")

i = 1
for nodes in nx.dfs_edges(G, source='A', depth_limit=2):
  print(f"{i}. {nodes[0]} - {nodes[1]}")
  i  += 1
```

```
Output:
Order of visited nodes:
1. A - B
2. B - D
3. B - E
4. A - C
5. C - F
6. C - G
```

6.2. DFS Implementation On Directed Graph

Implementation of DFS on directed graph involves exploring a directed graph by investigating each path or branch of the node early as deep as possible before switching to another lane. In a directed graph, the direction from each edge attention, so DFS just follows the direction edge when exploring node.

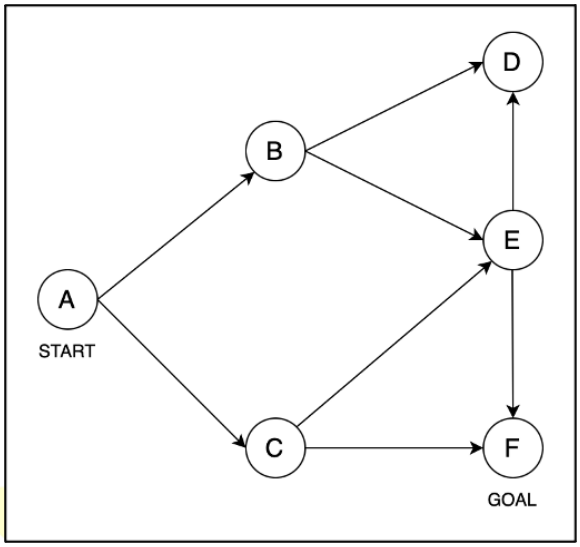


Figure 13. Example of directed graph for DFS.

Here is the implementation code for DFS on *directed graph* for Figure *13*:

Import *library* which is used

[1]:
```
import network as ns # to create a graph
import matplotlib.pyplot as plt # for graphical plot
```

Function to print the plot

[2]:
```
# Support function for printing graphs
def show_graph(G, pos=None, title='', labels={}) :
  # Create a post if a post is not given
  if pos is None:
      pos = nx.spring_layout(G)

  # Function to draw nodes
  nx.draw(
      G,                  # Graf NetworkX
      pos,                # Node position
      with_labels=True,   # Displays the node name
      labels=labels       # Displays the label of each node
      node_color='red',   # Node color
      node_size=2000,     # Node size
      font_color="white", # Node label font color
      font_weight="bold", # Node label font thickness
      width=5             # Edge line thickness
```

```
    )

    # Takes edge labels if there is weight
    edge_labels = nx.get_edge_attributes(G, 'weight')
    # Function to draw nodes
    nx.draw_networkx_edge_labels(
        G,
        pos,
        edge_labels=edge_labels, # Data weight
        font_color='blue',       # Edge label font color
        font_weight="bold",      # Edge label font weight
        font_size=12,            # Edge label font size
    )

    plt.margins(0.2)  # Provides margin to the plot
    plt.title(title)  # Displays the graph title if given
    plt.show()        # Display graphs using matplotlib
```

Positions of nodes `pos` and labels *start* And *goal* to match the example image

[3]:

```
pos = {
    'A': (0, 1),
    'B': (1, 2),
    'C': (1, 0),
    'D': (2, 3),
    'E': (2, 1.5),
    'F': (2, 0)
}
```

Make *directed graph* with NetworkX

[4]:

```
# Initialize an empty directed graph
G = nx.DiGraph()
```

[5]:

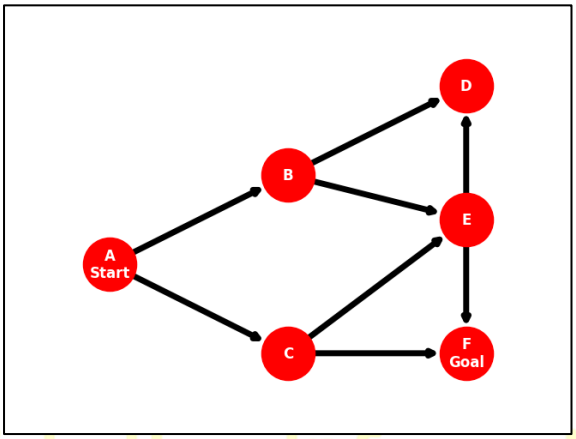
```
# Definition of nodes and edges according to the image
edges = [
    ('A', 'B'), ('A', 'C'),
    ('B', 'D'), ('B', 'E'),
    ('C', 'F'), ('C', 'G'),
    ('D', 'H'), ('D', 'I'),
    ('E', 'J'), ('E', 'K'),
    ('F', 'L'), ('F', 'M'),
    ('G', 'N'), ('G', 'O')
]

# Add edges to the graph
G.add_edges_from(edges)
```

Evaluate elements in a graph

[6]:

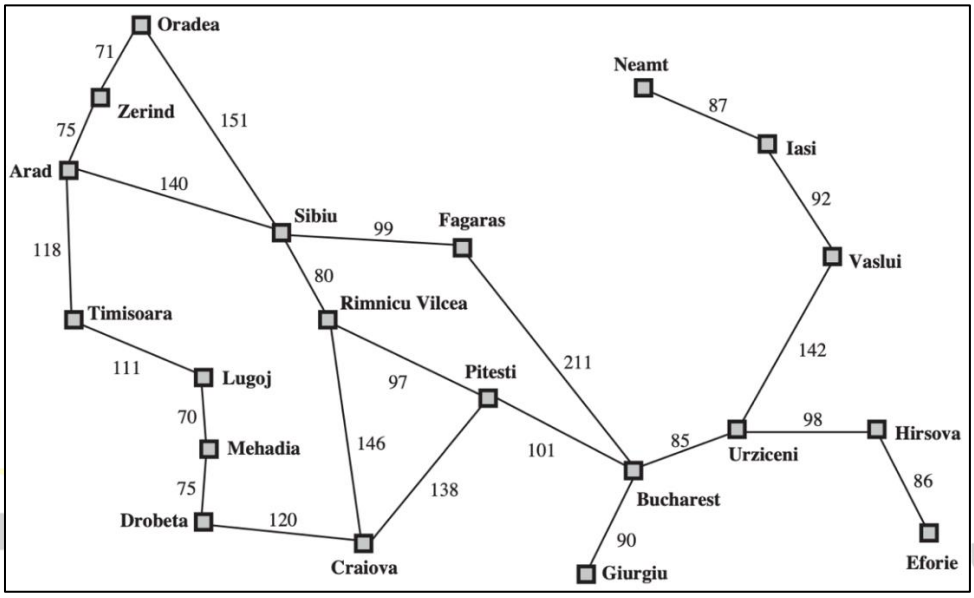
```
print("Elements in Graph:")
print("List of Nodes in the Graph:", G.nodes())
print("List of Edges in Graph:", G.edges())
print("Number of Nodes in the Graph", G.number_of_nodes())
print("Number of Edges in Graph", G.number_of_edges())
```

```
Output:
Elements on the Graph:
List of Graph Nodes: ['A', 'B', 'C', 'D', 'E', 'F'] List of Graph
Edges: [('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E'), ('C', 'E'),
('C', 'F'), ('E', 'D'), ('E', 'F')]
```

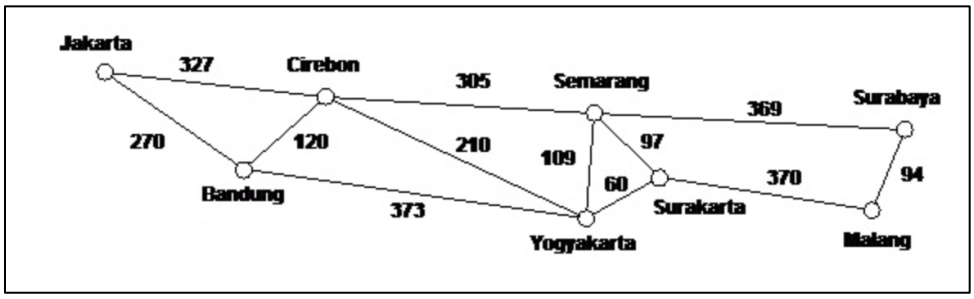
	Number of Nodes in Graph 6 Number of Edges on Graph 8
Add labels start And goal	
[7]:	# Records start and goal nodes **labels = {node: node for node in G.nodes()}** **labels['A'] = 'A\nStart'** **labels['F'] = 'F\nGoal'**
Graph visualization	
[8]:	# Prints the graph **show_graph(G, pos=pos, labels=labels)**
	Output: 
Order Depth-First Search	
[9]:	# Create a loop for the Depth First Search sequence **print("Sequence of visited nodes:")** **i = 1** **for nodes in nx.dfs_edges(G, source='A'):** ** print(f"{i}. {nodes[0]} - {nodes[1]}")** ** i += 1**
	Output: Order of visited nodes: 1. A - B 2. B - D 3. B - E 4. E - F 5. A - C
[10]:	# loop for Depth First Search sequence with depth 1 **print("Sequence of visited nodes:")** **i = 1** **for nodes in nx.dfs_edges(G, source='A', depth_limit=1):** ** print(f"{i}. {nodes[0]} - {nodes[1]}")** ** i += 1**
	Output: Order of visited nodes: 1. A - B 2. A - C

6.3. Exercise Questions

1. Create a weighted graph (*weighted graf*) from the following 2 graph images using the NetworkX library. Then do the search *Depth First Search* using the NetworkX library! Then check the results manually! For a city graph of large European cities, the city of origin (*source*) is Arad. For the graph of Java Island cities, the city of origin (*source*) is Bandung. Check whether weight information is used or not in this Breadth First Search algorithm!
    a. Map graph of major cities in Europe:



    b. Map graph of major cities in Indonesia (Java island):



Modul 7 IMPLEMENTATION UNIFORM COST SEARCH

Practical Objectives:
1. Students understand and are able to implement algorithms *Uniform Cost Search* in Python by using *library* NetworkX.

Reference:
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.shortest_paths.astar.astar_path.html

Uniform Cost Search (UCS) is an optimal path search algorithm in a graph that searches for a path with cost lowest of node early (start) the node objective (goal). UCS is a variant of BFS, but considering weight from each edge in the graph. This algorithm is useful for finding paths with totals weight the lowest.

In NetworkX, UCS is implemented by leveraging functions `` `nx.astar_path()` `` by setting a heuristic value (`` `heuristic` ``) to 0, the A* algorithm essentially functions as UCS because it only considers actual costs without adding estimates to the objective.

7.1. UCS Implementation On *Tree*

UCS on *tree* is a search that aims to find a path from *root* the *node* destination with the lowest total cost. On *tree*, UCS explores by selecting *node* which has the lowest cumulative cost of *root*.

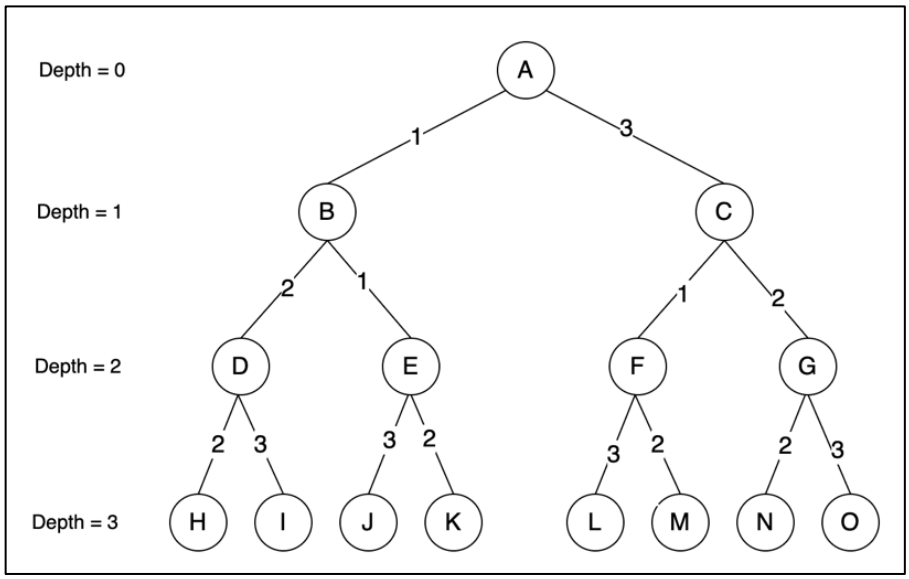


Figure 14. Example tree for UCS.

Here is the implementation code for UCS on *tree* for Figure *14*:

Import library which is used

[1]:
```
import network as ns # to create a graph
import matplotlib.pyplot as plt # for graphical plot
```

Function to print the plot

[2]:
```
# Support function for printing graphs
def show_graph(G, pos=None, title='') :
  # Create a post if a post is not given
  if pos is None:
      pos = nx.spring_layout(G)

  # Function to draw nodes
  nx.draw(
      G,                  # Graf NetworkX
      pos,                # Node position
      with_labels=True,   # Displays the node name
      node_color='red',   # Node color
      node_size=2000,     # Node size
      font_color="white", # Node label font color
      font_weight="bold", # Node label font thickness
      width=5             # Edge line thickness
  )

  # Takes edge labels if there is weight
  edge_labels = nx.get_edge_attributes(G, 'weight')
  # Function to draw nodes
  nx.draw_networkx_edge_labels(
      G,
      pos,
      edge_labels=edge_labels, # Data weight
      font_color='blue',       # Node label font color
      font_weight="bold",      # Node label font weight
      font_size=12,            # Edge label font size
  )

  plt.margins(0.2)  # Provides margin to the plot
  plt.title(title)  # Displays the graph title if given
  plt.show()        # Display graphs using matplotlib
```

Positions of nodes `pos` to match the example image

[3]:
```
pos = {
    'A': (0, 3),
    'B': (-10, 2), 'C': (10, 2),
    'D': (-15, 1), 'E': (-5, 1), 'F': (5, 1), 'G': (15, 1),
    'H': (-18, 0), 'I': (-12, 0), 'J': (-8, 0), 'K': (-2, 0),
    'L': (2, 0), 'M': (8, 0), 'N': (12, 0), 'O': (18, 0)
}
```

Make *tree* with NetworkX

[4]:
```
# Initialize empty graph
G = nx.Graph()
```

[5]:
```
# Definition of nodes and edges according to the image
edges = [
    ('A', 'B', 1), ('A', 'C', 3),
    ('B', 'D', 2), ('B', 'E', 1),
    ('C', 'F', 1), ('C', 'G', 2),
```

```
    ('D', 'H', 2), ('D', 'I', 3),
    ('E', 'J', 3), ('E', 'K', 2),
    ('F', 'L', 3), ('F', 'M', 2),
    ('G', 'N', 2), ('G', 'O', 3)
]
# Add edges to the graph
G.add_weighted_edges_from(edges)
```

Evaluate elements in a graph

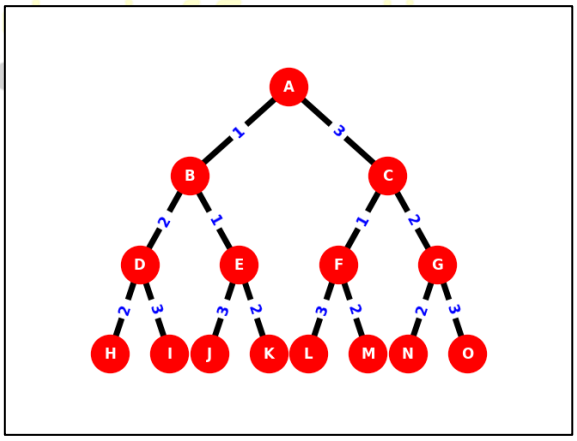
```
[5]: print("Elements in Graph:")
     print("List of Nodes in the Graph:", G.nodes())
     print("List of Edges in Graph:", G.edges())
     print("Number of Nodes in the Graph", G.number_of_nodes())
     print("Number of Edges in Graph", G.number_of_edges())
```

```
Output:
Elements on the Graph:
List of node in graph: ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I',
'J', 'K', 'L', 'M', 'N', 'O']
Daftar Edge pada Graf: [('A', 'B'), ('A', 'C'), ('B', 'D'), ('B',
'E'), ('C', 'F'), ('C', 'G'), ('D', 'H'), ('D', 'I'), ('E', 'J'),
('E', 'K'), ('F', 'L'), ('F', 'M'), ('G', 'N'), ('G', 'O')]
Jumlah Node pada Graf 15
Number of Edges on Graph 14
```

Graph visualization

```
[6]: # Prints the graph
     show_graph(G, pos=pos)
```

Output:



Create a heuristic function

```
[7]: def heuristable(a, b):
         return 0
```

Specifies the path from *node* And when *node* M with UCS

```
[8]: # Search for visited nodes with the smallest weight from A to
     path_node = nx.astar_path(G, 'A', 'M', heuristic=heuristable,
     weight='weight')

     # Find the smallest weight from A to M
     path_length = nx.astar_path_length(G, 'A', 'M', heuristic=heuristable,
     weight='weight')

     # Print results
```

	print("Node yang dikunjungi dari A ke M:", path_node) **print("Besar weight terkecil dari A ke M:", path_length)**
	Output: Nodes visited from A to M: ['A', 'C', 'F', 'M'] Smallest weight from A to M: 6

7.2. UCS Implementation On Directed Graph

UCS on a graph is used to find the path with the lowest cost *node* early (*start*) the *node* objective (*goal*). UCS is always expanding *node* with the lowest cumulative costs first. On *directed graph* direction *edge* affects node expansion.

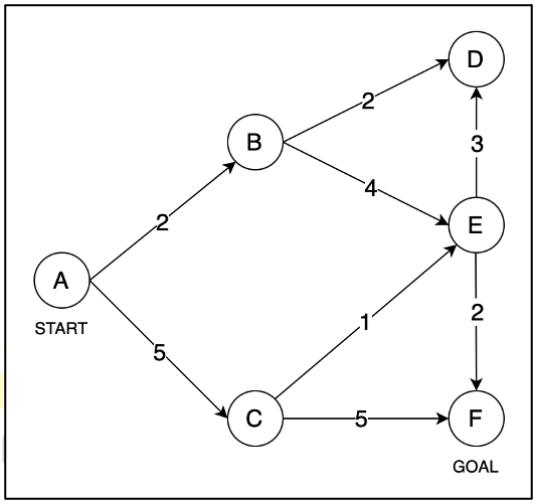


Figure 15. Example of directed graph for UCS.

Here is the implementation code for UCS on *directed graph* forFigure *15*:

Import *library* which is used	
[1]:	**import network as ns** # to create a graph **import matplotlib.pyplot as plt** # for graphical plot
Function to print the plot	

[2]:	# Support function for printing graphs
	def show_graph(G, pos=None, title='', labels={}) :
	  # Create a post if a post is not given
	  if pos is None:
	  pos = nx.spring_layout(G)
	
	  # Function to draw nodes
	  nx.draw(
	  G, # Graf NetworkX
	  pos, # Node positions
	  with_labels=True, # Displays the node name
	  labels=labels # Displays the label of each node
	  node_color='red', # Node color
	  node_size=2000, # Node size

```
        font_color="white", # Node label font color
        font_weight="bold", # Node label font thickness
        width=5             # Edge line thickness
    )

    # Takes edge labels if there is weight
    edge_labels = nx.get_edge_attributes(G, 'weight')
    # Function to draw nodes
    nx.draw_networkx_edge_labels(
        G,
        pos,
        edge_labels=edge_labels, # Data weight
        font_color='blue',       # Edge label font color
        font_weight="bold",      # Edge label font weight
        font_size=12,            # Edge label font size
    )

    plt.margins(0.2)  # Provides margin to the plot
    plt.title(title)  # Displays the graph title if given
    plt.show()        # Display graphs using matplotlib
```

Node positions `pos` **and labels *start* And *goal* to match the example image**

[3]:
```
pos = {
    'A': (0, 1),
    'B': (1, 2),
    'C': (1, 0),
    'D': (2, 3),
    'E': (2, 1.5),
    'F': (2, 0)
}
```

Make *directed graph* with NetworkX

[4]:
```
# Initialize an empty directed graph
G = nx.DiGraph()
```

[5]:
```
# Definition of nodes and edges according to the image
edges = [
    ('A', 'B', 2), ('A', 'C', 5),
    ('B', 'D', 2), ('B', 'E', 4),
    ('C', 'E', 1), ('C', 'F', 5),
    ('E', 'D', 3), ('E', 'F', 2)
]

# Add edges to the graph
G.add_weighted_edges_from(edges)
```

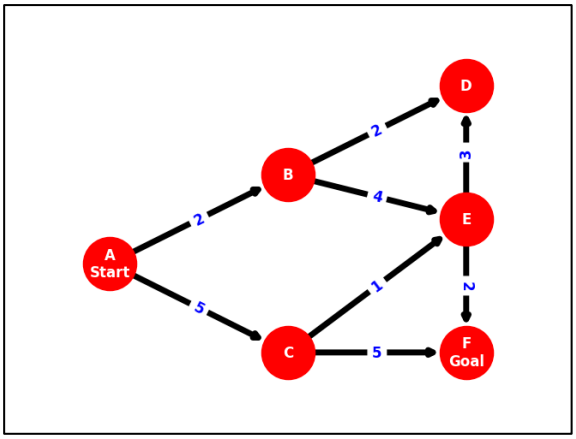
Evaluate elements in a graph

[6]:
```
print("Elements in Graph:")
print("List of Nodes in the Graph:", G.nodes())
print("List of Edges in Graph:", G.edges())
print("Number of Nodes in the Graph", G.number_of_nodes())
print("Number of Edges in Graph", G.number_of_edges())
```

```
Output:
Elements on the Graph:
List of Nodes in Graph: ['A', 'B', 'C', 'D', 'E', 'F']
List of edges in the graph: [('A', 'B'), ('A', 'C'), ('B', 'D'), ('B',
'E'), ('C', 'E'), ('C', 'F'), ('E', 'D'), ('E', 'F')]
Number of Nodes in Graph 6
```

	Number of Edges on Graph 8
Add labels *start* And *goal*	

[7]:	`# Records start and goal nodes`
	`labels = {node: node for node in G.nodes()}`
	`labels['A'] = 'A\nStart'`
	`labels['F'] = 'F\nGoal'`

Graph visualization	
[8]:	`# Prints the graph` `show_graph(G, pos=pos, labels=labels)`
	Output: 
Create a heuristic function	
[7]:	`def heuristable(a, b):` `  return 0`
Specifies the path from *node* And when *node* F with UCS	

[8]:	`# Search for visited nodes with the smallest weight from A to M`
	`path_node = nx.astar_path(G, 'A', 'F', heuristic=heuristable, weight='weight')`
	
	`# Find the smallest weight from A to M`
	`path_length = nx.astar_path_length(G, 'A', 'F', heuristic=heuristable, weight='weight')`
	
	`# Print results`
	`print("Node yang dikunjungi dari A ke M:", path_node)`
	`print("Besar weight terkecil dari A ke M:", path_length)`

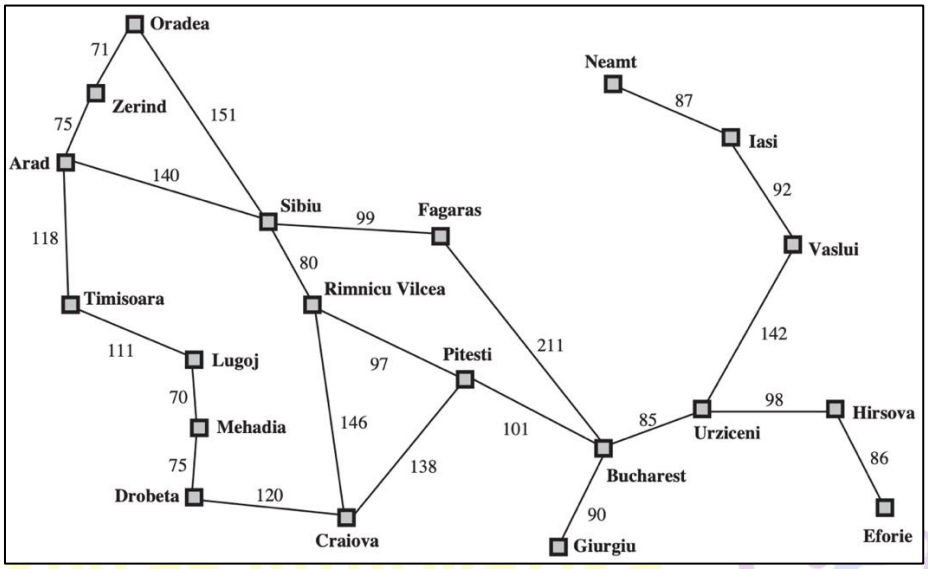
	Output: Nodes visited from A to M: ['A', 'B', 'E', 'F'] Smallest weight from A to M: 8

7.3. Exercise Questions

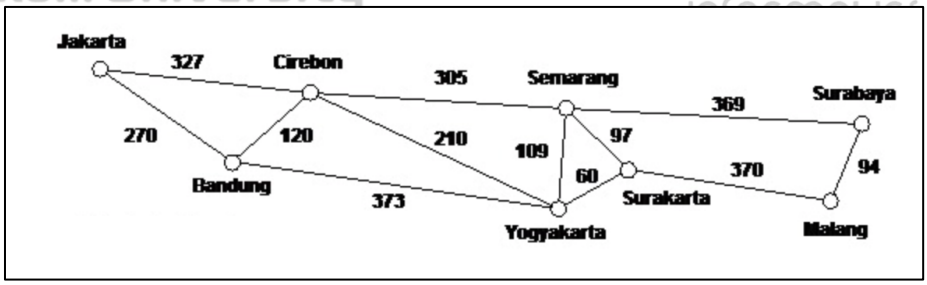
1. Create a weighted graph (*weighted graf*) from the following 2 graph images using the NetworkX library. Then do a Uniform Cost Search using the NetworkX library! Then check the results manually! For the European graph, the source city is Arad, and the destination city is Bucharest. For the Java Island graph, the source city is

Bandung, and the destination city is Malang. The returned value in the heuristic table is set to zero. Use a regular graph data type (not directed graph).

a. Map graph of major cities in Europe:



b. Map graph of major cities in Indonesia (Java island):



Modul 8 IMPLEMENTATION HILL CLIMBING

Practical Objectives:
1. Students understand and are able to implement algorithms *Hill Climbing* on Python.

Hill Climbing is one algorithm heuristic search which is used to find the best solution in the search space. The goal of this algorithm is to search for points in the search space that provide the highest or lowest value depending on the type of problem being solved.

The way this algorithm works can be analogized as follows, imagine there are climbers who are climbing a mountain in the dark. The climber doesn't have a map, but you always choose the path that looks the most uphill to reach the top. Algorithm hill climbing works on a similar principle. The climber begins the search from a random starting point, then continues moving in a direction that increases the value of a function (such as the height of a mountain) until it reaches a point where there is no further increase.

8.1. Implementation Example Hill Climbing

[*Traveling Salesman Problem*] One person salesman must visit 8 cities, where each city can only be visited once. The goal is to find a route with the minimum total distance from the distribution of cities in Figure 16.

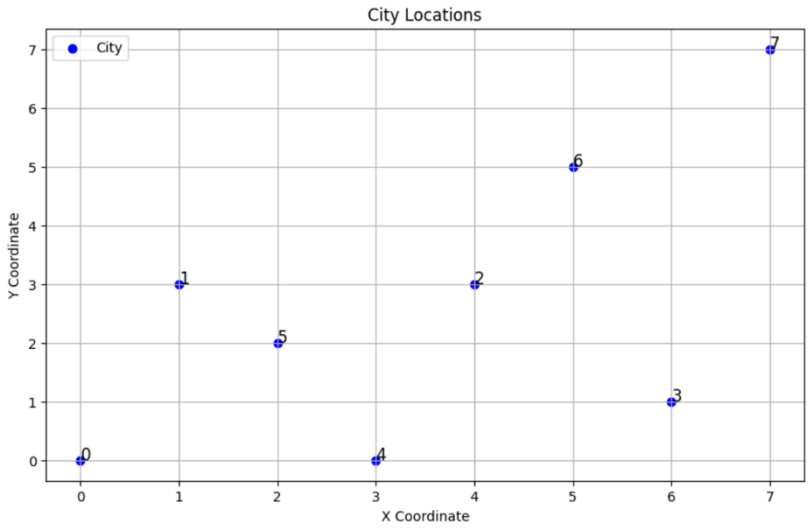


Figure 16. Graph of the distribution of cities that will be visited by salesmen in the Hill Climbing problem.

Following is the algorithm implementation code *Hill Climbing* to solve the problem:

Import *library* which is used

[1]:
```
import random
import math
```

Declaration *function* which will be used

[2]:
```
# Calculate the distance between 2 points using the Euclidean formula
def distance(city1, city2):
    return math.sqrt((city1[0] - city2[0]) ** 2 + (city1[1] - city2[1]) ** 2)

# Add up all the distances between points on the tour route
def total_distance(tour, cities):
    dist = 0
    for i in range(len(tour)):
        dist += distance(cities[tour[i]], cities[tour[(i + 1) % len(tour)]])
    return dist

# Provides all possible route combinations by swapping
# two dots in given tour route
def get_neighbors(tour):
    neighbors = []
    for i in range(len(tour)):
        for j in range(i + 1, len(tour)):
            neighbor = tour[:]
            neighbor[i], neighbor[j] = neighbor[j], neighbor[i]

            neighbors.append(neighbor)

    return neighbors
```

Hill Climbing Algorithm

[3]:
```
# Declaration of all city points (x, y) in tuple form into the list of cities
cities = [
    (0, 0), (1, 3), (4, 3), (6, 1),
    (3, 0), (2, 2), (5, 5), (7, 7)
]

current_solution = list(range(len(cities))) # initial city route
```

```
random.shuffle(current_solution)  randomize initial route using
function  random

# Sets the total distance for the initial route
current_distance = total_distance(current_solution, cities)

# The program will continue to loop until it does not find a distance value
# which is smaller than the current distance
while True:
    # Get all route combinations by swapping 2 points
    # on current solution
    neighbors = get_neighbors(current_solution)

    # Using the lambda function to find the minimum value
    # based on the total distance in the neighbors list
    best_neighbor = min(neighbors, key=lambda tour: total_distance(tour,
cities))

    # Calculate the total distance based on the best neighbors obtained
    best_distance = total_distance(best_neighbor, cities)

    # If there are no neighbors better, out of the loop
    if best_distance >= current_distance:
        break

    # Update current solutions with the best solutions from neighbors
    current_solution = best_neighbor
    current_distance = best_distance

print("Best solution (city order):", current_solution)
print("Total best distance:", current_distance)
```

```
Output:
Best solution (city order): [5, 2, 6, 7, 3, 4, 0, 1]
Best total distance: 24.12209449275384
```

Algorithm hill climbing does not always provide optimal solutions because it tends to get stuck on local optimum (the best solution in the vicinity), but not the best solution overall (global optimum). Therefore, for complex problems, these algorithms often do not find the most optimal solution without additional techniques such as simulated annealing or genetic algorithm.

8.2. Exercise Questions

One person salesman want to visit 5 different cities, and each city can only be visited once. Salesman must return to the starting city after visiting all the cities. The goal is to find the shortest route to visit all cities using an algorithm Hill Climbing with the distances listed in the Table *6*.

Table 6. Distance between cities that salesmen must visit.

Kota	A	B	C	D	E
A	0	10	15	20	25

B	10	0	35	25	15
C	15	35	0	30	20
D	20	25	30	0	10
E	25	15	20	10	0

1. Create an initial solution representation with the order of cities to be visited
2. Explain how to earn *neighbors* from the initial solution by swapping the two cities.
3. Use an algorithm *Hill Climbing* to find the shortest route. Give the process a change route from the initial solution to a better solution.
4. What is the shortest route found by the algorithm? Is this route the most optimal solution or is it still a problem? *local optimum*?

Modul 9 IMPLEMENTATION SIMULATED ANNEALING

Practical Objectives:
1. Students understand and are able to implement algorithms *Simulated Annealing* on Python.

Simulated Annealing is a technique used to find the best solution from various possibilities. This algorithm is inspired by processes in the world of physics called annealing, which is when a material is heated until very hot, then cooled slowly. This process allows the material to form a structure **strong and stable**. Algorithm Simulated Annealing works in a similar way to the concept annealing The algorithm starts by exploring various solutions, even if some of the solutions initially seem bad, then gradually becomes smarter at choosing better solutions as time goes on.

When compared with the algorithm Hill Climbing who always chooses the better solution at every step, Simulated Annealing more flexible. Simulated Annealing sometimes you can choose a worse solution, in order to explore more possible solutions. This is useful so as not to get stuck in local solutions (local optimum) which looks good but is actually not the best. So, Simulated Annealing more likely to find the best solution of all possibilities, not just the one that looks good around the current steps taken.

9.1. How Algorithms Work *Simulated Annealing*

Below is how the algorithm works *simulated annealing* in solving TSP problems:

1. **Initialize Initial Values:**
   The algorithm will produce an initial solution in the form of a random sequence of cities that need to be visited. For example, if we have cities 0 to 3, then the order will be randomized into an initial solution like `` `[3, 1, 0, 2]` ``.
2. **Initialize Temperature Parameters:**
   The algorithm will set the parameters **initial temperature** a large one, for example 10000. This value will decrease as the iteration progresses. How quickly the temperature drops will be regulated based on **cooling rate** (rate of temperature decrease). In addition to the initial temperature and cooling rate, the algorithm will determine **stopping temp** as the final iteration limit.
3. ***Loop* Based on Temperature:**
   The algorithm will go inside *loop* which will continue to run as long as the temperature is still above ***stopping temp*.** At each iteration, the temperature will

decrease slowly based on *cooling rate* by multiplying these two values (`temp *= cooling_rate`).

4. **Create a New Solution:**
   At each iteration, the algorithm will create a new solution with slight changes from the previous solution. This change can be done by: **swap two values randomly** in the previous solution.
5. **Evaluate New Solutions:**
   - If the new solution is better, the algorithm will immediately accept the new solution by doing *update* current solution value.
   - If the new solution is worse, the algorithm can accept the bad solution considering a certain probability based on the current temp value. The algorithm will calculate the probability using ***Metropolis Criterion*** which has the following equation:

$$P = e^{\frac{current\_distance - new\_distance}{temp}}$$

   If the random number is smaller than the P value, then a bad solution will be accepted.
6. ***Update Best Solution*:**
   In each iteration, the algorithm will compare the current solution with the best solution ever found. If the current solution is better, then the best solution will be replaced by the current solution.
7. **Final Result:**
   The algorithm will continue to lower the temperature based on *cooling rate*, the iteration will stop when the temperature falls below a predetermined limit, then the algorithm will provide the best solution found during the iteration as the optimal solution.

9.2. Implementation Example

[*Traveling Salesman Problem*] One person salesman must visit the 7 cities shown in the Figure 17, where each city can only be visited once. The goal is to find a route with the minimum total distance. Create a solution for the TSP case using the approach Simulated Annealing.

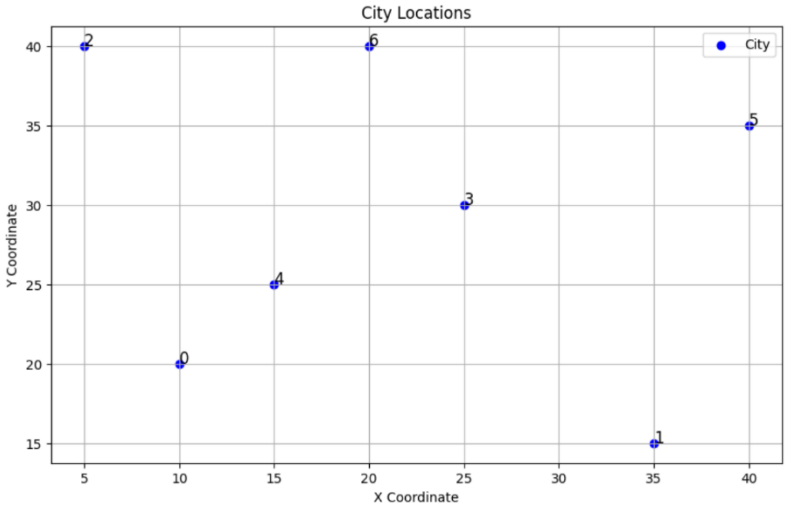


Figure 17. Graph of the distribution of cities that will be visited by salesmen in the example of Simulated Annealing questions.

Following is the algorithm implementation code *simulated annealing* to solve the problem:

Import *library* which is used

[1]:
```
import random
import math
```

Declaration *function* which will be used

[2]:
```
# Function to calculate distance of 2 points using the Euclidean
def distance(city1, city2):
    return math.sqrt((city1[0] - city2[0]) ** 2 + (city1[1] - city2[1]) ** 2)

# Function to add the entire distance between points on the tour route
def total_distance(tour, cities):
    dist = 0
    for i in range(len(tour)):
        dist += distance(cities[tour[i]], cities[tour[(i + 1) % len(tour)]])
    return dist

# Function to perform random swap of two points in the route
def random_swap(tour):
    new_tour = tour[:]
    i, j = random.sample(range(len(tour)), 2)
    new_tour[i], new_tour[j] = new_tour[j], new_tour[i]

    return new_tour
```

Simulated Annealing Algorithm

The algorithm will store 2 solution values. *Best solution* will save the best solution, and *current solution* will save the solution to do further exploration to avoid *local optimum*.

[3]:
```
def simulated_annealing(cities, initial_temp, cooling_rate, stopping_temp):
    # initialize the initial solution
    current_solution = list(range(len(cities))) # Rute awal (urut)
    random.shuffle(current_solution)  # Mengacak rute awal

    # Calculates the total distance for the initial route
    current_distance = total_distance(current_solution, cities)
```

```
    # Set best solution and best distance based on current solution
    best_solution = current_solution[:]
    best_distance = current_distance

    # Set initial temperature based on the parameters received by the function
    temp = initial_temp

    # The program will continue to loop until the temperature is less than the
stopping temperature
    while temp > stopping_temp:

        # Create a new solution by swapping two random points
        new_solution = random_swap(current_solution)
        new_distance = total_distance(new_solution, cities)

        # If the new solution is better, accept it
        if new_distance < current_distance:
            current_solution = new_solution[:]
            current_distance = new_distance
        else:
            # if the new solution is worse, accept it with some probability
            if random.random() < math.exp((current_distance -
new_distance)/temp):
                current_solution = new_solution[:]
                current_distance = new_distance

        # Save the best solution found
        if current_distance < best_distance:
            best_solution = current_solution[:]
            best_distance = current_distance

        # Reduce the temperature according to the cooling rate
        temp *= cooling_rate

    return best_solution, best_distance
```

[4]:

```
# Declaration of city points (x, y)
cities = [
    (0, 0), (1, 3), (4, 3), (6, 1),
    (3, 0), (2, 2), (5, 5), (7, 7)
]

# Algorithm parameters
init = 10000     # Initial temperature
Cooling = 0.995  # Cooling rate
Stop = 1         # Temperature when stopped

# Run the Simulated Annealing algorithm
best_solution, best_distance = simulated_annealing(cities, init,
cooling, stop)

print("Best solution (city order):", best_solution)
print("Total best distance:", best_distance)
```

```
Output:
Best solution (city order): [6, 2, 0, 4, 1, 5, 3]
Best total distance: 112.65453203138081
```

Importance Notes!

`` `[if random.random() < math.exp((current_distance - new_distance)/temp)]` ``

This stage is the core of the algorithm *Simulated Annealing* where this stage allows the algorithm to accept worse solutions based on probability ***Metropolis Criterion***. This aims to avoid *local optimum* and allows broader exploration of solutions.

- If the new solution is worse, the algorithm will accept that solution with probability ***Metropolis Criterion***
- `` `random.random()` `` will generate a random number between 0 and 1
- Whereas `` `math.exp((current_distance - new_distance) / temp`) `` is a probability equation known as ***Metropolis Criterion*** in *Simulated Annealing*. This equation will determine how likely a bad solution is to be accepted based on *temp*.

Algorithm *Simulated Annealing* used to find solutions *global optimum* by exploring possible solutions and avoiding pitfalls *local optimum*, different from algorithms *Hill Climbing* who tend to get stuck on the local best solution because they only accept immediate improvements without exploring worse solutions. With probability controlled by temperature (*temperature*), *Simulated Annealing* allows acceptance of a worse solution at the start of the for iteration **expand the search space**, but gradually becomes more selective as the value of temperature decreases (*cooling process*), thereby increasing the chances of achieving better global solutions.

9.3. Exercise Questions

[*Multiple Traveling Saleman Problem*] There are two people *salesman* who want to visit 8 cities. Each salesman was only allowed to visit each city once, and no city was visited more than once by a second *salesman*. The coordinates of the 8 cities have been determined based on the points on.

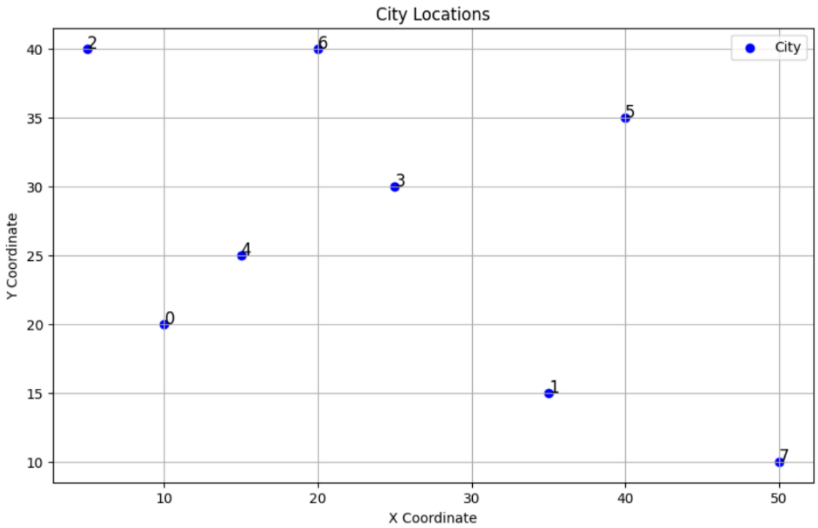


Figure 18. Graph of the distribution of cities that will be visited by salesmen in the Simulated Annealing practice questions.

Create a solution to determine the second travel route *salesman* using an algorithmic approach *Simulated Annealing*. The resulting route must minimize the total distance traveled by both *salesman* together.

Modul 10 IMPLEMENTATION GREEDY BEST-FIRST SEARCH

Practical Objectives:
1. Students understand and are able to implement algorithms *Greedy Best-First Search* in Python by using *library* NetworkX.

REFERENCE:
https://networkx.org/documentation/stable/reference/algorithms/shortest_paths.html#module-networkx.algorithms.shortest_paths.astar

Greedy Best-First Search (Greedy BFS) is a search algorithm that prioritizes the exploration of nodes that are considered the most promising to reach the goal as quickly as possible based on a heuristic function. Greedy BFS only considers the heuristic value of node now on to the destination. This algorithm uses an approximation greedy, where each step, node with the smallest heuristic value is explored first. However, Greedy BFS always finds the optimal path because it only focuses on the lagnsugn approach without considering total costs.

NetworkX, *Greedy* BFS can be implemented by modifying the function `` `nx.astar_path()` `` y assigning a weight value to each *edge* becomes 0, so only the heuristic function is used to guide the search. However, in this Greedy BFS implementation, `` `nx.astar_path_length()` `` cannot be used because each weight on the edge has a value of 0.

10.1. Implementation *Greedy* BFS on *Directed Graph*

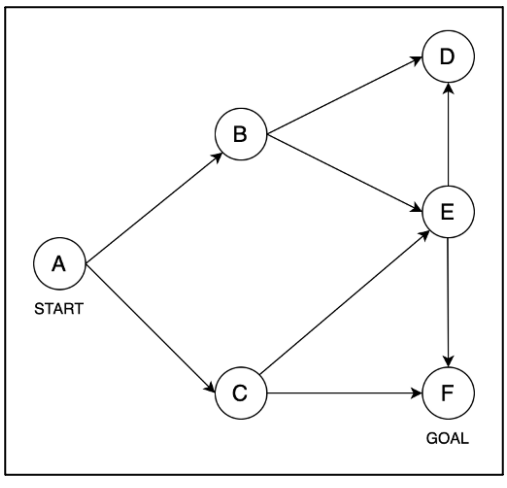


Figure 19. Example directed graph for Greedy BFS.

Table 7. Heuristic values for directed graph Greedy BFS.

Node	Nilai Heuristik
A	10
B	2
C	3
D	1
E	4
F	0

Here is the implementation code for *Greedy BFS* on *directed graph* for Figure *19* with the heuristic values listed in Table *7*:

Import *library* which is used

[1]:
```
import network as ns # to create a graph
import matplotlib.pyplot as plt # for graphical plot
```

Function to print the plot

[2]:
```
# Support function for printing graphs
def show_graph(G, pos=None, title='', labels={}) :
  # Create a post if a post is not given
  if pos is None:
      pos = nx.spring_layout(G)

  # Function to draw nodes
  nx.draw(
      G,                  # Graf NetworkX
      pos,                # Node position
      with_labels=True,   # Displays the node name
      labels=labels       # Displays the label of each node
      node_color='red',   # Node color
      node_size=2000,     # Node sizes
      font_color="white", # Node label font color
      font_weight="bold", # Node label font thickness
      width=5             # Edge line thickness
  )

  # Takes edge labels if there is weight
  edge_labels = nx.get_edge_attributes(G, 'weight')
  # Function to draw nodes
  nx.draw_networkx_edge_labels(
      G,
      pos,
      edge_labels=edge_labels, # Data weight
```

```
        font_color='blue',        # Edge label font color
        font_weight="bold",       # Edge label font weight
        font_size=12,             # Edge label font size
    )

    plt.margins(0.2)  # Provides margin to the plot
    plt.title(title)  # Displays the graph title if given
    plt.show()        # Display graphs using matplotlib
```

Position of nodes `pos` and labels *start* And *goal* to match the example image

[3]:
```
pos = {
    'A': (0, 1),
    'B': (1, 2),
    'C': (1, 0),
    'D': (2, 3),
    'E': (2, 1.5),
    'F': (2, 0)
}
```

Make *directed graph* with NetworkX

[4]:
```
# Initialize an empty directed graph
G = nx.DiGraph()
```

[5]:
```
# Definition of nodes and edges according to the image
edges = [
    ('A', 'B', 0), ('A', 'C', 0),
    ('B', 'D', 0), ('B', 'E', 0),
    ('C', 'E', 0), ('C', 'F', 0),
    ('E', 'D', 0), ('E', 'F', 0)
]

# Add edges to the graph
G.add_weighted_edges_from(edges)
```

Evaluate elements in a graph

[6]:
```
print("Elements in Graph:")
print("List of Nodes in the Graph:", G.nodes())
print("List of Edges in Graph:", G.edges())
print("Number of Nodes in the Graph", G.number_of_nodes())
print("Number of Edges in Graph", G.number_of_edges())
```

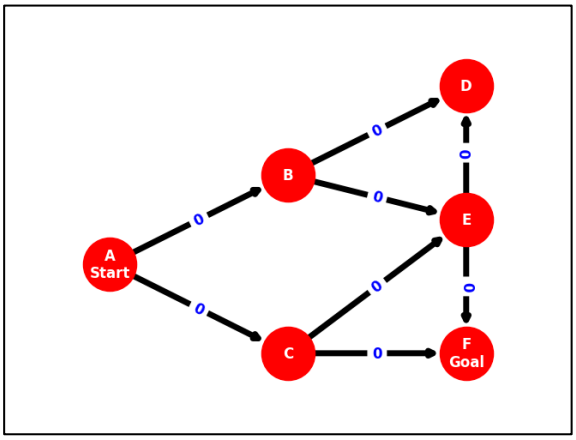
```
Output:
Elements on the Graph:
List of Nodes in Graph: ['A', 'B', 'C', 'D', 'E', 'F']
List of edges in the graph: [('A', 'B'), ('A', 'C'), ('B', 'D'), ('B',
'E'), ('C', 'E'), ('C', 'F'), ('E', 'D'), ('E', 'F')]
Number of Nodes in Graph 6
Number of Edges on Graph 8
```

Add labels *start* And *goal*

[7]:
```
# Records start and goal nodes
labels = {node: node for node in G.nodes()}
labels['A'] = 'A\nStart'
labels['F'] = 'F\nGoal'
```

Graph visualization

[8]:
```
# Prints the graph
show_graph(G, pos=pos, labels=labels)
```

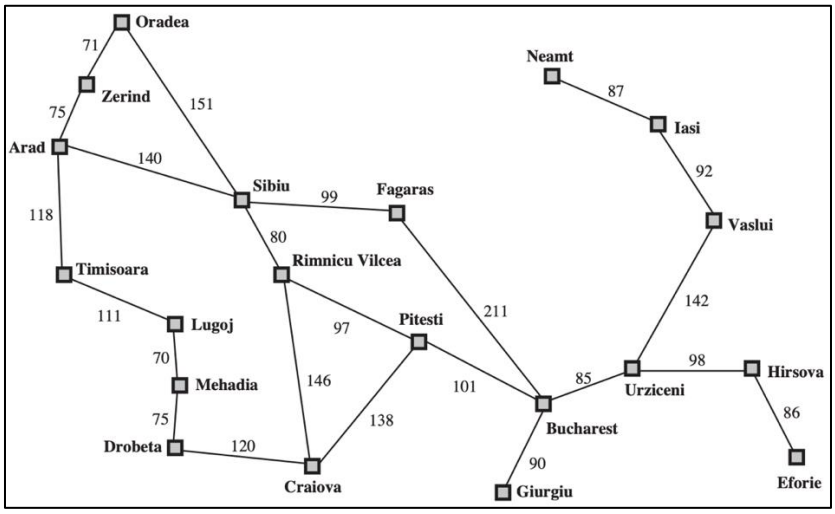
	Output: 
Create a heuristic function	
[7]:	**def heuristable(a, b):** ** if a == 'A' :** ** return 10** ** elif a == 'B' :** ** return 2** ** elif a == 'C' :** ** return 3** ** elif a == 'D' :** ** return 1** ** elif a == 'E' :** ** return 4** ** elif a == 'F' :** ** return 0**
Specifies the path from *node* And when *node* F with UCS	
[8]:	# Search for visited nodes with the smallest weight from A to F **path_node = nx.astar_path(G, 'A', 'F', heuristic=heuristable, weight='weight')** # Print results **print("Node yang dikunjungi dari A ke F:", path_node)**
	Output: Nodes visited from A to F: ['A', 'C', 'F']

10.2. Exercise Questions

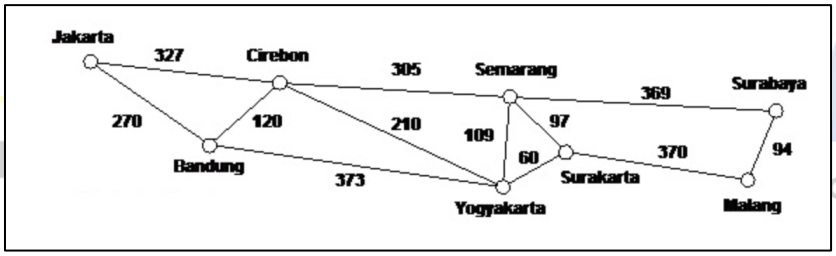
1. Create a weighted graph (*weighted graf*) from the following 2 graph images using the NetworkX library. Then do the search *Greedy Best-First Search* use *library* NetworkX! Don't forget to delete data *weight*first! For European graphs, the city of origin (*source*) is Arad, and the destination city (*goal*) is Bucharest. For the Java Island graph, the city of origin (*source*) is Bandung, and the destination city (*goal*) is Malang. To determine value *returned value* in the heuristic table, use the formula *Euclidan Distance* to calculate the distance between two points (i.e. the length of a

straight line connecting the two points). Use regular graph data types (not *directed graph*).

a. Map graph of major cities in Europe:



b. Map graph of major cities in Indonesia (Java island):



Modul 11 IMPLEMENTATION A*

Practical Objectives:
1. Students understand and are able to implement the A* algorithm in Python using *library* NetworkX.

REFERENCE:
https://networkx.org/documentation/stable/reference/algorithms/shortest_paths.html#module-networkx.algorithms.shortest_paths.astar

A* is a path finding algorithm used to find the shortest path from *node* early (*start*) the *node* objective (*goal*) in the graph. A* combines from UCS and *Greedy* BFS taking into account actual costs ($g(n)$) and the route that has been traversed, as well as the estimated cost ($h(n)$) from *node* currently to *node* goals based on heursitic functions.

A* uses a function $f(n) = g(n) + h(n)$ to prioritize *node* to be explored, where $g(n)$ is the cost from the starting node to the current node, and $h(\ )$ is the estimated cost from the current node to the destination. With this approach, A* ensures that the path found is the optimal path.

On NetworkX, A* can be implemented using functions `` `nx.astar_path()` ``, which automatically calculates the shortest path taking into account true costs and heuristic functions.

11.1. Implementation of A* On *Directed Graph*

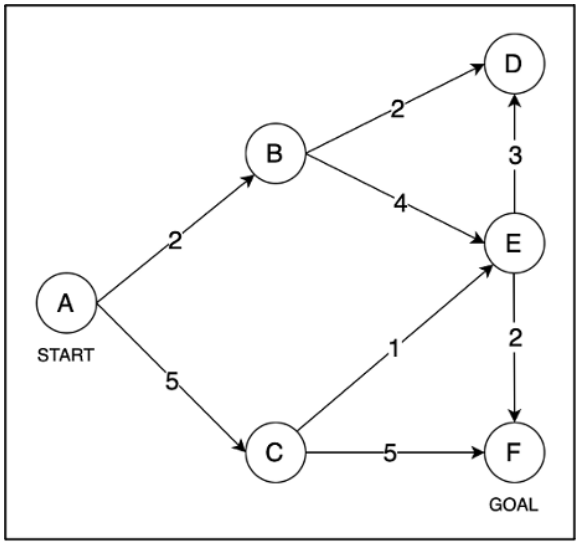


Figure 20. Example directed graph for A*.

Table 8. Heuristic values for directed graph A*.

Node	Nilai heuristik
A	10
B	2
C	3
D	1
E	4
F	0

Here is the implementation code for A* on *directed graph* for Figures *20* with the heuristic values listed in Table *8*:

Import *library* which is used

[1]:
```
import network as ns # to create a graph
import matplotlib.pyplot as plt # for graphical plot
```

Function to print the plot

[2]:
```
# Support function for printing graphs
def show_graph(G, pos=None, title='', labels={}) :
  # Create a post if a post is not given
  if pos is None:
      pos = nx.spring_layout(G)

  # Function to draw nodes
  nx.draw(
      G,                  # Graf NetworkX
      pos,                # Node position
      with_labels=True,   # Displays the node name
```

```
        labels=labels        # Displays the label of each node
        node_color='red',    # Node color
        node_size=2000,      # Node size
        font_color="white", # Node label font color
        font_weight="bold", # Node label font thickness
        width=5              # Edge line thickness
    )

    # Takes edge labels if there is weight
    edge_labels = nx.get_edge_attributes(G, 'weight')
    # Function to draw nodes
    nx.draw_networkx_edge_labels(
        G,
        pos,
        edge_labels=edge_labels, # Data weight
        font_color='blue',       # Edge label font color
        font_weight="bold",      # Edge label font weight
        font_size=12,            # Edge label font size
    )

    plt.margins(0.2)  # Provides margin to the plot
    plt.title(title)  # Displays the graph title if given
    plt.show()        # Display graphs using matplotlib
```

Position of nodes `pos` and start and goal labels to match the example image

[3]:

```
pos = {
    'A': (0, 1),
    'B': (1, 2),
    'C': (1, 0),
    'D': (2, 3),
    'E': (2, 1.5),
    'F': (2, 0)
}
```

Make *directed graph* with NetworkX

[4]:

```
# Initialize an empty directed graph
G = nx.DiGraph()
```

[5]:

```
# Definition of nodes and edges according to the image
edges = [
    ('A', 'B', 2), ('A', 'C', 5),
    ('B', 'D', 2), ('B', 'E', 4),
    ('C', 'E', 1), ('C', 'F', 5),
    ('E', 'D', 3), ('E', 'F', 2)
]

# Add edges to the graph
G.add_weighted_edges_from(edges)
```

Evaluate elements in a graph

[6]:

```
print("Elements in Graph:")
print("List of Nodes in the Graph:", G.nodes())
print("List of Edges in Graph:", G.edges())
print("Number of Nodes in the Graph", G.number_of_nodes())
print("Number of Edges in Graph", G.number_of_edges())
```

```
Output:
Elements on the Graph:
List of Nodes in Graph: ['A', 'B', 'C', 'D', 'E', 'F']
```

```
List of edges in the graph: [('A', 'B'), ('A', 'C'), ('B', 'D'), ('B',
'E'), ('C', 'E'), ('C', 'F'), ('E', 'D'), ('E', 'F')]
Number of Nodes in Graph 6
Number of Edges on Graph 8
```

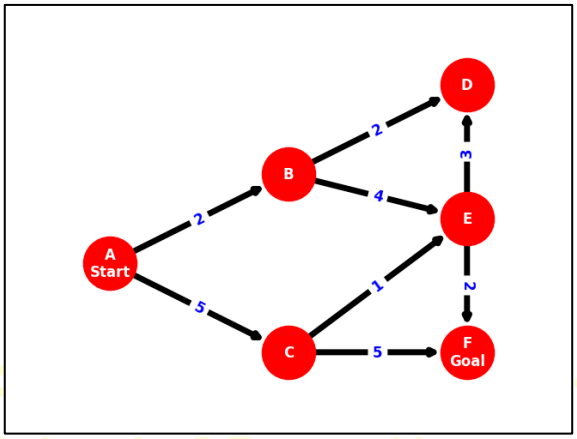
Add labels *start* And *goal*

[7]:
```
# Records start and goal nodes
labels = {node: node for node in G.nodes()}
labels['A'] = 'A\nStart'
labels['F'] = 'F\nGoal'
```

Graph visualization

[8]:
```
# Prints the graph
show_graph(G, pos=pos, labels=labels)
```

Output:



Create a heuristic function

[7]:
```
def heuristable(a, b):
  if a == 'A' :
    return 10
  elif a == 'B' :
    return 2
  elif a == 'C' :
    return 3
  elif a == 'D' :
    return 1
  elif a == 'E' :
    return 4
  elif a == 'F' :
    return 0
```

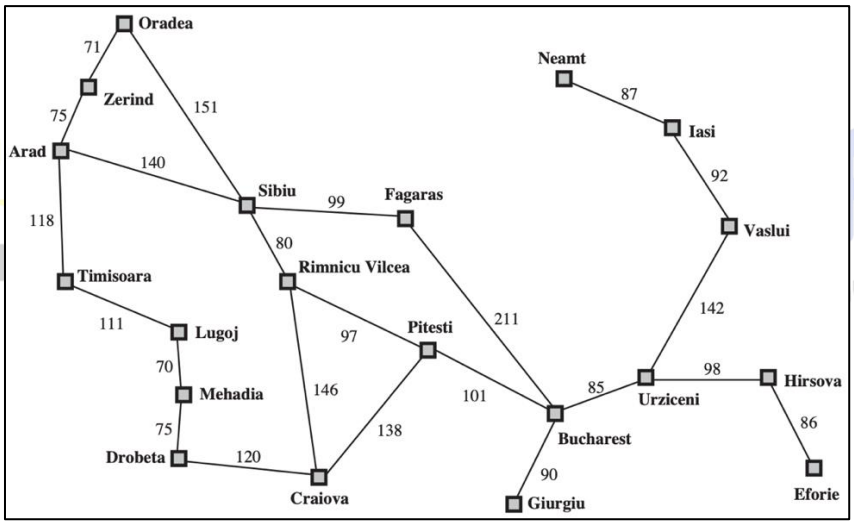
Specifies the path from *node* And when *node* F with UCS

[8]:
```
# Search for visited nodes with the smallest weight from A to F
path_node = nx.astar_path(G, 'A', 'F', heuristic=heuristable,
weight='weight')

# Find the smallest weight from A to F
path_length = nx.astar_path_length(G, 'A', 'F', heuristic=heuristable,
weight='weight')

# Print results
print("Node yang dikunjungi dari A ke F:", path_node)
print("Besar weighth hasil A* dari A ke F:", path_length)
```

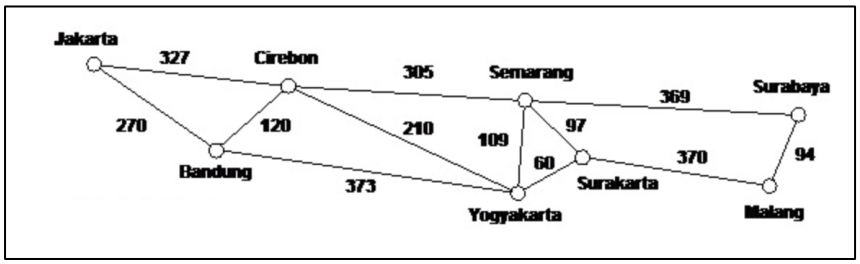
	Output: Nodes visited from A to F: ['A', 'B', 'E', 'F'] Large weight of A* results from A to F: 8

11.2. Exercise Questions

1. Create a weighted graph (*weighted graf*) from the following 2 graphs using *library* NetworkX. Then perform an A* search using *library* NetworkX! For European graphs, the city of origin (*source*) is Arad, and the destination city (*goal*) is Bucharest. For the Java Island graph, the city of origin (*source*) is Bandung, and the destination city (*goal*) is Malang. To determine the returned value in the heuristic table, use the formula *Euclidan Distance* to calculate the distance between two points (i.e. the length of the straight line connecting the two points). Use regular graph data types (not *directed graph*). Also remember that both weight values and heuristic table values are used in the A* algorithm calculations.
   a. Map graph of major cities in Europe:



   b. Map graph of major cities in Indonesia (Java island):



Modul 12 INTRODUCTION *LIBRARY* SYMPHY

Practical Objectives:
1. Students understand how to create programs using *library* Sympy. 2. Students know the main features available in *library* Sympy.

REFERENSI UTAMA:
https://docs.sympy.org/latest/tutorials/intro-tutorial/index.html
https://certik.github.io/scipy-2013-tutorial/html/index.html
REFERENSI TAMBAHAN:
https://docs.sympy.org/latest/guides/index.html
https://docs.sympy.org/latest/reference/index.html

SymPy is *library* Python is used to perform symbolic calculations in mathematics. With SymPy, mathematical expressions can be represented, simplified, evaluated, and manipulated symbolically, not just numerically.

The goal of SymPy is to provide tools that allow mathematical calculations to be performed with high symbolic accuracy, so that calculations such as algebra, calculus, and matrix operations can be solved analytically.

12.1. Method Import SymPy

SymPy should already be present in your Anaconda installation. To use the SymPy module, it must be done *import* first. Here is the standard way to import *library* Sympy:

```
import sympy as sp
```

12.2. Defining Symbols in SymP

SymPy allows mathematical calculations to be performed symbolically, meaning that variables and mathematical expressions can be defined and manipulated as algebraic symbols. With symbols, various mathematical calculations and manipulations can be carried out symbolically. To do this, the first step is to define the symbol using `` `symbols()` ``. Here are some examples in use `` `symbols()` ``:

[1]:	`# Use symbols() to define a single symbol` **`x = sp.symbols('x')`** **`x`**
	Output: x
[2]:	`# Definition of some symbols` `# Method 1`

	a, b, c = sp.symbols("a b c") # Method 2 **a, b, c = sp.symbols("a, b, c")** **a + b + c**
	Output: $a + b + c$
[3]:	# Defining symbols with lists **my_symbols = ['a', 'b', 'c']** **a, b, c = sp.symbols(my_symbols)** **a + b + c**
	Output: $a + b + c$
[4]:	# Definition of variable symbol with index **x1, x2, x3 = sp.symbols("x1, x2, x3")** **x1 + x2 + x3**
	Output: $x_1 + x_2 + x_3$
[5]:	# Defining symbols with range notation **x_n = sp.symbols('x:5')** **x_n[0] + x_n[1] + x_n[2] + x_n[3] + x_n[4]**
	Output: $x_1 + x_2 + x_3 + x_4 + x_5$
[6]:	# Definition of Greek symbols **alpha, beta, gamma = sp.symbols("alpha, beta, gamma")** **alpha + beta + gamma**
	Output: $\alpha + \beta + \gamma$

12.3. Basic Algebra in SymPy

SymPy makes it possible to perform basic algebraic operations symbolically, meaning variables and mathematical expressions can be manipulated as in algebra. SymPy provides tools for addition, subtraction, multiplication, division, expansion, and simplification of mathematical expressions. Here is a basic algebra implementation in SymPy:

[1]:	# Addition **x, y = sp.symbols('x, y')** **addition = x + y** **addition**
	Output: $x + y$
[2]:	# Reduction **x, y = sp.symbols('x, y')** **substraction = x - y** **substraction**
	Output: $x - y$
[3]:	# Multiplication **x, y = sp.symbols('x, y')** **multiplication = x * y** **multiplication**

	Output: xy
[4]:	# Distribution **x, y = sp.symbols('x, y')** **division = x / y** **division**
	Output: $\frac{x}{y}$
`expand()` This method is used to develop symbolic expressions, which decompose expressions into longer forms.	
[1]:	**x, y = sp.symbols('x, y')** **expr = (x + y) ** 2** **expr**
	Output: $(x+y)^2$
[2]:	# Develop expression **expanded_expr = expr.expand()** **expanded_expr**
	Output: x^2+xy+y^2
`.simplify()` This method is used to simplify symbolic expressions to their simplest form. This method may involve combining terms, factoring, trigonometric simplification, etc. to find the simplest expression.	
[1]:	**x = sp.symbols('x')** **expr = sp.sin(x)**2 + sp.cos(x)**2** **expr**
	Output: $\sin^2(x)+\cos^2(x)$
[2]:	# Simplifies expressions **simplified_expr = expr.simplify()** **simplified_expr**
	Output: **1**

12.4. Basic Calculus in SymPy

SymPy can also be used to perform basic calculus calculations symbolically. This includes differentiation, integration, and limit calculations. With SymPy, these operations can be performed on mathematical expressions analytically, allowing symbolic manipulation and evaluation without the need for direct conversion to numerical form. The following is an example of implementing basic calculus in SymPy:

`.diff()` This method is used to differentiate or calculate the derivative of a symbolic function for certain variables.	
[1]:	**x = sp.symbols('x')** # Definition of the function f(x) **f = x**3 + 2*x**2 + x** **f**
	Output:

	$x^3 + 2x^2 + x$
[2]:	`# Differentiation of the function f(x) with respect to x` **`f_prime = f.diff(x)`** **`f_prime`**
	Output: $3x^2 + 4x + 1$

`.integrate()`

This method is used to integrate a symbolic function. Integration can be carried out indefinitely (integral without limits) or definitely (integral with limits)

[1]:	**`x = sp.symbols('x')`** `# Definition of the function f(x)` **`f = sp.sin(x)`**
	Output: $\sin(x)$
[2]:	`# Indeterminate integration` **`f_integral_indef = sp.integrate(f, x)`** **`f_integral_indef`**
	Output: $-\cos(x)$

[3]:	`# Integration of course from 0 to pi`
	`f_integral_def = sp.integrate(f, (x, 0, sp.pi))`
	`# .integrate(function, (expression, lower_limit, upper_limit))`
	`f_integral_def`

	Output: 2

`.limit()`

This method is used to calculate the limit of a function when the variable approaches a certain value.

[1]:	**`x = sp.symbols('x')`** `# Definition of the function f(x)` **`f = sp.sin(x) / x`** **`f`**
	Output: $\frac{\sin(x)}{x}$
[2]:	`# Calculates the limit of f(x) as x approaches 0` **`f_limit = sp.limit(f, x, 0)`** **`f_limit`**
	Output: 1

12.5. Symbolic Equation Solving On SymPy

Sympy provides powerful tools for solving symbolic equations, both algebraic equations and differential equations. Two main methods are used in SymPy for symbolic solutions is `.solve()` dan `.dsolve()`. The following is an implementation and example of solving symbolic equations:

`.eq()`

This function is used to define a symbolic equation in SymPy. This function is used to state that two mathematical expressions are equal.

[1]:	**`x = sp.symbols('x')`**

	`# Create equations` **`equation = sp.Eq(x**2 - 4, 0)`** **`equation`**
	Output: $x^2 - 4 = 0$
[2]:	**`x, y = sp.symbols('x, y')`** `# Create equations` **`equation = sp.Eq(x**3 - 2, y - 2 + 3*x)`** **`equation`**
	Output: $x^3 - 2 = 3x + y - 2$

`.solve()`

It is a function used to find solutions to equations or systems of symbolic equations. This function solves algebraic equations by finding symbolic values that satisfy the equation.

[1]:	**`x = sp.symbols('x')`** `# Create equations` **`equation = sp.Eq(x**2 - 4, 0)`** **`equation`**
	Output: $x^2 - 4 = 0$
[2]:	`# Solve the equation for x` **`solution = sp.solve(equation, x)`** **`solution`**
	Output: $[-2, 2]$

`.dsolve()`

It is a function used to solve symbolic differential equations. This function finds the general solution of the differential equation in symbolic form.

[1]:	`# Definition of symbols and functions` **`t = sp.symbols('t')`** **`y = sp.Function('y')(t)`** **`equation = sp.Eq(y(t).diff(t,t) + y(t), 0)`** **`equation`**
	Output: $\frac{d}{dt}y(t) = -t + y(t)$
[2]:	`# Solving differential equations` **`solution = sp.dsolve(equation, y(t))`** **`solution`**
	Output: $y(t) = C_1 e^t + t + 1$

12.6. Matrix Manipulation in SymPy

SymPy provides a variety of tools for matrix manipulation, including matrix definition, determinant operations, inverse operations, and matrix multiplication. The following is an implementation and example of matrix manipulation in SymPy:

`.Matrix()`

This function is used to define a matrix that can be filled with numbers, symbolic variables, or symbolic expressions.

[1]:	`# Definition of 2x2 matrix` **`A = sp.Matrix([[1, 2], [3,4]])`** **`A`**
	Output: $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$
[2]:	`# Definition of 3x3 zero matrix` **`A = sp.zeros(3)`** **`A`**
	Output: $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$
[3]:	`# Definition of 3x3 identity matrix` **`A = sp.eye(3)`** **`A`**
	Output: $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
`.det()` This function is used to calculate the determinant of a square matrix.	
[1]:	`# Matrix Definition` **`A = sp.Matrix([[1, 2], [3, 4]])`** **`A`**
	Output: $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$
[2]:	`# Calculate the determinant of matrix A` **`det_A = A.det()`** **`det_A`**
	Output: -2
`.inv()` Fungsi ini digunakan untuk menghitung invers dari sebuah matriks.	
[1]:	`# Matrix Definition` **`A = sp.Matrix([[1, 2], [3, 4]])`** **`A`**
	Output: $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$
[2]:	`# Solving differential equations` **`solution = sp.dsolve(equation, y(t))`** **`solution`**
	Output: $\begin{bmatrix} -2 & 1 \\ \frac{3}{2} & -\frac{1}{2} \end{bmatrix}$
Matrix Multiplication	

Matrix multiplication is done with `*`, produces a new matrix which is the result of multiplying two matrices.	

[1]:	`# Matrix Definition`
	`A = sp.Matrix([[1, 2], [3, 4]])`
	`B = sp.Matrix([[5, 6], [7, 8]])`
	
	`# Multiplication of matrices A and B`
	`mat_mul = A * B`
	`mat_mul`

	Output: $\begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$

12.7. Exercise Questions *Library* SymPy

A. By using *library* Sympy, find the roots of the following equations (if the roots are imaginary or do not exist, write down what output appears on the screen):

1. $x - 2 = 0$
2. $x^2 - 3x - 4 = 0$
3. $x^2 - 6x + 9 = 0$
4. $x^2 - x - 1 = 0$
5. $x^3 - 6x^2 + 11x - 6 = 0$
6. $x^3 - 7x^2 + 15x - 9 = 0$
7. $x^5 + 7x^4 - 9x^2 + 15x + 2 = 0$
8. $x^5 + 9x^2 + 2 = 0$

B. Using the Sympy library, find the first derivative or $\frac{dy}{dx}$ from:

1. $y = x^3 + 3x^2 + 6x$
2. $y = x^5 + x^4$
3. $y = (3x + 5)^3$
4. $y = (3 - 5x)^5$
5. $y = \sin(7x)$
6. $y = \sin(x^3)$
7. $y = \dfrac{1}{x - 1}$
8. $y = \dfrac{3x}{1 - x}$

C. Using the Sympy library, find the definite integral (*definite integral*) from:

1. $\int_0^2 x^3\,dx$	**2.** $\int_{-1}^2 x^4\,dx$
3. $\int_{-1}^2 (3x^2 - 2x + 3)\,dx$	**4.** $\int_1^2 (4x^3 + 7)\,dx$
5. $\int_1^4 \frac{1}{w^2}\,dw$	**6.** $\int_1^3 \frac{2}{t^3}\,dt$
7. $\int_0^4 \sqrt{t}\,dt$	**8.** $\int_1^8 \sqrt[3]{w}\,dw$
9. $\int_{-4}^{-2} \left(y^2 + \frac{1}{y^3}\right) dy$	**10.** $\int_1^4 \frac{s^4 - 8}{s^2}\,ds$
11. $\int_0^{\pi/2} \cos x\,dx$	**12.** $\int_{\pi/6}^{\pi/2} 2\sin t\,dt$
13. $\int_0^1 (2x^4 - 3x^2 + 5)\,dx$	**14.** $\int_0^1 (x^{4/3} - 2x^{1/3})\,dx$

D. Given matrix A as follows. By using *library* Sympy, find the determinant of matrix A.

$$A = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 3 & 1 & 2 & 2 \\ 1 & 0 & -2 & 1 \\ 2 & 0 & 0 & 1 \end{bmatrix}$$

Modul 13 INTRODUCTION *LIBRARY* SCIKIT-FUZZY

Practical Objectives:
1. Students understand how to create programs using *library* Scikit-Fuzzy. 2. Students know the main features available in *library* Scikit-Fuzzy.

REFERENSI UTAMA:
https://scikit-fuzzy.github.io/scikit-fuzzy/userguide/getting_started.html
https://scikit-fuzzy.github.io/scikit-fuzzy/auto_examples/plot_tipping_problem_newapi.html#example-plot-tipping-problem-newapi-py
REFERENSI TAMBAHAN:
https://scikit-fuzzy.github.io/scikit-fuzzy/user_guide.html
https://scikit-fuzzy.github.io/scikit-fuzzy/auto_examples/index.html
https://github.com/scikit-fuzzy/scikit-fuzzy
https://pypi.org/project/scikit-fuzzy/

Fuzzy Logic is a computational approach that mimics the way humans make decisions based on uncertain or ambiguous information. Unlike traditional logic which only recognizes the values 0 (false) and 1 (true), fuzzy logic introduced the concept of degrees of truth, where values can be between 0 and 1, reflecting degrees of certainty and uncertainty.

In *fuzzy logic*, a condition need not be completely true or false, for example a temperature can be categorized as "warm" by a certain degree, not just "hot" or "cold". This allows the system to handle uncertainties and variations more flexibly.

In Python, calculations *fuzzy logic* can be done easily using *library* scikit-fuzzy. *Library* this makes it easier to build, test, and deploy the system *fuzzy logic* by defining variables *fuzzy*, membership functions, rules *fuzzy*, as well as carrying out fuzzification, inference and defuzzification processes.

13.1. Scikit-Fuzzy Installation

Scikit-Fuzzy is not available on *package* Anaconda automatically *default*. Here are the steps to add *package* Scikit-Fuzzy on *environment* Anaconda:

1. Open **Anaconda Prompt** for Windows or **Terminal** on MacOS.
2. Run the command `` `pip install scikit-fuzzy` ``.

3. If successful, you will see a display like Figure *21*.

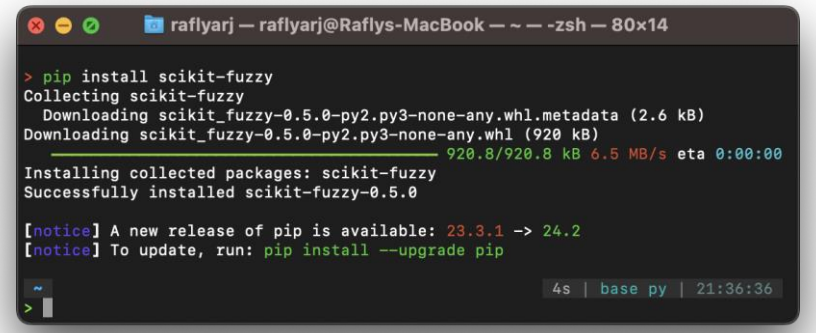


Figure 21. Terminal display when Scikit-Fuzzy installation is successful.

13.2. Create Programs with *Library* Scikit-Fuzzy

13.2.1. Variable Initialization *Fuzzy*

In fuzzy logic, there are two variables that define how fuzzy system operate and produce decisions. These variables consist of **variable input** And **variable output**. variable input is the value or data given by keapda fuzzy system for processing, these variables represent conditions or factors that influence the final result in a system. Meanwhile, the output variable is the result or decision produced by fuzzy system after going through the rules fuzzy which has been determined.

In Scikit-Fuzzy, variables *input* defined by `` `ctrl.Antecedent()` `` who plays the role *input* in *fuzzy logic*. Here are the parameters used in `` `ctrl.Antecedent()` ``:

- `` `universe` ``: This parameter is the range or domain of values that the variable can take *fuzzy*. These parameters are defined with NumPy *array* by using `` `np.arange(start, stop, step)` ``, which shows the range of values of `` `start` `` until `` `stop` `` with steps `` `step` ``.
- `` `label` ``: Label or name given to a variable *fuzzy* will be used for reference in the rules *fuzzy logic*.

Here is an example of usage `` `ctrl.Antecedent()` ``:

```
from skfuzzy import control as ctrl
# Definition of quality control variables
quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')
```

In the code above, `quality` is a variable *fuzzy* which can have values up to 0 to 10 and variable *fuzzy* this is given a label `quality`.

Meanwhile variables *output*, Scikit-Fuzzy provides `ctrl.Consequent()` which is used to define variables *fuzzy* who plays the role *output* of the system to be determined based on *input* given and rules *fuzzy logic* which has been determined. `ctrl.Consequent()` has the same parameters a `ctrl.Antecedent()`.

Following is an implementation and example of `ctrl.Consequent()`:

```
from skfuzzy import control as ctrl
# Definition of tip output variable
tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')
```

the example above, `tip` is a variable *fuzzy* which can have a value of 0 to 25, and is labeled "*tip*" as a marker.

Here is the relationship between `ctrl.Antecedent()` as a variable *input* And `ctrl.Consequent()` as a variable *output*:

- `ctrl.Antecedent()` functions as a variable *input* which will be evaluated in the rules *fuzzy*, whereas `ctrl.Consequent()` is a variable *output* whose value will be influenced by the evaluation of these rules.
- In a control system *fuzzy*, rules are defined to relate the values of `ctrl.Antecedent()` the `ctrl.Consequent()`. For example "**If quality (*Antecedent*) bad, then tip (*Consequent*) low**".

13.2.2. Create a Membership Function

Internal membership function *fuzzy logic* is a curve that defines how deep each point is *input* or *output* mapped to the degree of membership in the set *fuzzy*. This function converts a numeric value to a value *fuzzy* which indicates how much a value falls into a particular category, such as “low,” “medium,” or “high.”

In scikit-fuzzy, `.trimf()` is one of the functions for creating membership in *fuzzy logic* which is used to determine the degree of membership of a value in a set *fuzzy*. ***Triumph*** is an abbreviation of *Triangular Membership Function* (Triangle membership function).

This function is used for mapping *fuzzy system* to map *input* into a membership value between 0 and 1. This function is the simplest and most widely used because its form is easy to understand.

Function `trimf()` defined as follows

```
import skfuzzy as fuzz
# Definition of membership in quality
quality['membership'] = fuzz.trimf(x, [a, b, c])
```

The following is an explanation of the parameters `trimf()`:

- `x`: *Array from the input value, this can be taken from Antecedent or Consequent which has been made.*
- `a` (*Lower Bound*): The starting point of the triangle, where the degree of membership starts from 0.
- `b` (*Peak / Middle Point*): The vertex of the triangle, where the degree of membership reaches 0.
- `c` (*Upper Bound*): End point of the triangle, where the degree of membership returns to 0.

The membership function that has been created can be visualized in the following way:

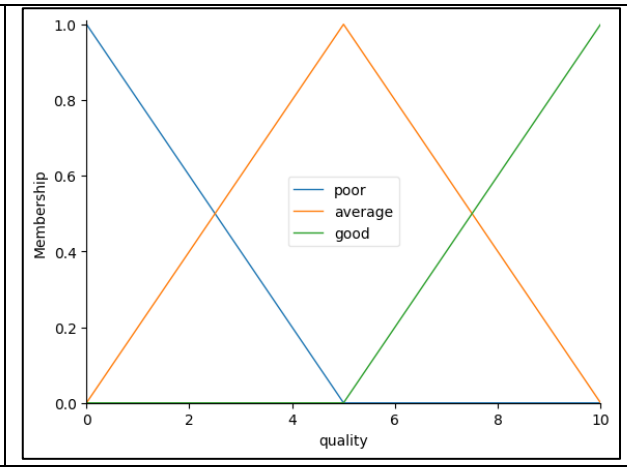
```
# Displays a visualization of the membership functions that have been provided
quality.show()
```

The result is a graph with a triangular shape that shows the degree of membership for each value in the given range.

Here is an implementation and example to create a membership function:

[1]:	`import skfuzzy as fuzz`
	`from skfuzzy import control as ctrl`
	`# Defines input variables`
	`quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')`
	`# Create a membership function for quality`
	`quality['poor'] = fuzz.trimf(quality.universe, [0, 0, 5])`
	`quality['average'] = fuzz.trimf(quality.universe, [0, 5, 10])`
	`quality['good'] = fuzz.trimf(quality.universe, [5, 10, 10])`
	
	`# Displays a visualization of quality membership`
	`quality.show()`

	Output:



13.2.3. Determination of Rules *Fuzzy*

Rule *fuzzy* is an important component in the system *fuzzy logic* which is used to connect *input* with *output* based on the specified logic. Rule *fuzzy* expressed in the form "**If… Then…"** that connects input conditions with outcomes *output*.

In scikit-fuzzy, the rule is defined by `.Rule()`. Paramater `.Rule()` consist of *Antecedent* or terms / conditions and *Consequent* or results/actions. Following is the implementation of `.Rule()`:

```
# quality, service = Antecedent, tip = Consequent
rule1 = ctrl.Rule(quality['poor'] | service['poor'], tip['low'])
```

The meaning of the rule above is: "**If the quality or service is poor then the tip will be low**".

Following are the internal logical operators `.Rule()` on parameters ***Antecedent***.

- `&` (*AND*): Connects two conditions that must both be met for the rule to trigger.
- `|` (*OR*): Connects two conditions where one of the conditions being met is sufficient to trigger the rule.
- `~` (*NOT*): Reverses the condition of the variable *fuzzy*.

Once the rules are defined, they are entered into *fuzzy system* use `.ControlSystem()`, which can then be simulated to produce *output* based on *input* which are given. Object to simulate *fuzzy system* must also be made using `.ControlSystemSimulation()`. Following is the implementation `.ControlSystemSimulation()`:

```
# Create a control system from several rules that have been created
tipping_ctrl = ctr.ControlSystem([rule1, rule2, rule3])
```

```
tipping_simulation = ctrl.ControlSystemSimulation(tipping_ctrl)
```

The following is an example of implementation of rule determination *fuzzy*:

```
# Definition of rules
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'])

# Combining rules
tipping_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
# Creation of objects to perform simulations
tipping_simulation = ctrl.ControlSystemSimulation(tipping_ctrl)
```

13.2.4. Giving *Input* and Calculation of *Fuzzy System*

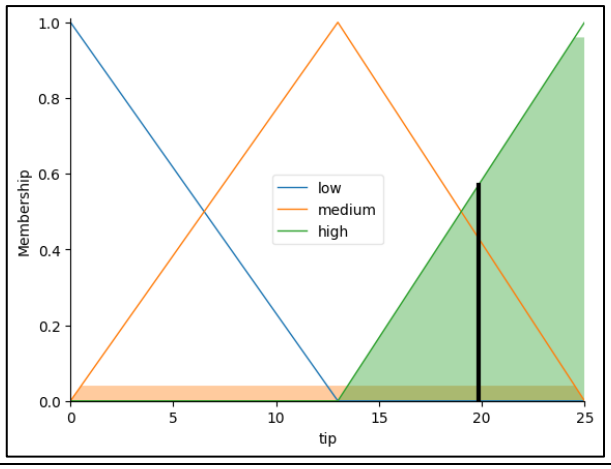
Giving input to ***Antecedent*** *fuzzy system* can use `.input()`. With *method* this is, the numeric value for the variable *input* can be processed *fuzzy system* to produce *output*. Then, after all *input* given to the variable *fuzzy*, this value can be calculated by `.compute()` to get results. Following is the implementation of `input` and `.compute()`:

[1]:
```
# Provides values for the quality and service variables
tipping_simulation.input['quality'] = 6.5
tipping_simulation.input['service'] = 9.8

# Calculates the output based on the given input
tipping_simulation.compute()
```

[2]:
```
# Print output and generate visualizations
print(tipping_simulation.output['tip']
tip.view(sim=tipping_simulation)
```

Output:

```
19.847607361963192
```



In the results above, *fuzzy system* provide recommendations if value *quality* 6.5 and *service* 9.8, for *tip* given is 19.84%.

13.3. *Study Case System Fuzzy with Scikit-Fuzzy*

You are asked to create a *fuzzy system* which models how *tip* given in a restaurant. When *tip* provided, two main factors will be considered: service quality (***service***) and food quality (***quality***). These two factors are rated on a scale of 0 to 10, where 0 is very poor and 10 is very good. Based on this assessment, *fuzzy system* will calculate the recommended amount *tip* between 0% to 25%. Following is the information provided:

***Input* (*Antecedents*):**

- ***Quality***
  - Value range: [0 – 10]
  - Fuzzy set:
    - Poor: *Triangular* with point [0, 0, 5]
    - Average: *Triangular* with point [0, 5, 10]
    - Good: *Triangular* with point [5, 10, 10]
- ***Service***
  - Value Range: [0 – 10]
  - *Fuzzy set*:
    - Poor: *Triangular* with point [0, 0, 5]
    - Average: *Triangular* with point, 5, 10]
    - Good: *Triangular* with point [5, 10, 10]

***Output* (*Consequent*):**

- **Tip**
  - Value Range = [0 – 25].
  - *Fuzzy set*:
    - *Low*: *Triangular* with point [0, 0, 13].
    - *Medium*: *Triangular* with point [0, 13, 15].
    - *High*: *Triangular* with point [13, 13, 25].

Aturan *Fuzzy*

- Aturan 1: If ***service*** `good` and ***quality*** `good` for ***tip*** `high`.
- Aturan 2: If ***service*** `average` for ***tip*** `medium`.
- Aturan 3: If ***service*** `poor` and ***quality*** `poor` for ***tip*** `low`.

Sample Case

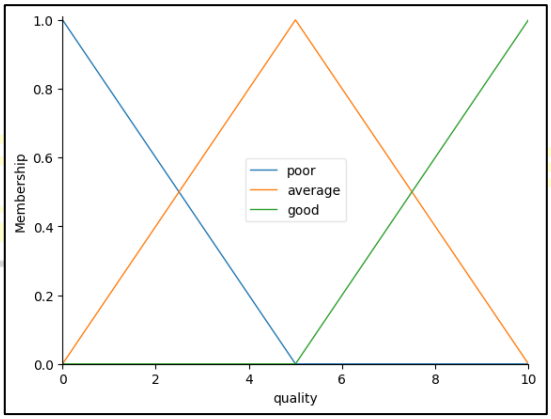
For example *service* rated with 9.8 and *quality* rated with 6.5. Size *fuzzy* above, a recommendation will provide recommendations *tip* amounted to 19.8%.

[1]:
```
# Import the required modules
import skfuzzy as fuzz
import numpy as np
from skfuzzy import control as ctrl
import matplotlib.pyplot as plt
```

[2]:
```
# Definition of input variable
quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')
service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')
# Definisi variabel output
tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')
```

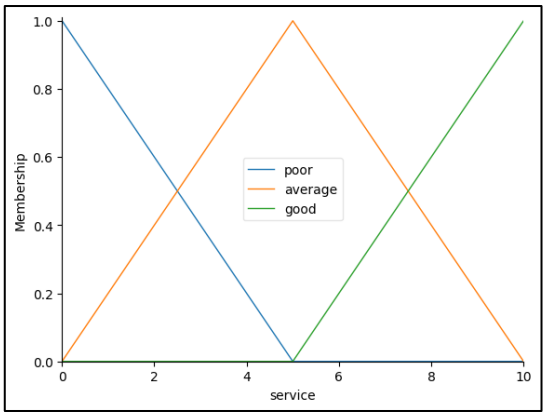
[3]:
```
# Membership function for quality
quality['poor'] = fuzz.trimf(quality.universe, [0,0,5])
quality['average'] = fuzz.trimf(quality.universe, [0,5,10])
quality['good'] = fuzz.trimf(quality.universe, [5,10,10])

# Visualization
quality.view()
```

Output:



[4]:
```
# Membership function for service
service['poor'] = fuzz.trimf(service.universe, [0, 0, 5])
service['average'] = fuzz.trimf(service.universe, [0, 5, 10])
service['good'] = fuzz.trimf(service.universe, [5, 10, 10])

service.view()
```

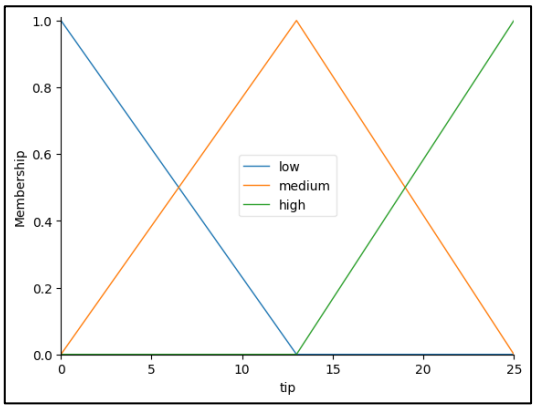
Output:



[5]:
```
# Membership function for tips
```

```
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])

tip.view()
```

Output:



[5]:

```
# Create rules from fuzzy systems
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'])
```

[6]:

```
# Create objects for defuzzification
tipping_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
tipping_simulation = ctrl.ControlSystemSimulation(tipping_ctrl)
```

[7]:

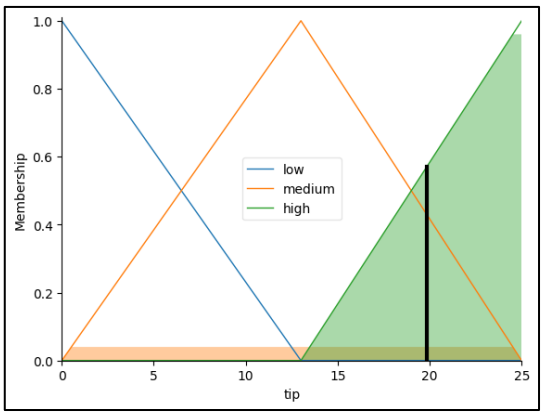
```
# Provides input to the Antecedent variable
tipping_simulation.input['quality'] = 6.5
tipping_simulation.input['service'] = 9.8
# Calculates the output of the fuzzy system
tipping_simulation.compute()
```

[8]:

```
# Displays results & visualization of calculation
print(tipping_simulation.output['tip']
tip.view(sim=tipping_simulation)
```

Output:

```
19.847607361963192
```



13.4. Scikit-Fuzzy Programming Practice Questions

Change the membership function *quality* And *service* on the question *study case* 13.3. Membership function *quality* can be changed a Figure *22*(a) and membership function *service* can be changed as Figure *22*(b).

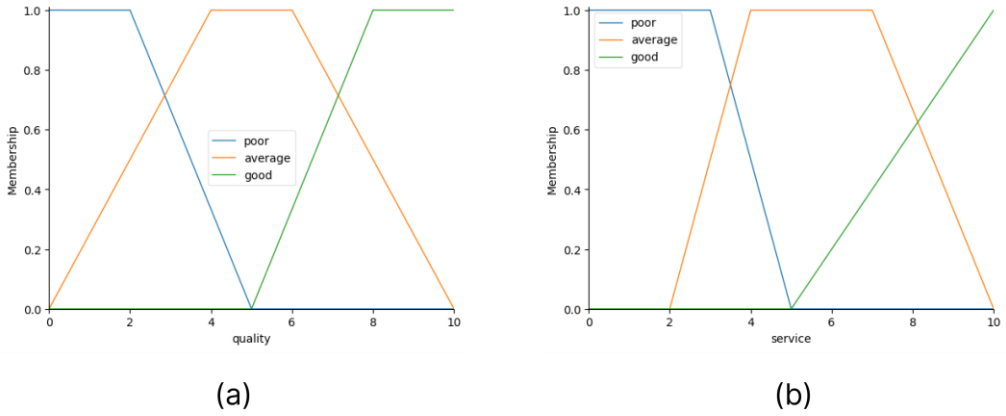


Figure 22. Membership function graph for practice questions. (a) Graph of the quality membership function. (b) Service membership function graph.

Kontak Kami :

@dky2921g

@informaticslab

@informaticslab_telu

informaticslab@telkomuniversity.ac.id

informatics.labs.telkomuniversity.ac.id