1. Program to identify the category of the IP address for the given IP address

2. Program to implement sliding window protocol

3. Program for Socket pair system call usage in IPC

4. Program for Socket options using signals

5. Program to implement Echo concurrent Stream Server

6. Program to implement Echo concurrent stream client

7. Program to implement Listener and Talker

8. Program to implement TCP time service

9. Program to implement UDP time service

10. Program to implement Ping service

11. Program to implement Route tracing program

12. Program to implement File Transfer Protocol

13. Program to implement any shortest path routing Algorithm

14. Program to implement Distance Vector Routing Implementation

15. Program to implement ICMP Error Message simulations

16. Program to implement Reverse Address Resolution Protocol

#function to determine the class of an Ip address

def findClass(ip):

if(ip[0] >= 0 and ip[0] <= 127):

return "A"

else if(ip[0] >=128 and ip[0] <= 191):

return "B"

else if(ip[0] >= 192 and ip[0] <= 223):

return "C"

else if(ip[0] >= 224 and ip[0] <= 239):

return "D"

else:

return "E"

#function to separate network and host id from the given ip address

def separate(ip, className):

#for class A network

if(className == "A"):

print("Network Address is : ", ip[0])

print("Host Address is : ", ".".join(ip[1:4]))

#for class B network

else if(className == "B"):

print("Network Address is : ", ".".join(ip[0:2]))

print("Host Address is : ", ".".join(ip[2:4]))

#for class C network

else if(className == "C"):

print("Network Address is : ", ".".join(ip[0:3]))

print("Host Address is : ", ip[3])

else:

print("In this Class, IP address is not divided into Network and Host ID")

#driver's code

if \_\_name\_\_ == "\_\_main\_\_":

ip = "192.226.12.11"

ip = ip.split(".")

ip = [int(i) for i in ip]

#getting the network class

networkClass = findClass(ip)

print("Given IP address belongs to class : ", networkClass)

#printing network and host id

ip = [str(i) for i in ip]

separate(ip, networkClass)

**Program to get Hostname and its IP address**

## importing socket module

import socket

## getting the hostname by socket.gethostname() method

hostname = socket.gethostname()

## getting the IP address using socket.gethostbyname() method

ip\_address = socket.gethostbyname(hostname)

## printing the hostname and ip\_address

print(f"Hostname: {hostname}")

print(f"IP Address: {ip\_address}")

**Build a GUI Application to ping the host using Python**

* Last Updated : 04 Dec, 2021

**Prerequisite:**[Python GUI –](https://www.geeksforgeeks.org/python-gui-tkinter/)Tkinter

In this article, we are going to see how to ping the host with a URL or IP using the python pingmodule in Python. This module provides a simple way to ping in python. And It checks the host is available or not and measures how long the response takes.

“Before” starting we need to install this module into your system.

pip install pythonping

The GUI would look like below:

***Syntax:****ping(‘URL or IP’)*

***Parameter:***

* *verbose : enables the verbose mode, printing output to a stream*
* *timeout : is the number of seconds you wish to wait for a response, before assuming the target is unreachable*
* *payload : allows you to use a specific payload (bytes)*
* *size : is an integer that allows you to specify the size of the ICMP payload you desire*

**Code:**

* Python3

|  |
| --- |
| # import module  from pythonping import ping    # pinging the host  ping('www.google.com', verbose=True) |

**Output:**

Reply from 142.250.71.4, 9 bytes in 61.09ms

Reply from 142.250.71.4, 9 bytes in 60.24ms

Reply from 142.250.71.4, 9 bytes in 60.22ms

Reply from 142.250.71.4, 9 bytes in 60.04ms

Reply from 142.250.71.4, 9 bytes in 61.09ms

Reply from 142.250.71.4, 9 bytes in 60.24ms

Reply from 142.250.71.4, 9 bytes in 60.22ms

Reply from 142.250.71.4, 9 bytes in 60.04ms

Round Trip Times min/avg/max is 60.04/60.4/61.09 ms

**Implementation for GUI:**

Pinging GUI Application with Tkinter

* Python3

|  |
| --- |
| # import modules  from tkinter import \*  from pythonping import ping    def get\_ping():      result = ping(e.get(), verbose=True)      res.set(result)    # object of tkinter  # and background set for light grey  master = Tk()  master.configure(bg='light grey')    # Variable Classes in tkinter  res = StringVar()    # Creating label for each information  # name using widget Label  Label(master, text="Enter URL or IP :",        bg="light grey").grid(row=0, sticky=W)  Label(master, text="Result :", bg="light grey").grid(row=1, sticky=W)    # Creating label for class variable  # name using widget Entry  Label(master, text="", textvariable=res, bg="light grey").grid(      row=1, column=1, sticky=W)    e = Entry(master)  e.grid(row=0, column=1)    # creating a button using the widget  # Button that will call the submit function  b = Button(master, text="Show", command=get\_ping)  b.grid(row=0, column=2, columnspan=2, rowspan=2, padx=5, pady=5)   mainloop() |

Introduction

We want to find the path with the smallest total weight among the possible paths we can take.

Dijkstra's shortest path algorithm

Dijkstra's algorithm is an iterative algorithm that provides us with the shortest path from one particular starting node (**a** in our case) to all other nodes in the graph.

To keep track of the total cost from the start node to each destination we will make use of the **distance** instance variable in the **Vertex** class. The **distance** instance variable will contain the current total weight of the smallest weight path from the start to the vertex in question. The algorithm iterates once for every vertex in the graph; however, the order that we iterate over the vertices is controlled by a **priority queue** (actually, in the code, I used **heapq**).

The value that is used to determine the order of the objects in the priority queue is **distance**. When a vertex is first created **distance** is set to a very large number.

When the algorithm finishes the distances are set correctly as are the predecessor (**previous** in the code) links for each vertex in the graph.

Code

In the code, we create two classes: **Graph**, which holds the master list of vertices, and **Vertex**, which represents each vertex in the graph (see [Graph data structure](http://www.bogotobogo.com/python/python_graph_data_structures.php)).

The source file is [Dijkstra\_shortest\_path.py](http://www.bogotobogo.com/python/files/Dijkstra/Dijkstra_shortest_path.py).

The function **dijkstra()** calculates the shortest path. The **shortest()** function constructs the shortest path starting from the target ('e') using predecessors.

import sys

class Vertex:

def \_\_init\_\_(self, node):

self.id = node

self.adjacent = {}

# Set distance to infinity for all nodes

self.distance = sys.maxint

# Mark all nodes unvisited

self.visited = False

# Predecessor

self.previous = None

def add\_neighbor(self, neighbor, weight=0):

self.adjacent[neighbor] = weight

def get\_connections(self):

return self.adjacent.keys()

def get\_id(self):

return self.id

def get\_weight(self, neighbor):

return self.adjacent[neighbor]

def set\_distance(self, dist):

self.distance = dist

def get\_distance(self):

return self.distance

def set\_previous(self, prev):

self.previous = prev

def set\_visited(self):

self.visited = True

def \_\_str\_\_(self):

return str(self.id) + ' adjacent: ' + str([x.id for x in self.adjacent])

class Graph:

def \_\_init\_\_(self):

self.vert\_dict = {}

self.num\_vertices = 0

def \_\_iter\_\_(self):

return iter(self.vert\_dict.values())

def add\_vertex(self, node):

self.num\_vertices = self.num\_vertices + 1

new\_vertex = Vertex(node)

self.vert\_dict[node] = new\_vertex

return new\_vertex

def get\_vertex(self, n):

if n in self.vert\_dict:

return self.vert\_dict[n]

else:

return None

def add\_edge(self, frm, to, cost = 0):

if frm not in self.vert\_dict:

self.add\_vertex(frm)

if to not in self.vert\_dict:

self.add\_vertex(to)

self.vert\_dict[frm].add\_neighbor(self.vert\_dict[to], cost)

self.vert\_dict[to].add\_neighbor(self.vert\_dict[frm], cost)

def get\_vertices(self):

return self.vert\_dict.keys()

def set\_previous(self, current):

self.previous = current

def get\_previous(self, current):

return self.previous

def shortest(v, path):

''' make shortest path from v.previous'''

if v.previous:

path.append(v.previous.get\_id())

shortest(v.previous, path)

return

import heapq

def dijkstra(aGraph, start, target):

print '''Dijkstra's shortest path'''

# Set the distance for the start node to zero

start.set\_distance(0)

# Put tuple pair into the priority queue

unvisited\_queue = [(v.get\_distance(),v) for v in aGraph]

heapq.heapify(unvisited\_queue)

while len(unvisited\_queue):

# Pops a vertex with the smallest distance

uv = heapq.heappop(unvisited\_queue)

current = uv[1]

current.set\_visited()

#for next in v.adjacent:

for next in current.adjacent:

# if visited, skip

if next.visited:

continue

new\_dist = current.get\_distance() + current.get\_weight(next)

if new\_dist < next.get\_distance():

next.set\_distance(new\_dist)

next.set\_previous(current)

print 'updated : current = %s next = %s new\_dist = %s' \

%(current.get\_id(), next.get\_id(), next.get\_distance())

else:

print 'not updated : current = %s next = %s new\_dist = %s' \

%(current.get\_id(), next.get\_id(), next.get\_distance())

# Rebuild heap

# 1. Pop every item

while len(unvisited\_queue):

heapq.heappop(unvisited\_queue)

# 2. Put all vertices not visited into the queue

unvisited\_queue = [(v.get\_distance(),v) for v in aGraph if not v.visited]

heapq.heapify(unvisited\_queue)

if \_\_name\_\_ == '\_\_main\_\_':

g = Graph()

g.add\_vertex('a')

g.add\_vertex('b')

g.add\_vertex('c')

g.add\_vertex('d')

g.add\_vertex('e')

g.add\_vertex('f')

g.add\_edge('a', 'b', 7)

g.add\_edge('a', 'c', 9)

g.add\_edge('a', 'f', 14)

g.add\_edge('b', 'c', 10)

g.add\_edge('b', 'd', 15)

g.add\_edge('c', 'd', 11)

g.add\_edge('c', 'f', 2)

g.add\_edge('d', 'e', 6)

g.add\_edge('e', 'f', 9)

print 'Graph data:'

for v in g:

for w in v.get\_connections():

vid = v.get\_id()

wid = w.get\_id()

print '( %s , %s, %3d)' % ( vid, wid, v.get\_weight(w))

dijkstra(g, g.get\_vertex('a'), g.get\_vertex('e'))

target = g.get\_vertex('e')

path = [target.get\_id()]

shortest(target, path)

print 'The shortest path : %s' %(path[::-1])

Output:

Graph data:

( a , f, 14)

( a , c, 9)

( a , b, 7)

( c , d, 11)

( c , f, 2)

( c , a, 9)

( c , b, 10)

( b , d, 15)

( b , a, 7)

( b , c, 10)

( e , d, 6)

( e , f, 9)

( d , c, 11)

( d , e, 6)

( d , b, 15)

( f , a, 14)

( f , c, 2)

( f , e, 9)

Dijkstra's shortest path

updated : current = a next = f new\_dist = 14

updated : current = a next = c new\_dist = 9

updated : current = a next = b new\_dist = 7

updated : current = b next = d new\_dist = 22

not updated : current = b next = c new\_dist = 9

updated : current = c next = d new\_dist = 20

updated : current = c next = f new\_dist = 11

updated : current = f next = e new\_dist = 20

not updated : current = d next = e new\_dist = 20

The shortest path : ['a', 'c', 'f', 'e']