EXPERIMENT 4: SHORTEST PATH ALGORITHMS USING PYTHON

(TEAM 9 – LINK STATE ROUTING)

AIM:

To demonstrate the link state routing algorithm and protocols using Python.

SOFTWARES REQUIRED:

1. Microsoft Visual Studio Code, Microsoft Corporation
2. Oracle VM VirtualBox 6.1.38, Oracle Corporation
3. Python 3.10.7 (64-bit), Python Software Foundation
4. Ubuntu 22.04 (64-bit) Operating System
5. Cisco Packet Tracer 8.2.0

THEORY:

Link State Routing Algorithm is a routing technique which is used by routers to share information/knowledge about their neighbouring routers with the rest of the routers on their network. The link state algorithm helps each router in the network to make its routing table, with the help of the information about the network topology.

Routing:

1. Routing is a process in which the path each data packet has to follow is established. In this process, a routing table is created.
2. This table contains all the information on the path a data packet has to follow to reach its appropriate destination.
3. Various algorithms are used to find the optimized path for the transmission of a data packet.
4. An optimal route is a path in which the connection cost is the least from the source to the destination.
5. Here we use an algorithm called Dijkstra's algorithm for the calculation of the path with the least cost, after which the routing table is created.
6. This table created by each router is interchanged with other routers in the network.
7. This in turn helps in the speedy and reliable transmission of data.

Dijkstra's Algorithm: It solves the single-source shortest-path problem for graphs with nonnegative edge costs.

Declare all nodes unscanned and initialize *d* and parent.

while there is an unscanner node with a tentative distance < $+\infty$ do

u: = the unscanned node with minimal tentative distance

relax all edges *(u, v)* out of u and declare *u* scanner

Flooding:

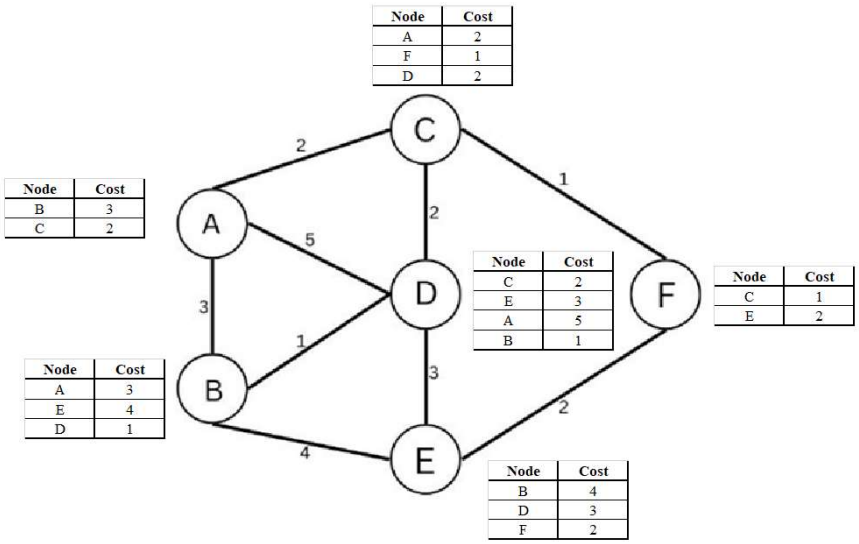
1. It is a process in which every router (except its neighbours) transmits information to other routers about its neighbouring routers. This process is called flooding.
2. Every router receiving the information sends copies of that information to all its neighbouring routers.
3. Finally, all routers in the network have a copy of the same information. This information is only exchanged when there is a modification in the information, which makes the link state routing algorithm effective.

Phases of Link State Routing:

There are two phases in link state routing: –

1. Reliable Flooding – In this phase, the information about neighbouring routers is shared and gathered through the process of flooding. This phase is again divided into two phases,
    - Initial State – In this state, every router becomes aware of the cost of transmission of its neighbours.
    - Final State – In this state, every router knows the information about the entire graph.
2. Route Calculation – In this phase all routers in the network use algorithms like Dijkstra's algorithm to find the shortest or optimal path to all the other routers in the network.

Example of Link State Routing:



Iteration	Tree	B	C	D	E	F
Initial	{A}	3	2	5	∞	∞
1	{A, C}	3	-	4	∞	3
2	{A, B, C}	-	-	4	7	3
3	{A, B, C, F}	-	-	4	5	-
4	{A, B, C, D, F}	-	-	-	5	-
5	{A, B, C, D, E, F}	-	-	-	-	-

SOURCE CODE:

Node.py

```
#!/usr/bin/python3
class Node:
    def __init__(self,element):
        self._element = element

    def __str__(self):
        return str(self._element)
    def __lt__(self,v):
        return self._element < v.element()
    def element(self):
        return self._element
```

Link.py

```
#!/usr/bin/python3
class Link:

    def __init__(self,v,w,element):
        #create link between node v and w
        self._nodes=(v,w)
        self._element = element

    def __str__(self):
        return("\n" + str(self._nodes[0])+"-"+
               str(self._nodes[1])+":"+str(self._element))

    def nodes(self):
        return self._nodes

    def start(self):
        return self._nodes[0]

    def end(self):
        return self._nodes[1]

    def opposite(self,v):
        if self._nodes[0]==v:
```

```
            return self._nodes[1]
        elif self._nodes[1]==v:
            return self._nodes[0]
        else:
            return None

    def element(self):
        return self._element
```

APQ_Heap.py

```
#!/usr/bin/python3
class APQHeap:
    """ Maintain an collection of items, popping by lowest key.

        This implementation maintains the collection using a binary heap.
        Keys and or values can be updated.
        Items can be arbitrarily removed.
    """
    class Element:
        """ An element with a key and value and an index to the heap-array.
"""

        def __init__(self, k, v, index):
            self._key = k
            self._value = v
            self._index = index

        def __eq__(self, other):
            """ Return True if this key equals the other key. """
            return self._key == other._key

        def __lt__(self, other):
            """ Return True if this key is less than the other key. """
            return self._key < other._key

        def _wipe(self):
            """ Set the instance variables to None. """
            self._key = None
            self._value = None
            self._index = None

    def __init__(self):
        """ Create an APQ with no elements. """
        self._heap = []
        self._size = 0

    def __str__(self):
        """ Return a breadth-first string of the values. """
```

```
        outstr = '<-'
        for elt in self._heap:
            outstr += str(elt._value) + ':' + str(elt._key) + '-'
        return outstr + '<'

    def add(self, key, value):
        """ Add Element(key,value,size) to the heap. """
        e = APQHeap.Element(key, value, self._size)
        self._heap.append(e)
        self._upheap(self._size)
        self._size += 1
        return e

    def min(self):
        """ Return the min priority key,value. """
        if self._size:
            return self._heap[0]._key, self._heap[0]._value
        return None, None

    def _swap_last_into_place(self, topos):
        """ Swap the last item into position topos. """
        if self._size > 1: #if other items, restructure
            self._heap[topos] = self._heap[self._size - 1]
            self._heap[topos]._index = topos
            self._heap.pop()
            self._size -= 1
            parent = self._parent(topos)
            if parent > -1 and self._heap[topos]._key <
self._heap[parent]._key:
                self._upheap(topos)
            else:
                self._downheap(topos)
        else:
            self._heap.pop()
            self._size -= 1

    def remove_min(self):
        """ Remove and return the min priority key,value. """
        returnvalue = None
        returnkey = None
        if self._size:
            returnkey = self._heap[0]._key
            returnvalue = self._heap[0]._value
            self._heap[0]._wipe()
            self._swap_last_into_place(0)
        return returnkey, returnvalue

    def length(self):
```

```
        """ Return the number of items in the heap. """
        return self._size

    def is_empty(self):
        """ Return True if the heap is empty. """
        return self._size == 0

    def _left(self, posn):
        """ Return the index of the left child of elt at index posn. """
        return 1 + 2*posn

    def _right(self, posn):
        """ Return the index of the right child of elt at index posn. """
        return 2 + 2*posn

    def _parent(self, posn):
        """ Return the index of the parent of elt at index posn. """
        return (posn - 1)//2

    def _upheap(self, posn):
        """ Bubble the item in posn in the heap up to its correct place. """
        if posn > 0 and self._upswap(posn, self._parent(posn)):
            self._upheap(self._parent(posn))

    def _upswap(self, posn, parent):
        """ If healp elt at posn has lower key than parent, swap. """
        if self._heap[posn] < self._heap[parent]:
            self._heap[posn], self._heap[parent] = self._heap[parent],
self._heap[posn]
            self._heap[posn]._index = posn
            self._heap[parent]._index = parent
            return True
        return False

    def _downheap(self, posn):
        """ Bubble the item in posn in the heap down to its correct place. """
        #find minchild position
        #if minchild is in the heap
        #    if downswap with minchild is true
        #        downheap minchild
        minchild = self._left(posn)
        if minchild < self._size:
            if (minchild + 1 < self._size and
                self._heap[minchild]._key > self._heap[minchild + 1]._key):
                minchild +=1
            if self._downswap(posn, minchild):
                self._downheap(minchild)
```

```
    def _downswap(self, posn, child):
        """ If healp elt at posn has lower key than child, swap. """
        #Note: this could be merged with _upswap to provide a general
        #heapswap(first, second) method, which swaps if the element
        #first has lower key than the element second
        if self._heap[posn]._key > self._heap[child]._key:
            self._heap[posn], self._heap[child] = self._heap[child],
self._heap[posn]
            self._heap[posn]._index = posn
            self._heap[child]._index = child
            return True
        return False

    def update_key(self, location, key):
        """ Update the key of the item in location. """
        #location is actually an Element
        #updating the key may require us to rebalance the heap
        location._key = key
        parent = self._parent(location._index)
        if parent > -1 and location._key < self._heap[parent]._key:
            self._upheap(location._index)
        else:
            self._downheap(location._index)

    def update_value(self, location, value):
        """ Update the value of the item in location. """
        #location is actually an Element
        location._value = value

    def get_key(self, location):
        """ Return the current key (priority value) for item in location. """
        #location is actually an Element
        return location._key

    def remove(self, location):
        """ Remove the item in location from the APQ, and return key,value.
"""
        #location is actually an Element
        returnkey = location._key
        returnvalue = location._value
        pos = location._index
        self._swap_last_into_place(pos)
        location._wipe()
        location = None
        return returnkey, returnvalue

    def _printstructure(self):
        """ Print out the elements one to a line. """
```

```
        for elt in self._heap:
            if elt is not None:
                print('(', elt._key, ',', elt._value, ',', elt._index, ')')
            else:
                print('*')
```

Graph.py

```
#!/usr/bin/python3
from node import Node
from link import Link
from apq_heap import APQHeap

class Graph:
    #the keys are the nodes and values are the link
    def __init__(self):
        #create an initial empty graph
        self._structure = dict()
    def __str__(self):
        strnodes="\nNodes"
        for v in self._structure:
            strnodes +="\t"+str(v)
        links=self.links()
        strlink="\nLinks:"
        for w in links:
            strlink +=str(w)
        return ("Total Nodes: " +str(self.num_nodes())+"\n"+"Total Links: "+
                                str(self.num_links())+ strnodes+strlink)
    #ADT methods to query the graph

    def num_nodes(self):
        return len(self._structure)

    def num_links(self):
        num = 0
        for v in self._structure:
            num +=len(self._structure[v])
        return num // 2

    def nodes(self):
        return [key for key in self._structure]

    def get_node_by_label(self, element):
        for v in self._structure:
            if v.element()== element:
                #print(v)
                return v
        return None
    def links(self):
```

```
        linklist = []
        for v in self._structure:
            for w in self._structure[v]:
                #to avoid duplicates,only return if v in the first nodes
                if self._structure[v][w].start() ==v:
                    linklist.append(self._structure[v][w])
        return linklist

    def get_links(self,v):
        #list of all links
        if v in self._structure:
            linklist = []
            for w in self._structure[v]:
                linklist.append(self._structure[v][w])
            return linklist
        return None

    def get_link(self,v,w):
        #link between v and w
        if(self._structure != None and v in self._structure
           and w in self._structure[v]):
            return self._structure[v][w]
        return None

    def degree(self, v):
        #return degree of node v
        return len(self._structure[v])

    def get_weight(self,v,w):
        #Cost between v and w
        if(self._structure !=None and v in self._structure
            and w in self._structure[v]):
            return len(self._structure[v][w])

    def add_node(self,element):
        v=Node(element)
        self._structure[v]=dict()
        return v


    def add_node_if_new(self, element):
        for v in self._structure:
            if v.element()== element:
                #print("already there")
                return v
        return self.add_node(element)

    def add_link(self, v, w,element):
```

```
        if not v in self._structure or not w in self._structure:
            return None
        e=Link(v,w,element)
        self._structure[v][w]=e
        self._structure[w][v]=e
        return e

    def highestdegreenode(self):
        #return the vertex with highet degree
        hd=-1
        hdv=None
        for v in self._structure:
            if self.degree(v)>hd:
                hd=self.degree(v)
                hdv = v
        return hdv

    def linkState(self, src):
        closed = dict()
        locs = dict()
        pred = {src:None}
        apq = APQHeap()#empty apq
        locs[src]=apq.add(0,src)
        while not apq.is_empty():
            key, u = apq.remove_min()
            del locs[u]
            closed[u]=(key,pred[u])
            for e in self.get_links(u):
                v = e.opposite(u)
                if v not in closed:
                    newcost = e.element() + key
                    if v not in locs:
                        pred[v] = u
                        locs[v] = apq.add(newcost,v)
                    elif newcost< apq.get_key(locs[v]):
                        pred[v] = u
                        apq.update_key(locs[v],newcost)
        return closed

    def graphreader(filename):

        """ Read and return the route map in filename. """
        graph = Graph()
        file = open(filename, 'r')
        entry = file.readline() #either 'Node' or 'Edge'
        num = 0
        print("**********Welcome to Link State Topology*************")
        while entry == 'Node\n':
```

```
            num += 1
            nodeid = int(file.readline().split()[1])
            node = graph.add_node(nodeid)
            entry = file.readline() #either 'Node' or 'Edge'
        print("\tFound", num, 'Nodes and added into the graph')
        num = 0
        while entry == 'Edge\n':
            num += 1
            source = int(file.readline().split()[1])
            sv = graph.get_node_by_label(source)
            target = int(file.readline().split()[1])
            tv = graph.get_node_by_label(target)
            length = float(file.readline().split()[1])
            edge = graph.add_link(sv, tv, length)
            file.readline() #read the one-way data
            entry = file.readline() #either 'Node' or 'Edge'
        print("\tFound", num, 'Links and added into the graph')
        print(graph)
        return graph
```

Test.py

```
#!/usr/bin/python3
from  Graph import Graph
import networkx as nx
import matplotlib.pyplot as plt
#Test methods
def test1():
    graph = Graph.graphreader("graph1.txt")
    for i in range(1,6):
        src = graph.get_node_by_label(i)
        d = graph.linkState(src)
        for element in sorted(d):
            print("Path from " + str(src) + " to Node " +str(element) +
                  " : Length :" + str(d[element][0]) + ". Preceding : " +
str(d[element][1]) )
        for element in sorted(d):
            cost,pred = d[element]
            print("From Node:" + "Cost")
            print(element,'       :',cost, '(', pred ,')')
def test2():
    graph = Graph.graphreader("simplegraph2.txt")
    src = graph.get_node_by_label(14)

    d = graph.linkState(src)
    for element in d:
        print("Path from "+str(src) + "  to Node " + str(element)
            + ". Length :" + str(d[element][0]) + ". Preceding : " +
str(d[element][1]))
```

```
        for element in sorted(d):
            cost,pred = d[element]
            print("From Node:" + "Cost")
            print(element,'       :',cost, '(', pred ,')')

if __name__=="__main__":
    test1()
    #test2()
```

Node_Info.txt

Node	to: 3	from: 3
id: 1	length: 2	to: 4
Node	oneway: false	length: 2
id: 2	Edge	oneway: false
Node	from: 1	Edge
id: 3	to: 4	from: 3
Node	length: 5	to: 6
id: 4	oneway: false	length: 1
Node	Edge	oneway: false
id: 5	from: 2	Edge
Node	to: 4	from: 4
id: 6	length: 1	to: 5
Edge	oneway: false	length: 3
from: 1	Edge	oneway: false
to: 2	from: 2	Edge
length: 3	to: 5	from: 5
oneway: false	length: 4	to: 6
Edge	oneway: false	length: 2
from: 1	Edge	oneway: false

SCREENSHOTS OF INPUTS & OUTPUTS:

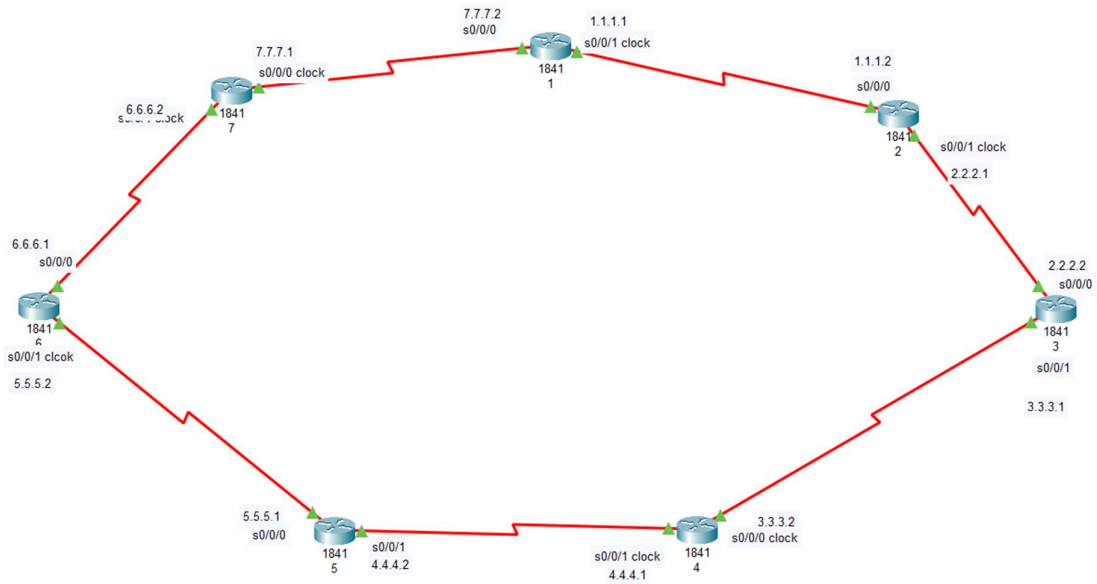
```
Total Nodes: 6
Total Links: 9
Nodes   1   2   3   4   5   6
Links:
1-2:3.0
1-3:2.0
1-4:5.0
2-4:1.0
2-5:4.0
3-4:2.0
3-6:1.0
4-5:3.0
5-6:2.0
Path from 1 to Node 1 : Length :0. Preceding : None
Path from 1 to Node 2 : Length :3.0. Preceding : 1
Path from 1 to Node 3 : Length :2.0. Preceding : 1
Path from 1 to Node 4 : Length :4.0. Preceding : 3
Path from 1 to Node 5 : Length :5.0. Preceding : 6
Path from 1 to Node 6 : Length :3.0. Preceding : 3
From Node:Cost
1        : 0 ( None )
From Node:Cost
2        : 3.0 ( 1 )
From Node:Cost
3        : 2.0 ( 1 )
From Node:Cost
4        : 4.0 ( 3 )
From Node:Cost
5        : 5.0 ( 6 )
From Node:Cost
6        : 3.0 ( 3 )
```

```
Total Nodes: 6
Total Links: 9
Nodes   1   2   3   4   5   6
Links:
1-2:3.0
1-3:2.0
1-4:5.0
2-4:1.0
2-5:4.0
3-4:2.0
3-6:1.0
4-5:3.0
5-6:2.0
Path from 1 to Node 1 : Length :0. Preceding : None
Path from 1 to Node 2 : Length :3.0. Preceding : 1
Path from 1 to Node 3 : Length :2.0. Preceding : 1
Path from 1 to Node 4 : Length :4.0. Preceding : 3
Path from 1 to Node 5 : Length :5.0. Preceding : 6
Path from 1 to Node 6 : Length :3.0. Preceding : 3
From Node:Cost
1        : 0 ( None )
From Node:Cost
2        : 3.0 ( 1 )
From Node:Cost
3        : 2.0 ( 1 )
From Node:Cost
4        : 4.0 ( 3 )
From Node:Cost
5        : 5.0 ( 6 )
From Node:Cost
6        : 3.0 ( 3 )
```

```
Total Nodes: 6
Total Links: 9
Nodes   1   2   3   4   5   6
Links:
1-2:3.0
1-3:2.0
1-4:5.0
2-4:1.0
2-5:4.0
3-4:2.0
3-6:1.0
4-5:3.0
5-6:2.0
Path from 1 to Node 1 : Length :0. Preceding : None
Path from 1 to Node 2 : Length :3.0. Preceding : 1
Path from 1 to Node 3 : Length :2.0. Preceding : 1
Path from 1 to Node 4 : Length :4.0. Preceding : 3
Path from 1 to Node 5 : Length :5.0. Preceding : 6
Path from 1 to Node 6 : Length :3.0. Preceding : 3
From Node:Cost
1        : 0 ( None )
From Node:Cost
2        : 3.0 ( 1 )
From Node:Cost
3        : 2.0 ( 1 )
From Node:Cost
4        : 4.0 ( 3 )
From Node:Cost
5        : 5.0 ( 6 )
From Node:Cost
6        : 3.0 ( 3 )
```


```
Total Nodes: 6
Total Links: 9
Nodes   1   2   3   4   5   6
Links:
1-2:3.0
1-3:2.0
1-4:5.0
2-4:1.0
2-5:4.0
3-4:2.0
3-6:1.0
4-5:3.0
5-6:2.0
Path from 1 to Node 1 : Length :0. Preceding : None
Path from 1 to Node 2 : Length :3.0. Preceding : 1
Path from 1 to Node 3 : Length :2.0. Preceding : 1
Path from 1 to Node 4 : Length :4.0. Preceding : 3
Path from 1 to Node 5 : Length :5.0. Preceding : 6
Path from 1 to Node 6 : Length :3.0. Preceding : 3
From Node:Cost
1        : 0 ( None )
From Node:Cost
2        : 3.0 ( 1 )
From Node:Cost
3        : 2.0 ( 1 )
From Node:Cost
4        : 4.0 ( 3 )
From Node:Cost
5        : 5.0 ( 6 )
From Node:Cost
6        : 3.0 ( 3 )
```

```
Total Nodes: 6
Total Links: 9
Nodes   1   2   3   4   5   6
Links:
1-2:3.0
1-3:2.0
1-4:5.0
2-4:1.0
2-5:4.0
3-4:2.0
3-6:1.0
4-5:3.0
5-6:2.0
Path from 1 to Node 1 : Length :0. Preceding : None
Path from 1 to Node 2 : Length :3.0. Preceding : 1
Path from 1 to Node 3 : Length :2.0. Preceding : 1
Path from 1 to Node 4 : Length :4.0. Preceding : 3
Path from 1 to Node 5 : Length :5.0. Preceding : 6
Path from 1 to Node 6 : Length :3.0. Preceding : 3
From Node:Cost
1        : 0 ( None )
From Node:Cost
2        : 3.0 ( 1 )
From Node:Cost
3        : 2.0 ( 1 )
From Node:Cost
4        : 4.0 ( 3 )
From Node:Cost
5        : 5.0 ( 6 )
From Node:Cost
6        : 3.0 ( 3 )
```



CONCLUSION:

Illustrated the link state routing algorithms and protocols. All the simulation results were verified successfully.

REFERENCES:

1. Foundations of Python 3 Network Programming, John Goerzen, Brandon Rhodes, Second Edition, Apress Publisher; ISBN-10: 1430258543, ISBN-13: 978-1430258544
2. Foundations of Python Network Programming by Beaulne, Alexandre Goerzen, John Membrey, Peter Rhodes, Brandon 2014.
3. (2008) Algorithms and Data Structures - The Basic Toolbox, Kurt Mehlorn, Peter Sanders, Springer; ISBN: 978-3-540-77977-3, e-ISBN: 978-3-540-77978-0

CONTRIBUTION:

1. Narendran S [CB.EN.U4CCE20036] – Code Work on Heaps + Testing with Node Information + Graph Visualization
2. Narun T [CB.EN.U4CCE20037] – Code Work on Graphs + Debugging
3. Pabbathi Greeshma [CB.EN.U4CCE20040] – Code Work on Linking + Reviewing
4. Santosh [CB.EN.U4CCE20053] – Code Work on Nodes + Documentation