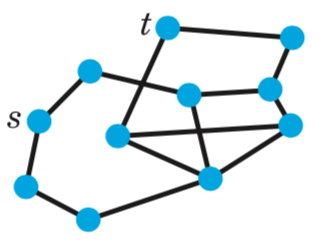
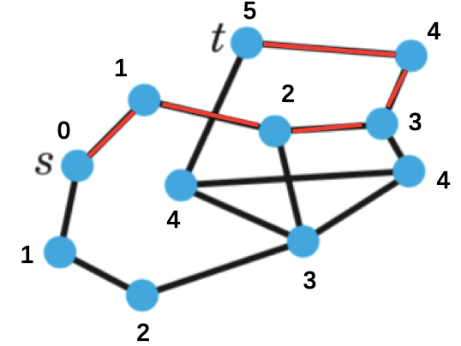
**ENGN2020 – HOMEWORK6**

### Problem 1

### K23-2-1:

The given graph is shown as below:



**(a)** **(b)**

**Fig 1.** **(a)**The given graph. **(b)** The shortest path from ***s*** to ***t***

### K23-2-2:

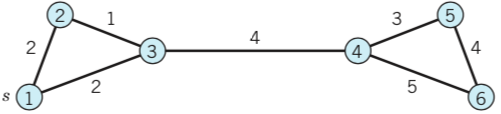
**(a)** **(b)**

**Fig 2.** **(a)**The given graph. **(b)** The shortest path from ***s*** to ***t***

### Problem 2

### K23-2-13:

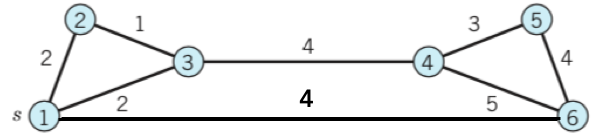
The postman problem is shown as below:



**Fig 3.** The given graph

The path should be: 1->2->3->4->5->6->4->3->1, the total length of the path is 26.

### K23-2-13 with modification:



**Fig 4.** The given graph

The path should be :1->2->3->4->5->6->1, the total length of the path is 18.

### Problem 3

### (b) get\_new\_neighbors method:

def get\_new\_neighbors(self, neighbors):

"""Compares the vertex names in the list "neighbors"

to the neighbors already contained in steptable.

Returns a (shorter) list of names of neighbors that have

not yet been discovered.

"""

#loop all items in the neighbors

for i in neighbors:

#loop all vertices that in the step table

for item in self.data:

#if already exits in the step talbe, remove it

if item[0].name == i:

neighbors.remove(i)

break

return neighbors

### (c) get\_reverse\_path method:

def get\_reverse\_path(self, vertex\_name):

"""Starting at the vertex named "vertex\_name", traces

backwards through the step table to find the shortest

distance to the origin. Note that this should only be

called\*after\*vertex\_name has been discovered.

"""

#initial list with given vertex name

result = [vertex\_name]

#get length of the recorded data

length = len(self.data)

#set the current vertex name

current = vertex\_name

#loop from step table

for i in range(length):

for vertex in self.data:

#find the current vertex in table

if current in vertex[2]:

result.insert(0,vertex[0].name)

current = vertex[0].name

return result

### (d) Moore’s algorithm

The function to get the shortest path from a given vertex to another given vertex is shown as below, please find the complete code in Appendix.

'''

\* @name: findPath

\* @description: use Moore`s algorithm to find the shortest path from start to end vertices

\* @param start: the name of the start vertex

\* @param end: the name of the end vertex

\* @return: list, the shortest path from start to end vertices

'''

def findPath(start,end):

#declare the queue

bfs = Queue();

#declare the step table

record = StepTable();

#create the start vertex

start = Vertex(start);

#append this start point into queue

bfs.append(start,0)

#save the this start point into step table

record.append(start,0);

#use Moore`s algorithm or so called bfs

while(bfs.queue!=[]):

#get the front the queue

nextNode = bfs.next()

#get the distance of this point

currentDistance = nextNode["distance"]

#get the neighbors of this point

neighbors = nextNode["vertex"].get\_neighbors();

#loop all neighbors

for item in neighbors:

#use a bool value to see if the vertex has been visited

visited = False

#loop all records in the step table to see if visited

for step in record.data:

if step[0].name == item:

visited = True

break

#if not visited

if(not visited):

#create the vertex

temp = Vertex(item);

#push it into queue

bfs.append(temp,currentDistance+1);

#record this step in step table

record.append(temp,currentDistance+1);

#record.print()

#get the path from start to end

path = record.get\_reverse\_path(end);

#print path

print(path)

The test result of this function to get the shortest path from “B” to “F” is:

['B', 'E', 'A', 'D', 'F']

The test result of this function to get the shortest path from “B” to “J” is:

['B', 'E', 'C', 'I', 'J']

### (e) shortest distance between two websites

Please find the complete code in Appendix.

The test result of this function to get the shortest path from “'https://www.brown.edu/academics/engineering/” to “https://www.brown.edu/academics/engineering/about” is:

﻿['https://www.brown.edu/academics/engineering/', 'https://www.brown.edu/academics/engineering/about']

The test result of this function to get the shortest path from “'https://www.brown.edu/academics/engineering/” to “https://www.brown.edu/academics/engineering/graduate-study/masters-and-phd-programs” is:

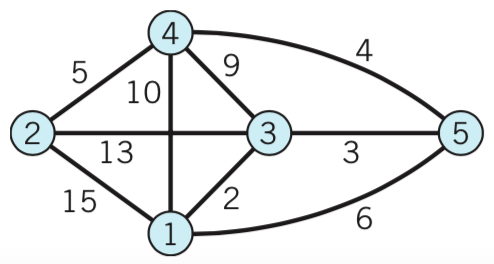
﻿['https://www.brown.edu/academics/engineering/',

'https://www.brown.edu/academics/engineering/graduate-study'

'https://www.brown.edu/academics/engineering/graduate-study/masters-and-phd-programs']

### Problem 4

### K23-3-4:



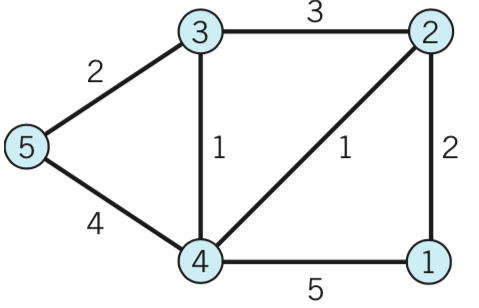
**Fig 5.** The given graph

Dijkstra’s Algorithm result is shown as below:

**Table 1.** Dijkstra’s Algorithm result

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Step | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | “15” | “2” | “10” | “6” |
| 1 | 0 | “15” | 2 | “10” | “5” |
| 2 | 0 | “15” | 2 | “9” | 5 |
| 3 | 0 | “14” | 2 | 9 | 5 |
| 4 | 0 | 14 | 2 | 9 | 5 |

### K23-3-5:



**Fig 6.** The given graph

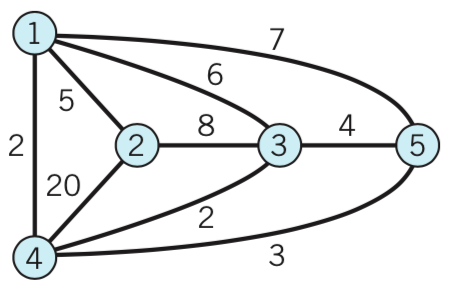
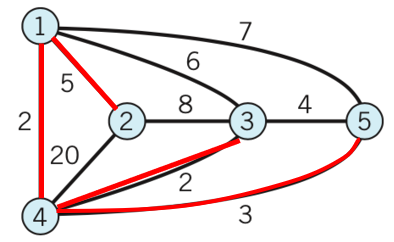
Dijkstra’s Algorithm result is shown as below:

**Table 2.** Dijkstra’s Algorithm result

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Step | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | “2” | “∞” | “5” | “∞” |
| 1 | 0 | 2 | “5” | “3” | “∞” |
| 2 | 0 | 2 | “4” | 3 | “7” |
| 3 | 0 | 2 | 4 | 3 | “6” |
| 4 | 0 | 2 | 4 | 3 | 6 |

### Problem 5

### K23-4-4



**(a)** **(b)**

**Fig 7.** **(a)**The given graph. **(b)** The shortest spanning tree by Kruskal’s algorithm

### K23-4-5

**(a)** **(b)**

**Fig 8.** **(a)**The given graph. **(b)** The shortest spanning tree by Kruskal’s algorithm

### Problem 6

### (a) Implementation of Prim Algorithm

import numpy as np

'''

\* @name: Prim

\* @description: the class to implement Prim`s algorithm for shortest spanning tree

\* @param adjMatrix: the given graph stored in adjacency matrix

'''

class Prim:

'''

\* @name: \_\_init\_\_

\* @description: constructor of the class

\* @param adjMatrix: the given graph stored in adjacency matrix

'''

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

#the input value is a adjacency matrix

self.graph = adjMatrix

'''

\* @name: nextVertex

\* @description: decide which is the next vertex to add to the tree

\* @param U: the vertices that already in the spanning tree

\* @param visited: list that save whether the vertex is used

\* @return: list, the new vertex added to the tree and the parent vertex of the new vertex

'''

def nextVertex(self,U,visited):

#get the number of vertices

vertexNum = self.graph.shape[0]

#initial the values

minValue = 10000

minIndex = 0

parent = 0

#loop all unvisited vertices

for i in range(vertexNum):

if not visited[i]:

#loop all nodes that already in the tree

for j in U:

#find the nearest distance to the vertices in the tree

if self.graph[i][j]< minValue and self.graph[i][j]!=0:

minValue = self.graph[i][j]

minIndex = i

parent = j

#return the list of the new vertex added to the tree and the parent vertex of the new vertex

return [minIndex,parent]

'''

\* @name: prim

\* @description: use Prim`s algorithm to create the shortest spanning tree

\* @param U: the vertices that already in the spanning tree

\* @param visited: list that save whether the vertex is used

'''

def prim(self):

#get the number of the vertices in the graph

vertexNum = self.graph.shape[0]

#set the all vertices by unvisited

visited = [False]\*vertexNum

#start with vertex 0

visited[0] = True

U =[0]

#loop all vertices

for i in range(vertexNum):

#call the member function to find next vertex

nextStep = self.nextVertex(U,visited)

#add it in U

U.append(nextStep[0]);

#set it as visited

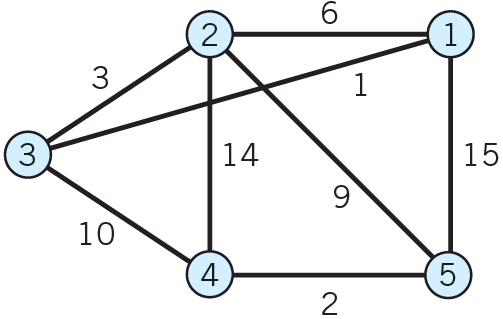
visited[nextStep[0]] = True;

#print it

if nextStep[0]!=nextStep[1]:

print("Parent: "+str(nextStep[1]+1)+" next: "+str(nextStep[0]+1))

### (b) K23-5-6



**Fig 9.** The given graph

The adjacency matrix of the given graph is:

Use the following code to call the class defined in previous section:

A = np.array([[0, 6, 1, 0, 15],

[6, 0, 3, 14, 9],

[1, 3, 0, 10, 0],

[0, 14, 0, 0, 2],

[15, 9, 0, 2, 0]])

a = Prim(A);

a.prim();

The output is:

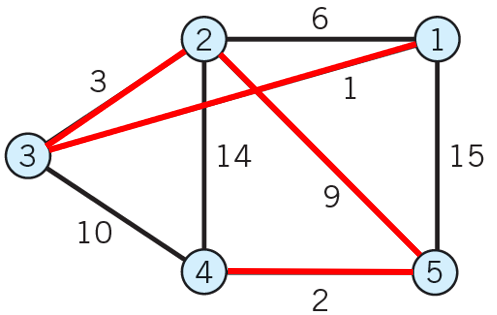
Parent: 1 next: 3

Parent: 3 next: 2

Parent: 2 next: 5

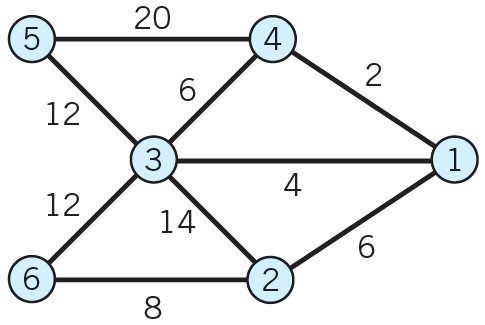
Parent: 5 next: 4

Based on the calculated result, the shortest spanning tree is show as below:



**Fig 10.** The shortest spanning tree by Prim`s algorithm

### (c) K23-5-7



**Fig 11.** The given graph

The adjacency matrix of the given graph is:

Use the following code to call the class defined in previous section:

A = np.array([[0, 6, 4, 2, 0, 0],

[6, 0, 14, 0, 0, 8],

[4, 14, 0, 6, 12,12],

[2, 0, 6, 0, 20, 0],

[0, 0, 12, 20, 0, 0],

[0, 8, 12, 0, 0, 0]])

a = Prim(A);

a.prim();

The output is:

Parent: 1 next: 4

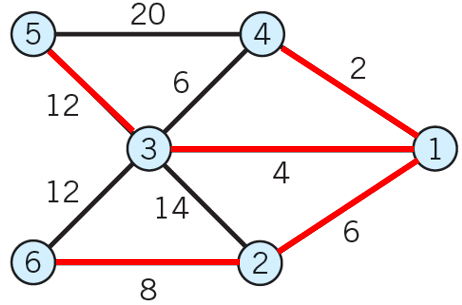
Parent: 1 next: 3

Parent: 1 next: 2

Parent: 2 next: 6

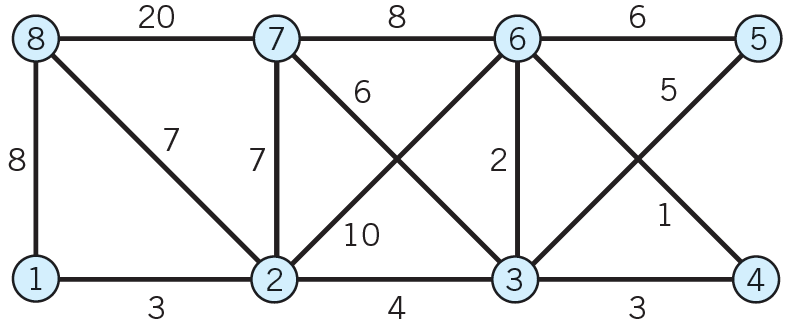
Parent: 3 next: 5

Based on the calculated result, the shortest spanning tree is show as below:



**Fig 12.** The shortest spanning tree by Prim`s algorithm

### (d) K23-5-8



**Fig 13.** The given graph

The adjacency matrix of the given graph is:

Use the following code to call the class defined in previous section:

A = np.array([[0, 3, 0, 0, 0, 0, 0, 8],

[3, 0, 4, 0, 0, 10, 7, 7],

[0, 4, 0, 3, 5, 2, 6, 0],

[0, 0, 3, 0, 0, 1, 0, 0],

[0, 0, 5, 0, 0, 6, 0, 0],

[0, 10, 2, 1, 6, 0, 8, 0],

[0, 7, 6, 0, 0, 8, 0, 20],

[8, 7, 0, 0, 0, 0, 20, 0]])

a = Prim(A);

a.prim();

The output is:

Parent: 1 next: 2

Parent: 2 next: 3

Parent: 3 next: 6

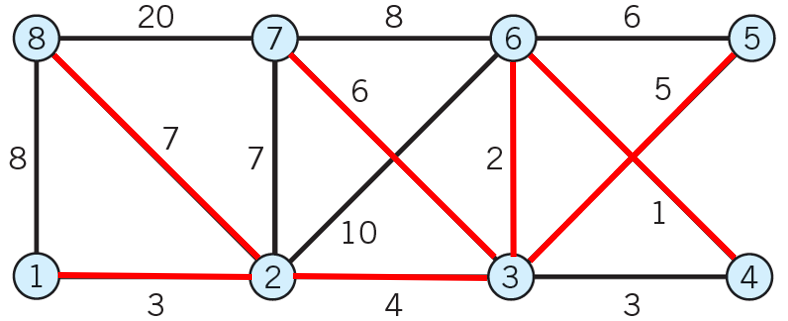
Parent: 6 next: 4

Parent: 3 next: 5

Parent: 3 next: 7

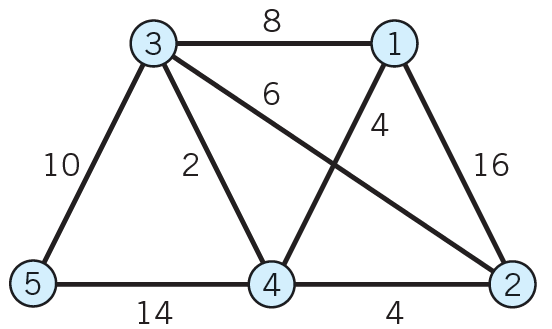
Parent: 2 next: 8

Based on the calculated result, the shortest spanning tree is show as below:



**Fig 14.** The shortest spanning tree by Prim`s algorithm

### (e) K23-5-9



**Fig 15.** The given graph

The adjacency matrix of the given graph is:

Use the following code to call the class defined in previous section:

A = np.array([[0, 3, 0, 0, 0, 0, 0, 8],

[3, 0, 4, 0, 0, 10, 7, 7],

[0, 4, 0, 3, 5, 2, 6, 0],

[0, 0, 3, 0, 0, 1, 0, 0],

[0, 0, 5, 0, 0, 6, 0, 0],

[0, 10, 2, 1, 6, 0, 8, 0],

[0, 7, 6, 0, 0, 8, 0, 20],

[8, 7, 0, 0, 0, 0, 20, 0]])

a = Prim(A);

a.prim();

The output is:

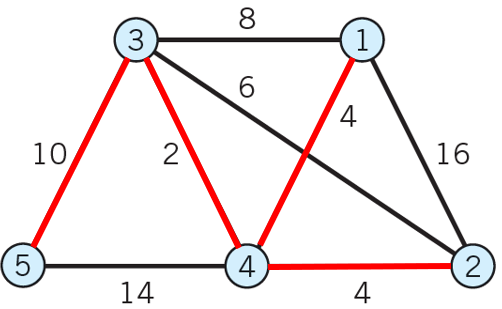
Parent: 1 next: 4

Parent: 4 next: 3

Parent: 4 next: 2

Parent: 3 next: 5

Based on the calculated result, the shortest spanning tree is show as below:



**Fig 16.** The shortest spanning tree by Prim`s algorithm

### Appendix

#### 1. Code of Problem 3(b)

class Vertex:

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

self.name = name

def get\_neighbors(self):

return Vertex.links[self.name];

#save the link in the vertex class

links = {

'A': ['B', 'D', 'G'],

'B': ['E', 'G', 'H'],

'C': ['A', 'H', 'I'],

'D': ['F'],

'E': ['H', 'A', 'C'],

'F': ['G', 'I'],

'G': ['C'],

'H': ['A', 'E'],

'I': ['C', 'J'],

'J': []

}

class Queue:

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

self.queue = []

def next(self):

if(len(self.queue)!=0):

result = self.queue[0]

self.queue.remove(result);

return result;

def append(self,vertex,distance):

result = {"distance":distance,"vertex":vertex}

self.queue.append(result)

class StepTable:

"""Container to remember the steps taken by Moore's algorithm.

"data" is a list of steps, where each step contains the vertex

object, the distance from the origin, and the new neighbors

encountered of that vertex. E.g.,

data = [ (<vertex object>, 0, ['B', 'D']),

(<vertex object>, 1, ['C']),

(<vertex object>, 1, ['F']),

...]

""”

def \_\_init\_\_(self, data=None):

if data is None:

self.data = []

else:

self.data = data

def append(self, vertex, distance):

"""Adds the given vertex object to the step table.

Distance is the distance from the origin."""

neighbors = vertex.get\_neighbors()

new\_neighbors = self.get\_new\_neighbors(neighbors)

self.data.append((vertex, distance, new\_neighbors))

def get\_new\_neighbors(self, neighbors):

"""Compares the vertex names in the list "neighbors"

to the neighbors already contained in steptable.

Returns a (shorter) list of names of neighbors that have

not yet been discovered.

"""

#loop all items in the neighbors

for i in neighbors:

#loop all vertices that in the step table

for item in self.data:

#if already exits in the step talbe, remove it

if item[0].name == i:

neighbors.remove(i)

break

return neighbors

def get\_reverse\_path(self, vertex\_name):

"""Starting at the vertex named "vertex\_name", traces

backwards through the step table to find the shortest

distance to the origin. Note that this should only be

called\*after\*vertex\_name has been discovered.

"""

#initial list with given vertex name

result = [vertex\_name]

#get length of the recorded data

length = len(self.data)

#set the current vertex name

current = vertex\_name

#loop from step table reversely

for i in range(length):

for vertex in self.data:

#find the current vertex in table

if current in vertex[2]:

result.insert(0,vertex[0].name)

current = vertex[0].name

return result

def print(self):

"""Attempts to pretty print the contents of the

step table."""

for row in self.data:

print('{:10s} {:3d} {:s}'.format(row[0].name, row[1], str(row[2])))

'''

\* @name: findPath

\* @description: use Moore`s algorithm to find the shortest path from start to end vertices

\* @param start: the name of the start vertex

\* @param end: the name of the end vertex

\* @return: list, the shortest path from start to end vertices

'''

def findPath(start,end):

#declare the queue

bfs = Queue();

#declare the step table

record = StepTable();

#create the start vertex

start = Vertex(start);

#append this start point into queue

bfs.append(start,0)

#save the this start point into step table

record.append(start,0);

#use Moore`s algorithm or so called bfs

while(bfs.queue!=[]):

#get the front the queue

nextNode = bfs.next()

#get the distance of this point

currentDistance = nextNode["distance"]

#get the neighbors of this point

neighbors = nextNode["vertex"].get\_neighbors();

#loop all neighbors

for item in neighbors:

#use a bool value to see if the vertex has been visited

visited = False

#loop all records in the step table to see if visited

for step in record.data:

if step[0].name == item:

visited = True

break

#if not visited

if(not visited):

#create the vertex

temp = Vertex(item);

#push it into queue

bfs.append(temp,currentDistance+1);

#record this step in step table

record.append(temp,currentDistance+1);

#record.print()

#get the path from start to end

path = record.get\_reverse\_path(end);

#print path

print(path)

return path

findPath('B','F')

#### 2. Code of Problem 3(e)

﻿import requests

from lxml import html

class Vertex:

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

self.name = name

def get\_neighbors(self):

#load the page

page = requests.get(self.name)

#get the tree structure of the urls

tree = html.fromstring(page.content)

#get all links

links = tree.xpath('//a/@href')

result = []

#save all links related to engineering school

for i in links:

if i.startswith('https://www.brown.edu/academics/engineering/'):

result.append(str(i))

if i.startswith('/academics/engineering'):

result.append('https://www.brown.edu'+i)

return result

class Queue:

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

self.queue = []

def next(self):

if(len(self.queue)!=0):

result = self.queue[0]

self.queue.remove(result);

return result;

def append(self,vertex,distance):

result = {"distance":distance,"vertex":vertex}

self.queue.append(result)

class StepTable:

"""Container to remember the steps taken by Moore's algorithm.

"data" is a list of steps, where each step contains the vertex

object, the distance from the origin, and the new neighbors

encountered of that vertex. E.g.,

data = [ (<vertex object>, 0, ['B', 'D']),

(<vertex object>, 1, ['C']),

(<vertex object>, 1, ['F']),

...]

"""

def \_\_init\_\_(self, data=None):

if data is None:

self.data = []

else:

self.data = data

def append(self, vertex, distance):

"""Adds the given vertex object to the step table.

Distance is the distance from the origin."""

neighbors = list(set(vertex.get\_neighbors()))

new\_neighbors = self.get\_new\_neighbors(neighbors)

self.data.append((vertex, distance, new\_neighbors))

def get\_new\_neighbors(self, neighbors):

"""Compares the vertex names in the list "neighbors"

to the neighbors already contained in steptable.

Returns a (shorter) list of names of neighbors that have

not yet been discovered.

"""

#loop all items in the neighbors

for i in neighbors:

#loop all vertices that in the step table

for item in self.data:

#if already exits in the step talbe, remove it

visited = False

if item[0].name == i:

neighbors.remove(i)

visited = True

break

for neighbor in item[2]:

if(i == neighbor):

neighbors.remove(i)

visited = True

break

if(visited):

break;

return neighbors

def get\_reverse\_path(self, vertex\_name):

"""Starting at the vertex named "vertex\_name", traces

backwards through the step table to find the shortest

distance to the origin. Note that this should only be

called\*after\*vertex\_name has been discovered.

"""

#initial list with given vertex name

result = [vertex\_name]

#get length of the recorded data

length = len(self.data)

#set the current vertex name

current = vertex\_name

start = self.data[0][0].name

#loop from step table reversely

for i in range(length):

for vertex in self.data:

#find the current vertex in table

if start == current:

return result

if current in vertex[2]:

result.insert(0,vertex[0].name)

current = vertex[0].name

return result

def print(self):

"""Attempts to pretty print the contents of the

step table."""

for row in self.data:

print('{:10s} {:3d} {:s}'.format(row[0].name, row[1], str(row[2])))

'''

\* @name: findPath

\* @description: use Moore`s algorithm to find the shortest path from start to end vertices

\* @param start: the name of the start vertex

\* @param end: the name of the end vertex

\* @return: list, the shortest path from start to end vertices

'''

def findPath(start,end):

#declare the queue

bfs = Queue();

#declare the step table

record = StepTable();

#create the start vertex

start = Vertex(start);

#append this start point into queue

bfs.append(start,0)

#save the this start point into step table

record.append(start,0);

#use Moore`s algorithm or so called bfs

while(bfs.queue!=[]):

#get the front the queue

nextNode = bfs.next()

currentName = nextNode["vertex"].name

#if find the target then break the while loop

if currentName==end:

break

#get the distance of this point

currentDistance = nextNode["distance"]

#get the neighbors of this point

neighbors = list(set(nextNode["vertex"].get\_neighbors()));

#loop all neighbors

for item in neighbors:

#use a bool value to see if the vertex has been visited

visited = False

#loop all records in the step table to see if visited

for step in record.data:

if step[0].name == item:

visited = True

break

#if not visited

if(not visited):

#create the vertex

temp = Vertex(item);

#push it into queue

bfs.append(temp,currentDistance+1);

#record this step in step table

record.append(temp,currentDistance+1);

#record.print()

#get the path from start to end

path = record.get\_reverse\_path(end);

#print path

print(path)

return path

findPath('https://www.brown.edu/academics/engineering/',\

'https://www.brown.edu/academics/engineering/graduate-study/masters-and-phd-programs')