# **Lab 4**

**Graphs:**

Consider a simple (directed) graph (digraph) having six nodes (A-F) and the following arcs

(directed edges):

A -> B

A -> C

B -> C

B -> D

C -> D

C -> F

D -> C

D -> E

E -> F

F -> C

F -> E

It can be represented by the following Python data structure:

graph = {'A': ['B', 'C'],

'B': ['C', 'D'],

'C': ['D', ’F’],

'D': ['C',’E’],

'E': ['F'],

'F': ['C',’E’]}

This is a dictionary whose keys are the nodes of the graph. For each key, the corresponding

value is a list containing the nodes that are connected by a direct arc from this node. This is

about as simple as it gets (even simpler, the nodes could be represented by numbers instead of

names, but names are more convenient and can easily be made to carry more information,

such as city names).

Let's write a simple function to determine a path between two nodes. It takes a graph and the

start and end nodes as arguments. It will return a list of nodes (including the start and end

nodes) comprising the path. When no path can be found, it returns None. The same node will

not occur more than once on the path returned (i.e. it won't contain cycles). The algorithm

uses an important technique called *backtracking*: it tries each possibility in turn until it finds a

solution.

def find\_path(graph, start, end, path=[]):

path = path + [start]

if start == end:

return path

if start not in graph:

return None

for node in graph[start]:

if node not in path:

newpath = find\_path(graph, node, end, path)

if newpath: return newpath

return None

A sample run of the function find\_path() (using the graph above):

>>> find\_path(graph, 'A', 'D')

['A', 'B', 'C', 'D']

**Example 2:**

######### Directed Weighted Graph and Search ############

class Graph:

def \_\_init\_\_(self, nodes=None, edges=None):

"""Initialize a graph object.

Args:

nodes: Iterator of nodes. Each node is an object.

edges: Iterator of edges. Each edge is a tuple of 2 nodes.

"""

self.nodes, self.adj = [], {}

if nodes != None:

self.add\_nodes\_from(nodes)

if edges != None:

self.add\_edges\_from(edges)

def \_\_len\_\_(self):

"""Returns the number of nodes in the graph.

>>> g = Graph(nodes=[x for x in range(7)])

>>> len(g)

7

"""

return len(self.nodes)

def \_\_contains\_\_(self, x):

"""Return true if a node x is in the graph.

>>> g = Graph(nodes=[x for x in range(7)])

>>> 6 in g

True

>>> 7 in g

False

"""

return x in self.nodes

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

"""Iterate over the nodes in the graph.

>>> g = Graph(nodes=[x for x in range(7)])

>>> [x \* 2 for x in g]

[0, 2, 4, 6, 8, 10, 12]

"""

return iter(self.nodes)

def \_\_getitem\_\_(self, x):

"""Returns the iterator over the adjacent nodes of x.

>>> g = Graph(nodes=[x for x in range(7)], edges=[(1,0), (1,2),

(1,3)])

>>> [x for x in g[1]]

[0, 2, 3]

"""

return iter(self.adj[x])

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

return 'V: %s\nE: %s' % (self.nodes, self.adj)

def add\_node(self, n):

if n not in self.nodes:

self.nodes.append(n)

self.adj[n] = []

def add\_nodes\_from(self, i):

for n in i:

self.add\_node(n)

def add\_edge(self, u, v): # undirected unweighted graph

self.adj[u] = self.adj.get(u, []) + [v]

self.adj[v] = self.adj.get(v, []) + [u]

def add\_edges\_from(self, i):

for n in i:

self.add\_edge(\*n)

def number\_of\_nodes(self):

return len(self.nodes)

def number\_of\_edges(self):

return sum(len(l) for \_, l in self.adj.items()) // 2

class DGraph(Graph):

def add\_edge(self, u, v):

self.adj[u] = self.adj.get(u, []) + [v]

class WGraph(Graph):

def \_\_init\_\_(self, nodes=None, edges=None):

"""Initialize a graph object.

Args: nodes: Iterator of nodes. Each node is an object.

edges: Iterator of edges. Each edge is a tuple of 2 nodes and a weight. """

self.nodes, self.adj, self.weight = [], {}, {}

if nodes != None:

self.add\_nodes\_from(nodes)

if edges != None:

self.add\_edges\_from(edges)

def add\_edge(self, u, v, w):

self.adj[u] = self.adj.get(u, []) + [v]

self.adj[v] = self.adj.get(v, []) + [u]

self.weight[(u,v)] = w

self.weight[(v,u)] = w

def get\_weight(self, u, v):

return self.weight[(u,v)]

class DWGraph(WGraph):

def add\_edge(self, u, v, w):

self.adj[u] = self.adj.get(u, []) + [v]

self.weight[(u,v)] = w

###########################################################################

####

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

pass

**Lab Task:**

1. Change the function find\_path to return a list of all paths (without cycles) instead of the first path it finds.

2. Consider a simple (directed) graph (digraph) having six nodes (A-F) and the

following arcs (directed edges) with respective cost of edge given in parentheses:

A -> B (2)

A -> C (1)

B -> C (2)

B -> D (5)

C -> D (1)

C -> F (3)

D -> C (1)

D -> E (4)

E -> F (3)

F -> C (1)

F -> E (2)

Using the code for a directed weighted graph in Example 2, instantiate an object of DWGraph in \_\_main\_\_, add the nodes and edges of the graph using the relevant functions, and implement a function find\_path() that takes starting and ending nodes as arguments and returns at least one path (if one exists) between those two nodes. The function should also keep track of the cost of the path and return the total cost as well as the path. Print the path and its cost in \_\_main\_\_.