Chapter 4: Graphs

DRAFT

Graph theory

See Alberto Montresor theory here: http://disi.unitn.it/~montreso/sp/slides/06-grafi.pdf (http://disi.unitn.it/~montreso/sp/slides/06-grafi.pdf)

See Graphs on the book (https://interactivepython.org/runestone/static/pythonds/Graphs/toctree.html)

In particular, see :

- Vocabulary and definitions (https://interactivepython.org/runestone/static/pythonds/Graphs/VocabularyandDefinitions.html)

To keep it short, a graph is a set of vertices linked by edges.

Directed graphs

In this worksheet we are going to use so called Directed Graphs (`DiGraph` for brevity), that is graphs that have *directed* edges: each edge can be pictured as an arrow linking source node *a* to target node *b*. With such an arrow, you can go from *a* to *b* but you cannot go from *b* to *a* unless there is another edge in the reverse direction.

- The `DiGraph` is represented as a
- A `DiGraph` for us can also have no edges or no verteces at all.
- In our model, edges simply lik vertices and have no weights
-

1) Building graphs

```
def full_graph(verteces):
    """ Returns a DiGraph which is a full graph with provided verteces list.

        In a full graph all verteces link to all other verteces)
    """

def dag(verteces):
    """ Returns a DiGraph which is DAG made out of provided verteces list

        Provided list is intended to be in topological order.
    """
```

2) Manipulate graphs

```
def reverse(self):
    """ Reverses the direction of all the edges """

def remove_self_loops(self):
    """ Removes all of the self loops """
```

TODO: graph union, intersection, ...

3) Query graphs

Today we query graphs the "Do it yourself" way with Depth First Search (DFS) or Breadth First Search (BFS).

If you have a big graph and complex query needs, there are off-the-shelves query languages and databases (example: Cypher and Neo4J)

3.1) Play with dfs and bfs

Create small graphs (like linked lists a->b->c, triangles, mini-full graphs, trees) and try to predict the visit sequence (verteces order, with discovery and finish times) you would have running a dfs or bfs. Then write tests that assert you actually get those sequences bwhen running provided dfs and bfs

4) Do cool stuff with theory

- find connected components
- determine if a graph is acyclic
- find node distances

In [2]:

```
import unittest
from pprint import PrettyPrinter
from Queue import Queue
import traceback

pp = PrettyPrinter()

class VertexLog:
    """ Represents the visit log a single DiGraph vertex

        This class is very simple and doesn't even have getters methods.

        You can just access fields by using the dot:

            print vertex_log.discovery_time

        and set them directly:

            vertex_log.finish_time = 5
```

```
            vertex_log.finish_time = 5

        If you want, an instances you can set your own fields:

            vertex_log.my_own_field = "whatever"
    """

    def __init__(self, vertex):
        self.vertex = vertex
        self.discovery_time = -1
        self.finish_time = -1
        self.parent = None

    def __repr__(self):
        return pp.pformat(vars(self))

class Visit:
    """ The visit of a DiGraph visit sequence.

    """

    def __init__(self):
        """ Creates a Visit """

        self._logs = {}


    def is_discovered(self, vertex):
        """ Returns true if given vertex is present in the log and
            has discovery_time != -1
        """
        return vertex in self._logs and self._logs[vertex].discovery_time != -1

    def log(self, vertex):
        """ Returns the log of the given vertex.

            If there is no existing log, a new one will be created and returned
        """

        if not vertex in self._logs:
            self._logs[vertex] = VertexLog(vertex)

        return self._logs[vertex]

    def logs(self,
             sort_by=lambda log: log.discovery_time,
             descendant=False,
             get_all=False):
        """ Returns an array with sequence of discovered VertexLogs, sorted by disco
very time.

            Optionally, they can be sorted by:
            - a custom field using 'sort_by' parameter
            - in descendent order with 'descendant' parameter.

            By default only discovered vertex logs are returned:
             to get all, use get_all=True
        """
        if get_all:
            ret = self._logs.values()
        else:
            ret = filter(lambda log: log.discovery_time > -1, self._logs.values())

        ret.sort(key= sort_by, reverse= descendant)
        return ret
```

```
    def verteces(self,
                 sort_by=lambda log: log.discovery_time,
                 descendant=False,
                 get_all=False):
        """ Returns an array with sequence of the discovered VertexLogs, sorted by d
iscovery time.

            Optionally, they can be sorted by:
            - a custom field using 'sort_by' parameter
            - in descendent order with 'descendant' parameter.

            By default only discovered vertex logs are returned:
             to get all, use get_all=True
        """
        return map(lambda vertex_log:vertex_log.vertex,

                    self.logs(sort_by=sort_by,
                              descendant=descendant,
                              get_all=get_all))


    def last_time(self):
        """ Return the maximum time found among discovery and finish times.

            If no node was visited, returns zero.
        """

        max_time = 0
        for log in self._logs.values():
            if log.discovery_time > max_time:
               max_time = log.discovery_time
            if log.finish_time > max_time:
               max_time = log.finish_time
        return max_time


class DiGraph:
    """ A simple graph data structure, represented as a dictionary of adjacency list
s

        Verteces can be of any type, to keep things simple in this data model they c
oincide with their labels.
        Adjacency lists hold the target verteces.
        Attempts to add duplicate targets will be silently ignored.

        For shorthand construction, see separate dig() function
    """

    def __init__(self):
        self._edges = {}

    def add_vertex(self, vertex):
        """ Adds vertex to the DiGraph. A vertex can be any object.

            If the vertex already exist, does nothing.
        """
        if vertex not in self._edges:
            self._edges[vertex] = []

    def verteces(self):
        """ Returns a set of the graph verteces. Verteces can be any object. """

        # Note dict keys() return a list, not a set. Bleah.
        # See http://stackoverflow.com/questions/13886129/why-does-pythons-dict-keys
-return-a-list-and-not-a-set
```

```
-return-a-list-and-not-a-set
        return set(self._edges.keys())

    def has_vertex(self, vertex):
        """ Returns true if graph contains given vertex. A vertex can be any object.
 """
        return vertex in self._edges

    def remove_vertex(self, vertex):
        """ Removes the provided vertex  and returns it

            If the vertex is not found, raises an Exception.
        """

        if not vertex in self._edges:
            raise Exception("Couldn't find vertex:" +str(vertex))

        for key in self.verteces:
            self.verteces[key].remove(vertex)

        return self.verteces.pop(vertex)

    def add_edge(self, vertex1, vertex2):
        """ Adds an edge to the graph, from vertex1 to vertex2

            If verteces don't exist, raises an Exception.
            If there is already such an edge, exits silently.
        """

        if not vertex1 in self._edges:
            raise Exception("Couldn't find source vertex:" + str(vertex1))

        if not vertex2 in self._edges:
            raise Exception("Couldn't find target vertex:" + str(vertex2))

        if not vertex2 in self._edges[vertex1]:
            self._edges[vertex1].append(vertex2)

    def __str__(self):
        """ Returns a string representation like the following:

            >>> print gr('a',['b','c', 'd'],
                         'b', ['b'],
                         'c', ['a'])

            a: [b,c]
            b: [b]
            c: [a]
            d: []

        """

        if (len(self._edges) == 0):
            return "DiGraph()"

        max_len=0

        for source in self._edges:
            max_len = max(max_len, len(str(source)))

        strings = []

        for source in self._edges:
            strings.append(str(source).ljust(max_len))
            strings.append(': ')
            strings.append(str(self._edges[source]))
```

```
            strings.append(str(self._edges[source]))

            strings.append('\n')

        return ''.join(strings)

    def adj(self, vertex):

        if not vertex in self._edges:
            raise Exception("Couldn't find a vertex " + str(vertex))

        return self._edges[vertex]

    def __eq__(self, other):

        if not isinstance(other, DiGraph):
            return False

        if self.verteces() != other.verteces():
            return False

        for source_vertex in self._edges:
            if self._edges[source_vertex] != other._edges[source_vertex]:
                return False

        return True

    def is_empty(self):
        """  A DiGraph for us is empty if it has no verteces and no edges """

        return len(self._edges) == 0

    def dfs(self, source, visit=None):
        """ Performs a simple depth first search on the graph

            Returns a Visit of the visited nodes. If the graph is empty, raises an E
xception.
            Optionally, you can pass the initial visit trace.
        """

        if self.is_empty():
            raise Exception("Cannot perform DFS on an empty graph!")

        if visit == None:
            visit = Visit()

        # we just discovered the vertex
        source_log = visit.log(source)
        source_log.discovery_time = visit.last_time() + 1

        for neighbor in self.adj(source):
            if not visit.is_discovered(neighbor):

                visit.log(neighbor).parent = source

                self.dfs(neighbor, visit)

        source_log.finish_time = visit.last_time() + 1

        return visit


    def bfs(self, source):
        """ Performs a simple breadth first search in the graph, starting from
            provided source vertex.
```

```
            Returns a Visit of the discovered nodes.
            NOTE: it stores discovery but not finish times.

            If source is not in the graph, raises an Exception

        """

        if self.is_empty():
            raise Exception("Cannot perform BFS on an empty graph!")

        if not source in self.verteces():
            raise Exception("Can't find vertex:" + str(source))

        visit = Visit()

        queue = Queue()
        queue.put(source)

        while not queue.empty():
            vertex = queue.get()

            if not visit.is_discovered(vertex):
                # we just discovered the node
                visit.log(vertex).discovery_time = visit.last_time() + 1

                for neighbor in self.adj(source):
                    neighbor_log = visit.log(neighbor)
                    if neighbor_log.parent == None:
                        neighbor_log.parent = vertex
                    queue.put(neighbor)

        return visit




def str_compare_digraphs(dg1, dg2):
    """ Returns a string representing a comparison side by side
        of the provided digraphs

    """

    if (dg1 == None) ^ (dg1 == None):
        return "At least one graph is None ! " +"\n\n Graph 1: " + str(dg1)  +"\n\n
Graph 2: " + str(dg2)

    max_len1 = 0
    for source in dg1.verteces():
        max_len1 = max(max_len1, len(str(source)))

    max_len2 = 0
    for source in dg2.verteces():
        max_len2 = max(max_len2, len(str(source)))


    strings = []


    common_edges = set(dg1.verteces()) & set(dg2.verteces())

    all_edges = set(dg1.verteces()).union( dg2.verteces())

    different_edges = all_edges - common_edges

    if len(different_edges > 0):
```

```
    if len(different_edges > 0):
        vs = list(common_edges)
        vs.extend(different_edges)
    else:
        vs = dg1.verteces()

    strings = []

    for vertex in vs:

        strings.append(vertex)
        strings.append(': ')

        if vertex in dg1.verteces():
            strings.append(str(dg1.adj(vertex)).ljust(max_len1 + 4))
        else:
            strings.append(" " * (max_len1 + 4))

        if vertex in dg2.verteces():
            strings.append(dg2.adj(vertex))
        else:
            strings.append(" " * (max_len2 + 4))

        if (dg1.adj(vertex) != dg2.adj(vertex)):
            strings.append("  <---- DIFFERENT ! ")

        strings.append("\n")

    return ''.join(strings)



def dig(*args):
    """ Shorthand to construct a DiGraph with provided arguments

        To use it, provide source vertex / target vertex pairs like in the following
 examples:

        >>> print dig()

        DiGraph()

        >>> print dig('a',['b','c'])

        a: [b,c]
        b: []
        c: []

        >>> print dig('a',['b','c'],
                      'b', ['b'],
                      'c', ['a'])

        a: [b,c]
        b: [b]
        c: [a]

    """

    g = DiGraph()

    if len(args) % 2 == 1:
        raise Exception("Number of arguments must be even! You need to provide"
                      + " vertex/list pairs like 'a',['b', 'c'], b, ['d'], ... !")

    i = 1
    for a in args:
```

```
    for a in args:

        if i % 2 == 1:
            vertex = a
            g.add_vertex(vertex)

        else:
            try:
                iter(a)
            except TypeError:
                raise Exception('Targets of ' + str(vertex) + ' are not iterable: '
+ str(a) )
            for target in a:
                if not g.has_vertex(target):
                    g.add_vertex(target)
                g.add_edge(vertex, target)
        i += 1

    return g



def gen_graphs(n):
    """ Returns a list with all the possible 2^(n^2) graphs of size n

        Verteces will be identified with numbers from 1 to n
    """

    def gen_bits(n):
        """  Generates a sequence of 2^(n^2) lists, each of n^2 0 / 1 ints  """

        bits = n*n;
        nedges = 2**bits

        ret = []
        for i in range(0, nedges):

            right = [int(x) for x in bin(i)[2:]]
            lst = ([0] * (bits - len(right)))
            lst.extend(right)

            ret.append(lst)
        return ret

    if n == 0:
        return [DiGraph()]

    i = 0

    ret = []

    for lst in gen_bits(n):

        g = DiGraph()
        for j in range(1, n+1):
            g.add_vertex(j)

        source = 0
        for b in lst:
            if i % n == 0:
                source += 1
            if b:
                g.add_edge(source, (i % n) + 1)
            i += 1
        ret.append(g)
    return ret
```

```
    return ret

def gen_list(n):
    """ Generates a graph of n verteces displaced like a
        monodirectional list:  1 -> 2 -> 3 -> ... -> n
    """

    if n == 0:
        return DiGraph()

    g = DiGraph()
    for j in range(1, n+1):
        g.add_vertex(j)

    for k in range(1, n):
            g.add_edge(k, k+1)

    return g

GRAPHS_3 = gen_graphs(3)


class VisitTest(unittest.TestCase):

    def test_log(self):
        """ Checks it doesn't explode with non-existing verteces """
        self.assertEqual(-1, Visit().log('a').discovery_time)
        self.assertEqual(-1, Visit().log('a').finish_time)

    def test_verteces(self):
        self.assertEqual([], Visit().verteces())

        visit = Visit()
        visit.log('a')
        self.assertEqual([], visit.verteces())
        self.assertEqual(['a'], visit.verteces(get_all=True))
        visit.log('a').discovery_time = 1
        self.assertEqual(['a'], visit.verteces())
        visit.log('b').discovery_time = 2
        self.assertEqual(['a', 'b'], visit.verteces())
        #  descendant=False, get_all=False):
        self.assertEqual(['b', 'a'], visit.verteces(descendant=True))
        self.assertEqual(['b', 'a'], visit.verteces(descendant=True))

        visit.log('a').finish_time = 4
        visit.log('b').finish_time = 3
        self.assertEqual(['b', 'a'], visit.verteces(sort_by=lambda log:log.finish_ti
me))

class DiGraphTest(unittest.TestCase):

    def assertDiGraphEqual(self, dg1, dg2):
        if not dg1 == dg2:
            raise AssertionError("Graphs are different: \n\n" + str_compare_digraphs
 )

    def assertSubset(self, set1, set2):
        """ Asserts set1 is a subset of set2 """

        if not set1.issubset(set2):
            raise AssertionError(str(set1) + " is not a subset of " + str(set2))

    def raise_graph(self, exception, graph, visit):
        """ Emulates reraising an exception for a given graph visit """

        raise Exception(traceback.format_exc(exception))
```

```
        raise Exception(traceback.format_exc(exception)
            +"\n Failed graph was: \n" + str(graph)
            +"\n Failed graph visit was: \n" + pp.pformat(visit.logs()))

    def test_str(self):
        self.assertTrue("DiGraph()" in str(dig()))
        self.assertTrue("x" in str(dig('x',['y'])))
        self.assertTrue("y" in str(dig('x',['y'])))
        self.assertEquals(set(['x','y']), dig('x',['y']).verteces())
        self.assertEquals(set(['x','y','z','w', 'z']),
                          dig('x',['y'], 'z', ['w','x']).verteces())

    def test_gen_list(self):
        self.assertEquals(gen_list(0), dig())
        self.assertEquals(gen_list(1), dig(1,[]))
        self.assertEquals(gen_list(3), dig(1,[2],2,[3]))

    def test_gen_graphs(self):

        gs0 = gen_graphs(0)
        self.assertEquals(1, len(gs0))
        self.assertTrue(dig() in gs0)

        gs1 = gen_graphs(1)

        self.assertEquals(2, len(gs1))
        self.assertTrue(dig(1, []) in gs1)

    def test_assert_dig(self):

        self.assertDiGraphEqual(dig(), dig())

        with self.assertRaises(Exception):
            self.assertDiGraphEqual(dig(), dig('a',[]))

    def test_dfs(self):

        with self.assertRaises(Exception):
            self.assertEquals([], dig().dfs('a'))

        self.assertEquals(['a'], dig('a',[]).dfs('a').verteces())

        for g in GRAPHS_3:
            try:
                visit = g.dfs(1)
                self.assertLessEqual(visit.last_time(), 3*2)
                self.assertEquals(visit.log(1).finish_time,
                                  visit.last_time())
            except Exception as e:
                self.raise_graph(e, g, visit)


    def test_bfs(self):

        with self.assertRaises(Exception):
            self.assertEquals([], dig().bfs('a'))

        self.assertEquals(['a'], dig('a',[]).bfs('a').verteces())

        for g in GRAPHS_3:
            try:
                visit = g.bfs(1)
                self.assertSubset(set(visit.verteces()), g.verteces() )

                self.assertLessEqual(visit.last_time(), 3)
            except Exception as e:
```

```
            except Exception as e:
                self.raise_graph(e, g, visit)
```

In [3]:

```
algolab.run(VisitTest)
```

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.002s

OK
```

In [4]:

```
algolab.run(DiGraphTest)
```

```
......
----------------------------------------------------------------------
Ran 6 tests in 0.075s

OK
```

In [5]:

```

```