2019

SOFTENG 364: Lab 5 Python Programming and Routing

Craig Sutherland

Contents

Overview

The objectives of this lab are:

1. To become familiar with the syntax of Python and key data structures in its Standard Library. We'll use Python in subsequent labs and in Assignment 2, and it may well be useful in a project in 2019 or beyond.
2. To reinforce our discussion of routing algorithms via programming and exposure to two relevant APIs:
   - The NetJSON data interchange format
   - The NetworkX library for network algorithms and visualization
3. To become acquainted with an implementation Dijkstra's algorithm that we'll extend in Assignment 2.
4. To become acquainted with asynchronous programming, which we'll employ in Assignment 2.

To receive credit for completing the worksheet, please complete the Lab 5 Quiz on Canvas. You'll receive feedback immediately afterwards, and may choose to redo the quiz if you wish.

Several activities on the lab worksheet are framed as "questions", but responses needn't be submitted (i.e. the on-line quiz is the only submission required). Nonetheless, please don't hesitate to speak to a member of the 364 team during the lab if you are unsure of what a suitable response might be.

Conventions

- Lab tasks are marked with bullets, like this.

Notes have a blue marker on the edge

Preparation

The software we need this week is installed in the Engineering computer labs. If you're working on your own PC, please install Anaconda for Python 3 (https://www.anaconda.com/download/).

The next three steps are required on a lab PC or your own laptop:

1. Launch the Anaconda Prompt.
2. Activate a new Python 3.6 environment using conda.
3. Install the Python modules networkx (https://networkx.github.io/) and gevent (http://www.gevent.org/).

Linux & MacOS users need replace activate with source activate; please see https://conda.io/docs/user-guide/tasks/manage-environments.html#activating-an-environment for details.

```
> conda create -n softeng364python3 python=3.6
> activate softeng364python3
> conda install networkx gevent
```

Launch Spyder - the Python IDE shipped with Anaconda and set its working directory to your preferred location using Spyder's File Explorer tab or address bar.

Create a new Python script called e.g. softeng364lab5.py.

As you proceed through the worksheet, you can append new code snippets and re/execute them by clicking Run file (F5) or by typing %run softeng364lab5.py at Spyder's ipython command prompt.

Spyder provides integrated debugging; however, if you prefer a different editor, you are welcome to use it.

You may like to make a new git repository in which to track your SOFTENG 364 lab- and assignment work.

Assessment

To receive credit for completing the worksheet, please complete the Lab 5 Quiz on Canvas. You'll receive feedback immediately afterwards, and may choose to redo the quiz if you wish.

This worksheet and the associated quiz comprise Lab 5 and contribute 1% to the final mark. The due date (for the quiz) is the 5pm, Thursday 9th May.

Graph representation in JSON and Python

NetJSON is a proposed interchange/file format for network entities.

- Visit netjson.org and briefly (<3 minutes) consider the following questions:
  - What is JSON?
  - Which primitive types does JSON support? Which data structures?
  - What types of network-related entities does NetJSON support? Which one is used to encode network graphs?
  - What identifiers are available for nodes?
  - How are attributes associated with nodes and links?
- Encode the network below in NetJSON format, including its link costs and the following node coordinates. Save your file as KuroseRoss5-15.json.

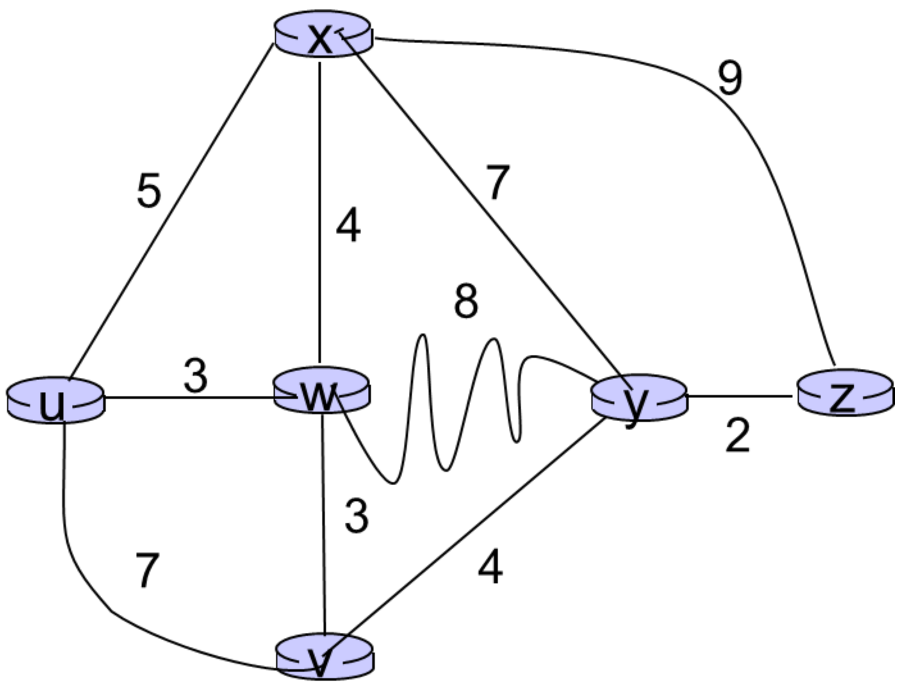

Figure 1: Network example

```
{
    "u": [0.0, 0.0],
    "v": [1.0, -1.0],
    "w": [1.0, 0.0],
    "x": [1.0, 1.0],
    "y": [2.0, 0.0],
    "z": [3.0, 0.0]
}
```

The Python Standard Library includes a module, json, that provides functions (json.load and json.dump) for de/serialization of JSON to/from Python data structures.

- Use the following snippet of code to deserialize your NetJSON file; open is a Standard Library function.

```python
import json
import os
from pprint import pprint  # "pretty print"
filename = os.path.join('.', 'KuroseRoss5-15.json')  # modify as required
netjson = json.load(open(filename))
pprint(netjson)
```

- The resulting value is a Python dictionary: Python's anaoplog of HashMap in Java or std::map in C++.

```
>>> print(type(netjson))
<class 'dict'>
>>> dir(netjson)  # "what members?"
```

- Execute each of the following commands and discuss any queries with your neighbour.

```
>>> netjson['type']  # retrieves value associated with field 'type'
'NetworkGraph'

>>> pprint(netjson['nodes'])  # this field is a list of dicts
[{'id': 'u', 'properties': {'name': 'u', 'pos': [0.0, 0.0]}},
 {'id': 'v', 'properties': {'name': 'v', 'pos': [1.0, -1.0]}},
 {'id': 'w', 'properties': {'name': 'w', 'pos': [1.0, 0.0]}},
 {'id': 'x', 'properties': {'name': 'x', 'pos': [1.0, 1.0]}},
 {'id': 'y', 'properties': {'name': 'y', 'pos': [2.0, 0.0]}},
 {'id': 'z', 'properties': {'name': 'z', 'pos': [3.0, 0.0]}}]

>>> netjson['nodes'][1]['properties']['name']  # "Do we expect 'u' or 'v'?"
'v'
```

While this dict allows us full programmatic access to the graph, many of the things we'd want to do are already available in NetworkX (https://networkx.github.io/): A comprehensive package of Python classes and function for computing with graphs.

Network representation with NetworkX

The following code snippet converts our `dict` representation into a `networkx.Graph` with the original node- and edge attributes. Spend a few minutes (<5) investigating the sub-expressions in this snippet and discuss them with your neighbour.

```python
import networkx as nx  # saves typing later on
graph = nx.Graph()
graph.add_nodes_from((
    (node['id'], node['properties'])  # node-attributes
        for node in netjson['nodes']))
graph.add_edges_from((
    (link['source'], link['target'], {'cost': link['cost']})  # source-target-attributes
        for link in netjson['links']))

for node, data in graph.nodes(data=True):
    pprint((node, data))

for source, target, data in graph.edges(data=True):
    pprint((source, target, data))  # edges & attributes

for node in graph:
    pprint((node, dict(graph[node])))  # neighbours
```

The resulting `networkx.Graph` contains the same information as the original struct, but all of NetworkX's functions are now directly applicable.

- Replace `graph = nx.Graph()` with `graph = nx.DiGraph()` and re-execute the snippets above.
  - What is the impact on node neighbours?
  - Are twice as many edges actually stored in Graph?
  - Are link attributes duplicated in Graph?

Network visualization with NetworkX

Visualize the graph and its node- and edge attributes, using the specified node coordinates, as follows:

```
node_positions = nx.get_node_attributes(graph, name='pos')
edge_label_positions = nx.draw_networkx_edge_labels(
        graph,
        pos=node_positions,
        node_labels=nx.get_node_attributes(graph, name='name'),
        edge_labels=nx.get_edge_attributes(graph, name='cost'))
nx.draw_networkx(graph, pos=node_positions)
```

When node coordinates are not already available (ours were just estimated!), one can employ automatic graph drawing algorithms.

Duplicate the preceding snippet and use `spring_layout` as follows.

```
node_positions = nx.spring_layout(graph)
# Copy preceding snippet from second line
```

Before you continue, have a quick look at some of the other algorithms available in NetworkX's drawing layout module and their parameters (https://networkx.github.io/documentation/stable/reference/drawing.html#module-networkx.drawing.layout).

Least-cost paths with NetworkX

NetworkX provides a large collection of algorithms for working on instances of Graph and DiGraph.

- Scan the list of functions in NetworkX's Algorithms module and briefly (<5 minutes) discuss any that look familiar or interesting - especially if you've met them outside of the context of SOFTENG 250. (https://networkx.github.io/documentation/stable/reference/algorithms/index.html)

We're presently concerned with least-cost paths.

Note: while Kurose & Ross distinguish between **least-cost** paths and **shortest** paths, NetworkX does not.

- Scan the list of algorithms provided for least-cost paths and contrast the following interfaces with your neighbour: Why are they all provided? Which is the minimal interface that we'd need to compute a forwarding table? (https://networkx.github.io/documentation/stable/reference/algorithms/shortest_paths.html)

  Interface
  path()
  paths()

length()
distance()
predecessor()

- Use one of these functions to compute the **predecessor** map and the **distances** of the least-cost paths for our network KuroseRoss5-15. Satisfy yourself that the outputs match those that we computed on Slide 5-15, as shown below.

Note: you'll need to use Graph rather than DiGraph to reproduce these results.

```
>>> pprint(D)  # distances from source 'u'
{'u': 0, 'v': 6, 'w': 3, 'x': 5, 'y': 10, 'z': 12}

>>> pprint(p)  # predecessor map
{'u': [], 'v': ['w'], 'w': ['u'], 'x': ['u'], 'y': ['v'], 'z': ['y']}
```

- Use `networkx.convert.from_dict_of_lists()` to generate the **least-cost path tree** from your predecessor list, as follows. Ensure that the edge list is consistent with Slide `5-15`.

```
>> sp_tree = nx.convert.from_dict_of_lists(p).edges()
>> print(sp_tree)
[('u', 'w'), ('u', 'x'), ('v', 'w'), ('v', 'y'), ('y', 'z')]
```

- Hence, visualize the least-cost path tree (by highlighting the relevant edges of the original graph) using `nx.draw_networkx_edges`. This function has many optional parameters: We need only specify only a few. Again, ensure that the result is consistent with Slide `5-15`.

```
nx.draw_networkx_edges(
        graph,
        pos=node_positions,
        edgelist=sp_tree,
        edge_color='r',
        width=3)
```

Implementation of Dijkstra's algorithm

The pseudocode on Slide 19 of Lecture 1 might be translated into Python as follows:

```
def dijkstra_5_14(graph, source, weight='weight'):
    """
    Shortest paths via Dijkstra's algorithm,
    consistent with pseudocode on Slide 5-14.
    """
    import math

    # Definitions consistent with Kurose & Ross
    u = source
    def c(x, y):
        return graph[x][y][weight]
    N = frozenset(graph.nodes())
    NPrime = {u}  # i.e. "set([u])"
    D = dict.fromkeys(N, math.inf)

    # Initialization
    for v in N:
        if graph.has_edge(u, v):
            D[v] = c(u, v)
    D[u] = 0  # over-write inf entry for source

    # Loop
    while NPrime != N:
        candidates = {w: D[w] for w in N if w not in NPrime}
        w, Dw = min(candidates.items(), key=lambda item: item[1])
        NPrime.add(w)
        for v in graph[w]:
            if v not in NPrime:
                DvNew = D[w] + c(w, v)
                if DvNew < D[v]:
                    D[v] = DvNew
    return D
```

- Compare each line to the original pseudocode and discuss the following new constructs with your neighbour; trust your intuition and consult the documentation (or a 364 team member) if you are still unsure.
    - Initialization of D using dict.fromkeys() and math.inf
    - Initialization of the set of nodes N as a set/frozenset. Why is this necessary? "Isn't graph.nodes() already a set in the mathematical sense?"
    - Initialization of NPrime as a set.
    - Initialization of candidates using familiar set-builder notation.
    - Using an anonymous function (lambda expression) to map each key-value pair to its value.
    - Using the built-in function min to find a key-value pair with the smallest value.

Several components of Assignment 2 relate to this function:

1. Making the (small) modifications necessary to compute the predecessor list, which was not considered in the pseudocode.
2. Writing a function to compute a forwarding table for the source node from the predecessor list.
3. Discussing possible changes that could improve efficiency.
4. Generalizing the implementation to support widest-path- and minimax routing problems.

Note: more information about these and other parts of the assignment will be made available.

Asynchronous programming with `gevent`

Note: the lab quiz does not refer to anything in this section: Please complete the quiz first if you are running short of time.

Implementations of distance-vector algorithms are distributed and asynchronous.

Note: As part of Assignment 2, we will implement the asynchronous Bellman-Ford using a co-routines library. In this final set of lab activities, we'll become acquainted with the problem and gevent.

The following program illustates the meaning of a/synchronous execution; gevent is a Python library that facilitates asynchronous execution.

```
# This program is adapted from the GEvent tutorial:
# http://sdiehl.github.io/gevent-tutorial/
import gevent
import time

num_tasks = 5
```

```python
def now():
    return time.perf_counter()

def one_task(pid):
    # "pid" is "process identifier", a number
    expected = 1.0  # seconds
    print('{}:  "Working" for {:f} sec... '.format(pid, expected))
    start_time = now()
    gevent.sleep(seconds=expected)  # "hard work" :)
    actual = now() - start_time
    print('{}: Finished after {:f} sec'.format(pid, actual))

def run_timed(fun, *args, title="Running..."):
    print(title)
    start_time = now()
    fun(*args)
    print('Required: {:f} sec'.format(now() - start_time))

def run_tasks_synchronously():
    for pid in range(num_tasks):
        one_task(pid)

def run_tasks_asynchronously():
    threads = [gevent.spawn(one_task, pid) for pid in range(num_tasks)]
    gevent.joinall(threads)

run_timed(run_tasks_synchronously, title="Synchronous...")
run_timed(run_tasks_asynchronously, title="Asynchronous...")
```

- Study the code, execute it, and discuss what follows with your neighbour:
  - What does range() return? Run list(range(5)) to check.
  - Is it clear that run_tasks_synchronously and run_tasks_asynchronously complete identical sets of "tasks" (each task requiring about 1 second)?
  - (Why do actual times and expected times not match exactly?)
  - Why is run_tasks_synchronously around num_tasks-times slower run_tasks_asynchronously? What is happening in gevent.sleep()?

We see that gevent.sleep() is **non-blocking**: This means that the active thread t of execution yields control to another thread (while it "works"/sleeps), as opposed to hogging the processor during this time; gevent promises to return control to t as soon as possible after the "work"/sleep is finished.

- Discuss with your neighbour: "*In a real application, each thread would actually do something useful (as opposed to sleep, which requires no processing). Wouldn't yielding control to another thread mean that nothing useful could be done in the interim*?" Hint: Which tasks are **not** performed by your CPU?

We may conclude that asynchronous execution can potentially provide useful speed-ups when independent tasks utilize different computing resources e.g. processing on the CPU and network communication.

Note: as part of Assignment 2, we'll implement the asynchronous Bellman-Ford iteration described in the slides using gevent.

End of lab worksheet: Please don't forget to complete the lab quiz.