Data Structures and Algorithms with Python

Workshop 4

In this workshop you'll implement graph search algorithms in Python. The workshop has three parts:

1. Graphs and breadth-first search
2. Small worlds and Kevin Bacon
3. Challenge: depth-first search

You can find skeleton code for the exercises in the Python file on the Hub — open this file in Spyder and you're ready to go.

1 Graphs and breadth-first search

In the previous workshop, you began to explore the world of object-oriented programming. You learned to represent new types of data (such as pocket monsters, nodes, and queues) as Python classes. This may inspire you to think of elements of graph problems similarly as classes. For example, we might think of graph nodes and edges as classes. A node, or example, would have a name, while and edge would be connected to two nodes.

```
1 class Node():
2     def __init__(self,name):
3         self.name = str(name)
4     def getName(self):
5         return self.name
6     def __str__(self):
7         return self.name
```

```
class Edge():
    """ Has a source node and a destination node
    """
    def __init__(self,source,dest):
        self.source = source
        self.dest = dest
    def getSource(self):
        return self.source
    def getDestination(self):
        return self.dest
    def __str__(self):
        return str(self.source) + '->' + str(self.dest)
```

This is a very useful way of thinking in abstractions, especially if our nodes and edges become more complicated objects (like our pocket monsters). This is how the Guttag book presents graph problems. But for clearly defined and fairly specific applications, such as breadth-first search, using classes may also be a bit burdensome – eg a node has only a name here, no other properties. So we'll instead look at a slightly different graph implementation without using the above classes.

We'll follow the *adjacency list* graph specification you learned in the lecture. More specifically, we'll store the edges of a graph in a dictionary such that the keys of the dictionary are the nodes, and the values are a set of edges from that node (to its children nodes). So for example, the following dictionary would define the adjacency list of a cyclical graph with four nodes.

```
testDict = {'a':{'b'}, 'b':{'c'}, 'c':{'d'}, 'd':{'a'}}
```

We'll use a class, `Digraph`, similar to the one in your textbook, to represent such directed graphs. To implement an undirected graph, we'll similarly use the class `Graph`. These classes have been implemented in the Python file `dsap_w4_graphs`, which you'll need to `import` to your main Python code file as specified in the code in the skeleton code in `dsap_w4.py`.

For the purposes of this workshop, you can just use the functions provided in the classes without looking at the implementation details, see below. But if you want to go further into object-oriented

programming, you can also inspect the code and see how these classes work. [1]

Graph/Digraph methods to use:

- `g = Graph()` `g = Digraph()` – create a new empty undirected or directed graph
- `g.addEdge(v,w)` – adds an edge between the two specified nodes (if it does not already exist). If the nodes don't exist in the graph, adds them to the graph
- `g.childrenOf(v)` – returns the set of edges starting from `v` that you can then iterate through
- `print(g)` – prints the graph edges

With these methods, we're ready to start implementing search algorithms. We've seen that the breadth-first search (BFS) algorithm allows us to find the shortest distances between two nodes in a (unweighted) graph. You'll implement BFS, and apply it to a large real-world dataset in the next section. The algorithm "recipe" is described below.

Algorithm: BFS(graph G, start vertex s)

- **Initialize**: mark all nodes unexplored except s explored
- Use Q = queue data structure, add s to Q
- **Main loop**: While Q is not empty:
  - Remove the first node of Q and call it v
  - For each edge (v, w): if w unexplored:
    - Mark w explored
    - Add w to Q

The challenge problem in the previous workshop asked you to implement a queue data structure. This now proves very useful as BFS calls for us to use a queue to store the nodes we can directly reach from already visited nodes. For your convenience, a `Queue` implementation (which uses the class `QueueNode`) is provided in the Python file `dsap_w4_graphs`. Similarly to graphs, you can simply rely on the methods implemented in the file:

[1] Specifically, see how the `Graph` class is implemented as a *subclass* of `Digraph`. This is using the important OOP concept of *inheritance*, where you can create a new class based on another existing class, and just override some of its methods or attributes. Here, an undirected graph is just a directed graph with edges in two directions so it has pretty much the same characteristics.

- `q = Queue()` – creates a new empty queue
- `q.enqueue(v)` – adds `v` to end of queue
- `w = q.dequeue()` – returns the first item of queue (here assigns it to `w`)
- `q.isEmpty()` – returns `True` if the queue is empty, `False` otherwise

The file `dsap_w4` contains a skeleton implementation of the BFS algorithm. You start with a graph and a starting node, and keep track of a queue of reachable unexplored nodes and a dictionary of explored nodes. You'll need to complete the algorithm using the main loop steps above. When completing the algorithm, it may be useful to look at the skeleton code and the algorithm side-by-side, and see how the initialization steps of the algorithm translate into Python. You'll then need to use the graph and queue methods listed above as appropriate to complete the algorithm. Comparing the algorithm recipe and the methods, which graph and queue methods will you need?

Complete the algorithm and replicate it on the graph we used as an example in the lecture. Do you get the same result? However, what the algorithm does now is simply print the nodes as you visit them. In order to calculate distances, you'll need to add a variable `dists` (a dictionary) which keeps track of distances to all nodes. Specifically, it should be initialized to zero for `start`; when reaching node `w`, it should be set higher than the distance of node `v`.

You can similarly construct the paths taken using a dictionary that stores the previous node for every node visited, ie, it stores `v` for every `w`.

2 Small worlds and Kevin Bacon

In this section, you'll use your algorithm to examine the small worlds phenomenon on a large movie dataset from the Internet Movie Database. The idea is, in brief, that everyone in the world is connected by a surprisingly small number of connections in a social graph. You'll test this idea by finding the distances between Hollywood actors. Connections between actors are simply movies they acted in together. For example, the prolific actor Kevin Bacon is said to be very well connected by this measure.

The file `moviesg.txt` contains a small sample of movie data; a larger sample is provided in `movies.txt`.[2] The function `readMovieData` is provided for your convenience to read this data into

[2] Available freely from IMDB, cleaned data courtesy of Robert Sedgewick, Princeton University.

a graph. Make sure to set the working directory correctly in Spyder before reading the data: This can be done by changing the directory using the buttons on the top right corner of the program.

Use your BFS algorithm to find the shortest distances and paths from Kevin Bacon to actors like Nicole Kidman and Marlon Brando. What is the largest distance from Kevin Bacon in the graph? What is the distribution of distances? Notice that the algorithm will output double distances with these data, as it will count the distance from an actor to a movie as one edge and from the movie to another actor as another one.

3 Challenge problem: depth-first search

Another useful and widely applied graph-search algorithm is depth-first search (DFS). The idea, like that of BFS, is simple. From a starting node, we now explore aggressively in one direction until we cannot do it anymore. Then we trace back our steps until we can explore something new, and repeat. This is essentially the approach of exploring a maze — a famous early instance being the story of Theseus and the Minotaur.

Your task is to implement this algorithm in Python, as per the recipe below. The algorithm itself is very similar to BFS. The main difference is that in BFS we were using a queue: a first-in first-out data structure. This allowed us to explore the graph in layers. With DFS, we want to explore the graph depths aggressively, so instead we need a first-in last-out data structure. The data structure suitable for this is called a *stack*, but it can be implemented conveniently in Python simply using a list, and appending and popping in the end of the list as appropriate.

Algorithm (DFS):

- Pick a starting node s and initialize set of explored nodes as empty but for s
- Initialize an empty list L of reachable nodes, append s to list
- While L is not empty:
  - Remove last item of L as current node
  - If current node not explored:
    - Add current node to explored
    - Append current node's children to end of list

Notice how, comparing the algorithms, you can mostly just copy your BFS implementation and change it to use a list and removing nodes from its end instead of using a queue and removing them from the beginning.

Use the algorithm to search through the example from the class. How different is the DFS approach from BFS in this case?

We used BFS to find shortest paths in a graph — is DFS as good for this purpose? Search online for comparisons on what kinds of problems you would rather solve with either approach.