Programming fundamentals. Graphs pt. 2

Pepe García

2020-04-20

Programming fundamentals. Graphs pt. 2

https://slides.com/pepegar/graphs-2/live

Plan for today

- Graph traversals
- Path finding
- NetworkX library

Graph traversals

A graph traversal is the process of exploring a graph.

Graph traversals

There are two main ways of doing Graph traversals:

- Breadth First Search (**BFS**)
- Depth First search (**DFS**)

DFS

Depth-first search is a technique for traversing graphs in which we will go through a branch of it until we find its end before going to the next branch.

BFS

On the other hand, **Breadth-first search** will visit all the neighbors of a given node before moving to the next level of the graph.

This is the traversal we will focus on today

Understanding BFS

The technique we will use in order to traverse the queue can be described as follows:

1. Start from the given start node.
2. queue neighbors of start.
3. while the queue is not empty, keep exploring neighbors, in order.

our queue will be FIFO

Understanding BFS

```
graph = {
    1: [2,3,4],
    2: [5, 6],
    3: [],
    4: [7, 8],
    5: [],
    6: [],
    7: [],
    8: []
}
```

Let's see in the whiteboard how we would traverse this graph using BFS

Implementing BFS

```python
def bfs(graph, start):
    queue = [start]
    visited = []
    while queue:
        current = queue.pop(0)
        for neighbor in graph[current]:
            if neighbor not in visited:
                queue.append(neighbor)
        visited.append(current)
    return visited
```

Break!

Path finding

Path finding is one of the most recurrent problems in graphs.

We will solve this problem by doing some small modifications to the **bfs** implementation.

Path finding

Create a new function **find_all_paths(graph, start)** that uses **BFS** and returns a dictionary with all the paths to nodes connected to **start**.

You can start by copying the code in **bfs** and modifying it.

Path finding

```python
def find_all_paths(graph, start):
    queue = [start]
    paths = {start: [start]}

    while queue:
        current = queue.pop(0)
        for neighbor in graph[current]:
            if neighbor not in paths:
                paths[neighbor] = paths[current] + [neighbor]
                queue.append(neighbor)
    return paths
```

Path finding

Now that we have the paths to all nodes from our start node to all connected nodes, we can easily find the one we're looking for

```python
def find_path(graph, start, end):
    paths = find_all_paths(graph, start)

    if end in graph:
        return graph[end]
    else:
        return None
```

NetworkX library

The NetworkX library is a library for dealing with graphs. Its **very** powerful, and we can use it for most graph related tasks.

It is already included in Anaconda, so we don't need to download it again.

NetworkX. DiGraphs

The convention is to import the library under the **nx** name.

In this example we are creating a directed graph with three nodes and two edges.

```
import networkx as nx

G = nx.DiGraph()

G.add_node(1)
G.add_node(2)
G.add_node(3)

G.add_edge(1, 2)
G.add_edge(2, 3)
```

NetworkX. DiGraphs

We can also the **edges** and **nodes** method to get the relevant parts of the graph respectively.

```
# returns the edges
G.edges



# returns the nodes
G.nodes
```

NetworkX. DiGraphs

```python
import networkx as nx

# when called without params, will return
# shortest paths from all nodes to all nodes
nx.shortest_path(G)

# this will return all the shortest paths
# starting at node 4
nx.shortest_path(G, 4)

# this will return the shortest path from 1
# to 4
nx.shortest_path(G, 1, 4)
```

Exercises

- Create a function named **six_or_less** that returns whether two nodes in the graph are at distance 6 or less.
- Create a function **degrees** that receives a graph and returns a dictionary with the degree of each node.
- Investigate the NetworkX library. Use it to **draw the graph** we have been working on in class.