BFS (BREADTH FIRST SEARCH) GRAPH TRAVERSAL (PYTHON RECIPE) BY MOJAVE KID

ACTIVESTATE CODE (HTTP://CODE.ACTIVESTATE.COM/RECIPES/576675/)

Guido illustrated the DFS recursive traversal of a graph (http://www.python.org/doc/essays/graphs.html) However if the graph is too big, recursion error occurs.

Here im pitching in my recipe for an iterative BFS traversal.

Im using Guido's graph representation using dictionary.

Python, 77 lines

```python
# a sample graph
graph = {'A': ['B', 'C','E'],
             'B': ['A','C', 'D'],
             'C': ['D'],
             'D': ['C'],
             'E': ['F','D'],
             'F': ['C']}

class MyQUEUE: # just an implementation of a queue

        def __init__(self):
                self.holder = []

        def enqueue(self,val):
                self.holder.append(val)

        def dequeue(self):
                val = None
                try:
                        val = self.holder[0]
                        if len(self.holder) == 1:
                                self.holder = []
                        else:
                                self.holder = self.holder[1:]
                except:
                        pass

                return val
```

```
        def IsEmpty(self):
                result = False
                if len(self.holder) == 0:
                        result = True
                return result


path_queue = MyQUEUE() # now we make a queue


def BFS(graph,start,end,q):

        temp_path = [start]

        q.enqueue(temp_path)

        while q.IsEmpty() == False:
                tmp_path = q.dequeue()
                last_node = tmp_path[len(tmp_path)-1]
                print tmp_path
                if last_node == end:
                        print "VALID_PATH : ",tmp_path
                for link_node in graph[last_node]:
                        if link_node not in tmp_path:
                                new_path = []
                                new_path = tmp_path + [link_node]
                                q.enqueue(new_path)

BFS(graph,"A","D",path_queue)
```

```
-------------results------------------
['A']
['A', 'B']
['A', 'C']
['A', 'E']
['A', 'B', 'C']
['A', 'B', 'D']
VALID_PATH :  ['A', 'B', 'D']
['A', 'C', 'D']
VALID_PATH :  ['A', 'C', 'D']
['A', 'E', 'F']
['A', 'E', 'D']
VALID_PATH :  ['A', 'E', 'D']
['A', 'B', 'C', 'D']
VALID_PATH :  ['A', 'B', 'C', 'D']
['A', 'E', 'F', 'C']
['A', 'E', 'F', 'C', 'D']
VALID_PATH :  ['A', 'E', 'F', 'C', 'D']
```

Tags: bfs, breath

6 COMMENTS

Matteo Dell'Amico 9 years ago

I think you should use collections.deque rather than writing your own queue class.

Plus, a search algorithm should not visit nodes more than once.

I think the Pythonic way of implementing visits should be a generator. To give accessing methods enough information to do useful things, every time I visit a node I return its parent. Here is my own implementation of BFS and DFS, with a sample implementation of the shortest_path function.

For a complete Python graph library, I advise you to check NetworkX: http://networkx.lanl.gov/ .

```
from collections import deque

def bfs(g, start):
    queue, enqueued = deque([(None, start)]), set([start])
    while queue:
        parent, n = queue.popleft()
        yield parent, n
        new = set(g[n]) - enqueued
        enqueued |= new
        queue.extend([(n, child) for child in new])

def dfs(g, start):
    stack, enqueued = [(None, start)], set([start])
    while stack:
        parent, n = stack.pop()
        yield parent, n
        new = set(g[n]) - enqueued
        enqueued |= new
        stack.extend([(n, child) for child in new])

def shortest_path(g, start, end):
    parents = {}
    for parent, child in bfs(g, start):
        parents[child] = parent
        if child == end:
            revpath = [end]
            while True:
                parent = parents[child]
                revpath.append(parent)
                if parent == start:
                    break
                child = parent
            return list(reversed(revpath))
    return None # or raise appropriate exception

if __name__ == '__main__':
    # a sample graph
    graph = {'A': ['B', 'C','E'],
```

```
            'B': ['A','C', 'D'],
            'C': ['D'],
            'D': ['C'],
            'E': ['F', 'D'],
            'F': ['C']}

    print(shortest_path(graph, 'A', 'D'))
```


Matteo Dell'Amico 9 years ago

More compact implementation of the shortest_path function:

```
def shortest_path(g, start, end):
    paths = {None: []}
    for parent, child in bfs(g, start):
        paths[child] = paths[parent] + [child]
        if child == end:
            return paths[child]
    return None # or raise appropriate exception
```


Agnius Vasiliauskas 8 years, 11 months ago

To "Matteo Dell'Amico":

"Plus, a search algorithm should not visit nodes more than once"

You are wrong,- algorithm should not visit nodes more than once in one PATH. So it is allowable to visit node several times in different A-D routes. So mojave kid implementation of BFS is correct.

"More compact implementation of the shortest_path function"

I think this is redundant information for breadth first search algorithm, because it strongly depends on goal - what you want to find out from search. If you want to find just shortest route from A to D,- than OK, your suggestions is good. But what if I want to find ALL routes from A to D ? In that case your suggestion is meaningless and mojave kid implementation - is good for such problem. So I suggest, lets leave concrete implementation of BFS algorithm for the script user...

Matteo Dell'Amico 8 years, 11 months ago

Agnius: http://mathworld.wolfram.com/Breadth-FirstTraversal.html (and all other references I can find) explain clearly that a BFS shouldn't visit nodes more than once. Of course, the shortest_path function does what its name says: it finds the shortest path. Generating all paths is very different, and can easily be

computationally prohibitive.

Eknath 7 years ago

So has this (all shortest paths between 2 nodes) been included in the networkx's functions?

btw, thank you for this implementation. Works just fine.

Eknath 7 years ago

This doesn't work for a MultiGraph() consisting of 2 **different** edges between 2 adjacent nodes. What needs to be done then?