Notes - Algorithms & Data Structures - Graphs - Intro - Part 2

A brief intro to *graphs* for data structure and algorithm usage.

Contents

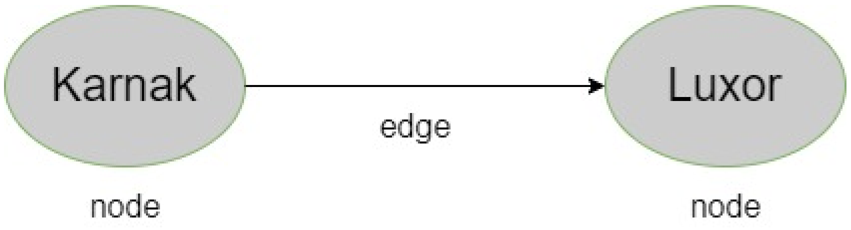
Intro

A graph data structure may be defined in computer science as a way to model a given set of connections.

We may commonly use a *graph* to model patterns and connections for a given problem. For example, such connections may infer relationships within data.

A graph includes *nodes* and *edges*, which help us define such connections.

For example, we have two nodes with a single edge



Each node may be connected to many other nodes in the graph, commonly referenced as *neighbour* nodes.

Implement algorithm

So, we can start to implement the algorithm by considering an outline of the underlying structure.

We might consider such an implementation relative to the earlier example for family members who have visited sites in Egypt.

For example,

- we can keep an initial *queue* for the names of family members to check
- we then *pop* a name off the queue
- we check this name to see if they have visited a defined location in Egypt, perhaps **Giza**
    - if *yes*, then the search is, of course, finished
    - if *no*, then we add all of the neighbours for this current family member to the queue
- then, we *loop* and repeat the search
- if the queue is empty, then no family member has visited Giza, for example.

The initial graph may be defined as follows, using the patterns of usage we've already seen for Python

```
# define graph for family members
graph = {}
graph["me"] = ["emma", "daisy", "yvaine"]
graph["daisy"] = ["rose", "violet"]
graph["emma"] = ["violet"]
graph["yvaine"] = ["tristram", "cat"]
graph["rose"] = []
graph["violet"] = []
graph["tristram"] = []
graph["cat"] = []
```

Current output is as follows for the graph nodes

```
{'me': ['emma', 'daisy', 'yvaine'], 'daisy': ['rose', 'violet'], 'emma':
['violet'], 'yvaine': ['tristram', 'cat'], 'rose': [], 'violet': [], 'tristram':
[], 'cat': []}
```

We may then begin by defining the queue, using the double-ended queue function (deque) in Python

```
# imports
from collections import deque

# define and create new queue
name_queue = deque();
```

Then, we may add all of the neighbours for the node `me` to the queue we'd like to query and search.

So, initially we begin by adding the immediate neighbour nodes to the queue, including `emma`, `daisy` and `yvaine`.

The current queue output is as follows,

```
deque(['emma', 'daisy', 'yvaine'])
```

We now have our initial queue to query for family members. So, we may now continue to check and populate the queue using the graph.

For example,

```
name_queue += graph["me"]
```

Obviously, we wouldn't want to do this manually so we may add a `while` loop to check the queue.

We'll use this loop as follows,

- while the queue is not empty
  - get the first name in the queue
  - check if the name has visited Giza
    - if yes, then return as we've found a family member who travels to Egypt
    - otherwise, add their neighbour nodes to the queue to broaden search
- exit loop if no matched name found...

So, we'll now modify and update our graph to include details for places visited

```
# define graph for family members
graph = {}
graph["me"] = [["emma", "cairo"], "daisy", "yvaine"]
graph["daisy"] = ["rose", "violet"]
graph["emma"] = [["violet", "giza"]]
graph["yvaine"] = ["tristram", "cat"]
graph["rose"] = []
graph["violet"] = []
graph["tristram"] = []
graph["cat"] = []
```

There are many different options for storing such data, but this will work fine for querying the graph of family members.

The update now includes two family members who have visited locations in Egypt, Cairo and Giza.

So, the `while loop` may be defined as follows

```
# query the queue while not empty
while name_queue:
    # get first name from queue
    name = name_queue.popleft()
    # check if the current node has visited giza
    if visited_giza(name):
        # print family member who has visited Giza...
        print(name)
        return True
    else:
        # check if name is array or not...
        if (isinstance(name, list)):
            # add just the name to queue...
            name_queue += graph[name[0]]
        else:
            # they haven't visited giza - add neighbour nodes...
            name_queue += graph[name]
print("no family member has visited Giza...")
# no family member has visited giza
return False
```

In this example, we can see that there is a check for a visit to Giza, the function `visited_giza(name)` and then a simple check of the `name` variable to ensure we pass the required string for the name in the graph.

We may add the function `visited_giza()` as follows,

```
# check name in graph
def visited_giza(name):
    if "giza" in name:
        # return just the name from the array...
        return name[0]
```

breadth-first search

So, we can see how breadth-first search may be implemented for our inital graph of family members.

We check the passed `name` variable with the function `visited_giza()`. If it contains the required string `giza`, we can return the name of this family member and exit the graph query. We've found our family mamber.

However, if we don't find the required family member we continue the `while` loop. However, we need to ensure we can grab the name from an inner list.

For example, if we check `emma`, a neighbour node to the node `me`, it is a `list`, an array, with a location of `cairo`. However, as this doesn't match the required location of `giza`, we simply grab the name itself, `emma`, and repeat the loop to continue checking the updated queue.

We can now update our app as follows,

```
def search(name):
    # define and create new queue
    name_queue = deque();
    # add all neighbours of 'me' to queue
    name_queue += graph[name]

    # query the queue while not empty
    while name_queue:
        # get first name from queue
        name = name_queue.popleft()
        # check if the current node has visited giza
        if visited_giza(name):
            # print family member who has visited Giza...
            print(name)
            return True
        else:
            # check if name is array or not...
            if (isinstance(name, list)):
                # add just the name to queue...
                name_queue += graph[name[0]]
            else:
                # they haven't visited giza - add neighbour nodes...
                name_queue += graph[name]
    print("no family member has visited Giza...")
    # no family member has visited giza
    return False
```

In effect, this algorithm will continue until either of the following conditions is met

- a family member is found who has visited Giza
- the name queue is empty, which means there is no family members left to check...

duplication of queries

However, we still have an issue with the current algorithm for this search.

If we check the initial graph, we can see that the family member `violet` is a neighbour of both `daisy` and `emma`. This means that `violet` will currently be added to the queue twice.

Of course, this means we are currently also searching this node twice as well. However, we only need to search each node once, regardless of defined neighbour.

One way to deal with this issue is to identify a node as *searched*, which will stop duplication of effort in the algorithm.

So, we may add a list of nodes already checked during the search.

updated breadth-first search

If we tried to search the following updated graph

```
# define graph for family members
graph = {}
graph["me"] = [["emma", "cairo"], "daisy", "yvaine"]
graph["daisy"] = ["rose", "violet"]
graph["emma"] = ["violet"]
graph["yvaine"] = [["tristram", "giza"], "cat"]
graph["rose"] = []
graph["violet"] = []
graph["tristram"] = []
graph["cat"] = []
```

we would need to keep a check of names already searched to avoid unnecessary duplication.

So, we may now update the code for *breadth-first* search as follows,

```
def search(name):
    # define and create new queue
    name_queue = deque();
    # add all neighbours of 'me' to queue
    name_queue += graph[name]
    # keep track of names already searched...
    names_searched = []
    # query the queue while not empty
    while name_queue:
        # get first name from queue
        name = name_queue.popleft()
        # check if name already searched...if not, then search
        if not name in names_searched:
            # check if the current node has visited giza
            if visited_giza(name):
                # print family member who has visited Giza...
                print(name[0] + " has visited " + name[1])
                return True
            else:
                # check if name is array or not...
                if (isinstance(name, list)):
                    # add just the name to queue...
                    name_queue += graph[name[0]]
                    # add name to already searched
                    names_searched.append(name[0])
                else:
                    # they haven't visited giza - add neighbour nodes...
                    name_queue += graph[name]
                    # add name to already searched
                    names_searched.append(name)
    print("no family member has visited Giza...")
    # no family member has visited giza
    return False
```

If we then search using `me` as the initial name, we can see how the `names_searched` array keeps a check of the family member names already searched in the graph.

```
[]
['emma']
['emma', 'daisy']
['emma', 'daisy', 'yvaine']
['emma', 'daisy', 'yvaine', 'violet']
['emma', 'daisy', 'yvaine', 'violet', 'rose']
tristram has visited giza
```

and we end with the found node for `tristram` in the graph.

performance and time

With the above example, as we search the graph for a family member, we may potentially need to follow each edge.

So, we may initially define Big O with a running time of at least `O(number of edges)`.

As we've just seen, we're also maintaing a queue of each name we need to search.

As expected, adding a single name to the queue takes *constant* time, `O(1)`. If we perform this task for each name in the graph, then we end up with a potential time of `O(number of names)`.

So, breadth-first search will take

```
O(number of names + number of edges)
```

which may be written for graphs as follows,

```
O(V+E)
```

where `V` is for the number of vertices, and `E` the number of edges. Basically, the nodes and edges of the graph.