Comp 363 - Design and Analysis of Computer Algorithms

Spring Semester 2020 - Week 12 - part 2

Dr Nick Hayward

graphs - implement algorithm - part 1

- start to implement algorithm by considering outline of underlying structure
- consider an implementation relative to earlier example
  - *i.e. family members who have visited sites in Egypt...*
- e.g.
  - *keep an initial queue for names of family members to check*
  - *then pop a name off queue*
  - *check this name to see if they have visited a defined location in Egypt, e.g. **Giza***
    - if *yes* - search is finished
    - if *no* - add all neighbours for this current family member to queue
  - *then, loop and repeat search*
  - *if queue is empty*
    - no family member has visited Giza...

Algorithms and Data Structures

graphs - implement algorithm - part 2

- initial graph may be defined as follows
  - *using 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 graph nodes

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

Algorithms and Data Structures

graphs - implement algorithm - part 3

- then begin by defining queue
  - *using double-ended queue function (deque) in Python*

```
# imports
from collections import deque

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

- then add all neighbours for node `me`
  - *add to queue we'd like to query and search*

Algorithms and Data Structures

graphs - implement algorithm - part 4

- begin by adding immediate neighbour nodes to queue
  - *including* `emma`, `daisy` *and* `yvaine`
- current queue output is as follows

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

- initial queue to query for family members
- now continue to check and populate queue using graph
  - *e.g.*

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

graphs - implement algorithm - part 5

- do **not** want to do this manually
  - *add a `while` loop to check queue*
- use this loop as follows,
  - *while the queue is not empty*
    - get first name in queue
    - check if name has visited Giza
      - if yes - return, a family member found who travels to Egypt
      - otherwise - add their neighbour nodes to queue to broaden search
  - *exit loop if no matched name found...*

Algorithms and Data Structures

graphs - implement algorithm - part 6

- modify and update 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"] = []
```

- many different options for storing such data
  - *this example works fine for querying graph of family members...*
- update now includes two family members who have visited locations in Egypt
  - *e.g. Cairo and Giza*

Algorithms and Data Structures

graphs - implement algorithm - part 7

- `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
  - *a check for a visit to Giza - function `visited_giza(name)`*
  - *then simple check of `name` variable*
  - *ensure we pass required string for name in graph*

Algorithms and Data Structures

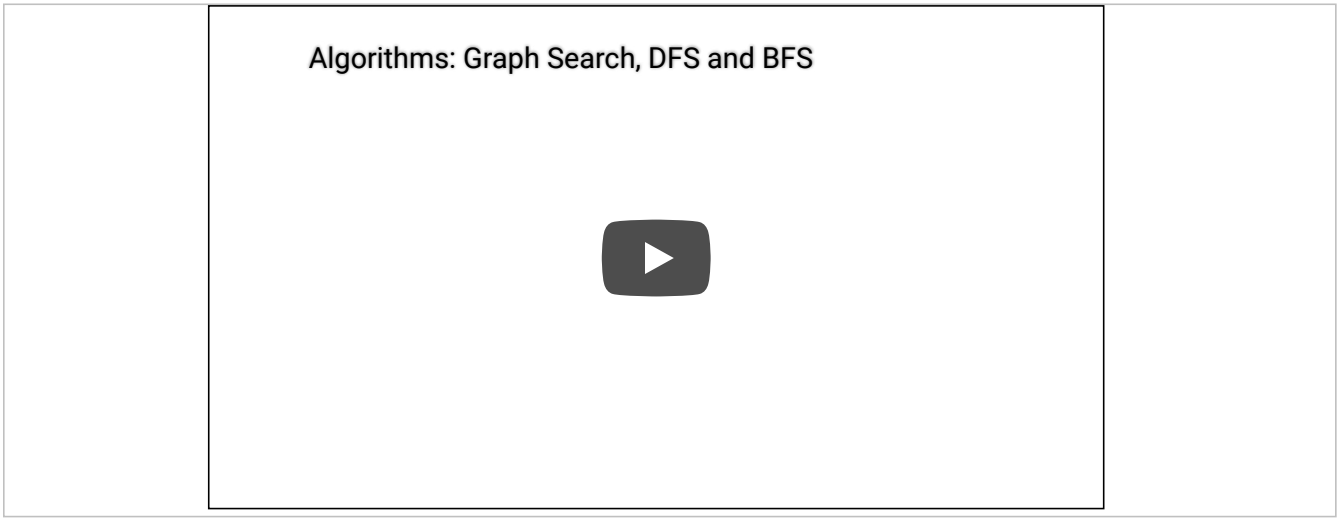
graphs - implement algorithm - part 8

- add 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]
```

Video - Algorithms and Data Structures

graphs - Java - part 5



Graphs - Java - Breadth-first Search - UP TO 9:04

Source - Graphs - Java - YouTube

Algorithms and Data Structures

graphs - implement algorithm - breadth-first search - part 1

- clearly see how breadth-first search may be implemented for graph of family members
- check passed `name` variable with function `visited_giza()`
  - *if it contains required string `giza`*
  - *return name of this family member*
  - *exit graph query - family mamber found...*
- if we don't find required family member
  - *continue `while` loop*
  - *need to ensure we can grab name from an inner list*
- e.g. if we check `emma` - a neighbour node to node `me`
  - *it is a `list` - an array*
  - *with a location of `cairo`*
- `cairo` doesn't match required location of `giza`
  - *grab the name, `emma`*
- repeat loop to continue checking updated queue

Algorithms and Data Structures

graphs - implement algorithm - breadth-first search - part 2

- 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
```

- i.e. algorithm will continue until either condition is met
  - *a family member is found who has visited Giza*
  - *name queue is empty - no family members left to check...*

graphs - implement algorithm - duplication of queries

- still have an issue with current algorithm for this search
- if we check initial graph
  - *family member* `violet` *is a neighbour of both* `daisy` *and* `emma`
  - *i.e.* `violet` *will currently be added to the queue twice*
- currently also searching this node twice as well
- only need to search each node once
  - *regardless of defined neighbour*
- resolve issue by identifying node as *searched*
  - *stops duplication of effort in algorithm*
- may add a list of nodes already checked during search

Algorithms and Data Structures

graphs - implement algorithm - updated breadth-first search - part 1

- 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"] = []
```

- need to keep a check of names already searched
  - *check to avoid unnecessary duplication*

Algorithms and Data Structures

graphs - implement algorithm - updated breadth-first search - part 2

- update 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
```

graphs - implement algorithm - updated breadth-first search - part 3

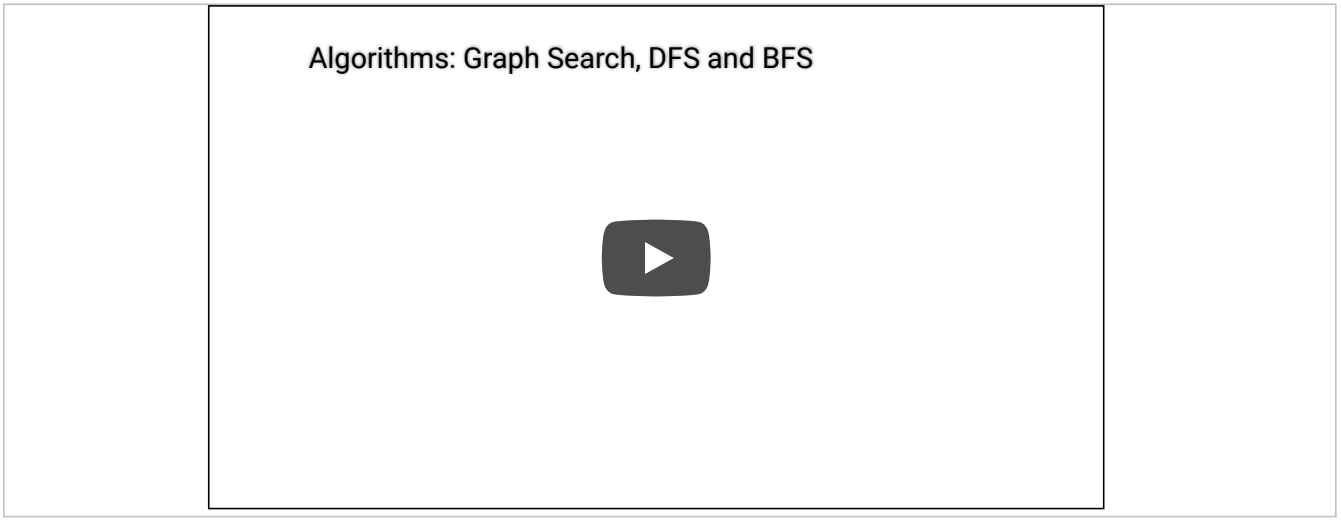
- if we then search using `me` as initial name
  - *`names_searched` array keeps check of family member names already searched in graph*

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

- end with found node for `tristram` in graph...

Video - Algorithms and Data Structures

graphs - Java - part 6



Graphs - Java - Breadth-first Search - UP TO 11:42

Source - Graphs - Java - YouTube

Algorithms and Data Structures

graphs - implement algorithm - performance and time

- with above example
  - *as w search graph for a family member, may potentially need to follow each edge*
- initially define Big O with running time of at least `O(number of edges)`
- also maintaing a queue of each name we need to search
- adding a single name to queue takes *constant* time, `O(1)`
- if we perform this task for each name in graph
  - *we end up with a potential time of `O(number of names)`*
- breadth-first search will take

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

- may be written for graphs as follows

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

- `V` = number of vertices
- `E` = number of edges
- i.e. the nodes and edges of the graph...

Resources

various

- Python patterns - implementing graphs

videos

- Graphs - Java - YouTube
- Joy of Data - YouTube