

# Depth-First Search and Breadth-First Search in Python

05 Mar 2014

Graph theory and in particular the graph ADT (abstract data-type) is widely explored and implemented in the field of Computer Science and Mathematics. Consisting of vertices (nodes) and the edges (optionally directed/weighted) that connect them, the data-structure is effectively able to represent and solve many problem domains. One of the most popular areas of algorithm design within this space is the problem of checking for the existence or (shortest) path between two or more vertices in the graph. Properties such as edge weighting and direction are two such factors that the algorithm designer can take into consideration. In this post I will be exploring two of the simpler available algorithms, Depth-First and Breadth-First search to achieve the goals highlighted below:

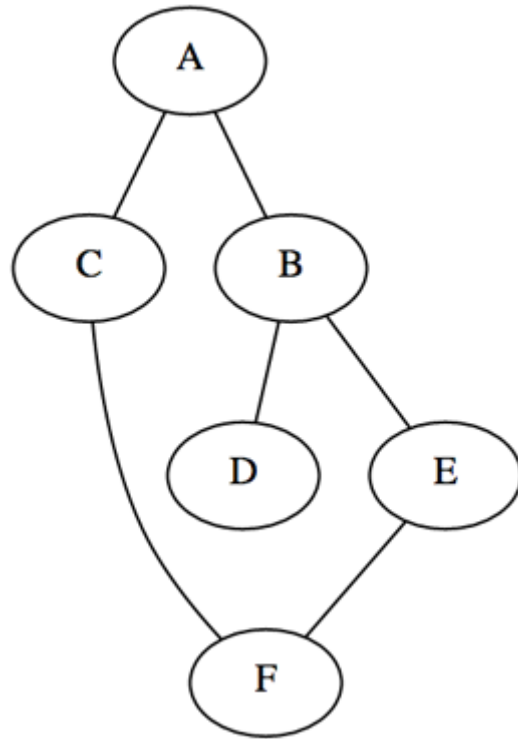
- Find all vertices in a subject vertices [connected component](#).
- Return all available paths between two vertices.
- And in the case of BFS, return the shortest path (length measured by number of path edges).

## The Graph

So as to clearly discuss each algorithm I have crafted a [connected graph](#) with six vertices and six incident edges. The resulting graph is undirected with no assigned edge weightings, as length will be evaluated based on the number of path edges traversed. There are two popular options for representing a graph, the first being an [adjacency matrix](#) (effective with dense graphs) and second an [adjacency list](#) (effective with sparse graphs). I have opted to implement an adjacency list which stores each node in a dictionary along with a set containing their adjacent nodes. As the graph is undirected each edge is stored in both incident nodes adjacent sets.

```
graph = {'A': set(['B', 'C']),  
         'B': set(['A', 'D', 'E']),  
         'C': set(['A', 'F']),  
         'D': set(['B']),  
         'E': set(['B', 'F']),  
         'F': set(['C', 'E'])}
```

Looking at the graph depiction below you will also notice the inclusion of a cycle, by the adjacent connections between 'F' and 'C/E'. This has been purposely included to provide the algorithms with the option to return multiple paths between two desired nodes.



## Depth-First Search

The first algorithm I will be discussing is Depth-First search which as the name hints at, explores possible vertices (from a supplied root) down each branch before backtracking. This property allows the algorithm to be implemented succinctly in both iterative and recursive forms. Below is a listing of the actions performed upon each visit to a node.

- Mark the current vertex as being visited.
- Explore each adjacent vertex that is not included in the visited set.

## Connected Component

The implementation below uses the stack data-structure to build-up and return a set of vertices that are accessible within the subjects connected component. Using Python's overloading of the subtraction operator to remove items from a set, we are able to add only the unvisited adjacent vertices.

```
def dfs(graph, start):
    visited, stack = set(), [start]
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            stack.extend(graph[vertex] - visited)
    return visited

dfs(graph, 'A') # {'E', 'D', 'F', 'A', 'C', 'B'}
```

The second implementation provides the same functionality as the first, however, this time we are using the more succinct recursive form. Due to a common Python gotcha with [default parameter values](#) being created only once, we are required to create a new visited set on each user invocation. Another Python language detail is that function variables are passed by reference, resulting in the visited mutable set not having to be reassigned upon each recursive call.

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    for next in graph[start] - visited:
        dfs(graph, next, visited)
```

```
return visited
```

```
dfs(graph, 'C') # {'E', 'D', 'F', 'A', 'C', 'B'}
```

## Paths

We are able to tweak both of the previous implementations to return all possible paths between a start and goal vertex. The implementation below uses the stack data-structure again to iteratively solve the problem, yielding each possible path when we locate the goal. Using a [generator](#) allows the user to only compute the desired amount of alternative paths.

```
def dfs_paths(graph, start, goal):
    stack = [(start, [start])]
    while stack:
        (vertex, path) = stack.pop()
        for next in graph[vertex] - set(path):
            if next == goal:
                yield path + [next]
            else:
                stack.append((next, path + [next]))

list(dfs_paths(graph, 'A', 'F')) # [['A', 'C', 'F'], ['A', 'B', 'E', 'F']]
```

The implementation below uses the recursive approach calling the ‘yield from’ [PEP380](#) addition to return the invoked located paths. Unfortunately the version of [Pygments](#) installed on the server at this time does not include the updated keyword combination.

```
def dfs_paths(graph, start, goal, path=None):
    if path is None:
        path = [start]
    if start == goal:
        yield path
    for next in graph[start] - set(path):
        yield from dfs_paths(graph, next, goal, path + [next])
```

## Breath-First Search

An alternative algorithm called Breath-First search provides us with the ability to return the same results as DFS but with the added guarantee to return the shortest-path first. This algorithm is a little more tricky to implement in a recursive manner instead using the queue data-structure, as such I will only be documenting the iterative approach. The actions performed per each explored vertex are the same as the depth-first implementation, however, replacing the stack with a queue will instead explore the breadth of a vertex depth before moving on. This behavior guarantees that the first path located is one of the shortest-paths present, based on number of edges being the cost factor.

## Connected Component

Similar to the iterative DFS implementation the only alteration required is to remove the next item from the beginning of the list structure instead of the stacks last.

```
def bfs(graph, start):
    visited, queue = set(), [start]
    while queue:
        vertex = queue.pop(0)
        if vertex not in visited:
            visited.add(vertex)
            queue.extend(graph[vertex] - visited)
    return visited
```

```
bfs(graph, 'A') # {'B', 'C', 'A', 'F', 'D', 'E'}
```

## Paths

This implementation can again be altered slightly to instead return all possible paths between two vertices, the first of which being one of the shortest such path.

```
def bfs_paths(graph, start, goal):
    queue = [(start, [start])]
    while queue:
        (vertex, path) = queue.pop(0)
        for next in graph[vertex] - set(path):
            if next == goal:
                yield path + [next]
            else:
                queue.append((next, path + [next]))

list(bfs_paths(graph, 'A', 'F')) # [['A', 'C', 'F'], ['A', 'B', 'E', 'F']]
```

Knowing that the shortest path will be returned first from the BFS path generator method we can create a useful method which simply returns the shortest path

found or 'None' if no path exists. As we are using a generator this in theory should provide similar performance results as just breaking out and returning the first matching path in the BFS implementation.

```
def shortest_path(graph, start, goal):  
    try:  
        return next(bfs_paths(graph, start, goal))  
    except StopIteration:  
        return None  
  
shortest_path(graph, 'A', 'F') # ['A', 'C', 'F']
```

## Resources


- [Depth-and Breadth-First Search](#)
- [Connected component](#)
- [Adjacency matrix](#)
- [Adjacency list](#)
- [Python Gotcha: Default arguments and mutable data structures](#)
- [PEP 380](#)
- [Generators](#)

25 Comments

Edd Mann

 Login ▾

 Recommend 9

 Share

Sort by Best ▾



Join the discussion...



**Gabe** • 2 years ago

The recursive DFS code is incorrect for Python. It will subtract the visited set according to how visited was set when the call was first made. So when the recursive calls are made and change visited, the original call ignore those changes and visits nodes that were already visited. The correct code would be:

```
def BFS(graph, node, visited=None):
    if visited is None:
        visited = set()
    print 'Visiting', node
    visited.add(node)
    for next in graph[node]:
        if next not in visited:
            BFS(graph, next, visited)
    return visited
```

You can't tell that with your implementation because you are printing the set(), which will never have duplicate elements. But if you add that print statement above to yours you will see the repetition.

4 ^ | v • Reply • Share ›

**Lima** • a year ago

Beautiful code, sir. Thank you so much for this very good explanation. I've never used any generator before.

1 ^ | v • Reply • Share ›

**lee101** • 19 days ago

Its more efficient to use a deque for BFS and use popleft() rather than pop(0) which shuffles the array and takes  $O(\text{len}(\text{queue}))$  time

^ | v • Reply • Share ›

**Tushar Jain** • 3 months ago

I'm using the shortest path using BFS, I get this " RecursionError: maximum

recursion depth exceeded in comparison" very quickly, any suggestion on how to make shortest\_path with BFS recursive ?

^ | v • Reply • Share >

**Arvinth Sanu** • 10 months ago

```
def iterative_bfs(graph, start):  
    """iterative breadth first search from start"""  
    bfs_tree = {start: {"parents": [], "children": [], "level": 0}}  
    q = [start] while q: current = q.pop(0) for v in graph[current]:  
        if not v in bfs_tree:  
            bfs_tree[v]={"parents": [current], "children": [], "level": bfs_tree[current]["level"] + 1}  
            bfs_tree[current]["children"].append(v) q.append(v) else: if bfs_tree[v]["level"] >  
            bfs_tree[current]["level"]: bfs_tree[current]["children"].append(v) bfs_tree[v]  
            ["parents"].append(current)
```

this code works well for BFS.. Improve upon it. Also gives us the edges and parent child relation..you can add edges as well

^ | v • Reply • Share >

**Neil Prasad** • a year ago

Wow awesome post!

^ | v • Reply • Share >

**GL** • a year ago

I'm confused with the set() used in dfs/bfs function, what's the reason of choosing an unsorted container to keep all the visited nodes? thanks.

^ | v • Reply • Share >

**Rasmi Ranjan Nayak** ➔ GL • 9 months ago

To avoid duplicate

^ | v • Reply • Share >

**GL** ➔ Rasmi Ranjan Nayak • 9 months ago

Yeap, i totally agrees with this piont. My concern actually is: functions return this unsorted set as the result. As a client of this method, if I call `dfs(graph, start)`, I will get a set, same result for `dfs()` and `bfs()`. Understood these functions just for demo, but I would say these two functions dont have a clean name. There'll be no confusion for me if the name is `get_unique_nodes_with_dfs()`

^ | v • Reply • Share ›

**mathew** • a year ago

I have really complicated scenario to be solved in python 2.7. Can anyone help me? if you are up for some challenge please email me on `matrix_zs` at [hotmail.com](mailto:matrix_zs@hotmail.com)

^ | v • Reply • Share ›

**The Power of Love** • a year ago

I just wanted to thank you for this excellent resource. I'm trying to branch out (bahahaha pun intended) in python, and this helped a lot. Thank you, Edd.

^ | v • Reply • Share ›

**Pat** • a year ago

How do you change the sets to a list data structure... specifically the statement

```
queue.extend(graph[vertex] - visited)
```

which I don't understand, though I understand the principle of depth first and breadth first traversals. What would be the equivalent if you are working with a dictionary instead of a set? ie

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

Thanks for any help you can give...

^ | v • Reply • Share ›

**sumesh shivan** • 2 years ago

First of all I want say thanks for the post. But does the list subtraction works in

python? Im getting an error for this "queue.extend(graph[vertex] - visited)". It says  
TypeError: unsupported operand type(s) for -: 'list' and 'set'.

^ | v • Reply • Share ›

**sumesh shivan** ➔ sumesh shivan • 2 years ago

I got what I did wrong. I used list for graph values instead of set.

^ | v • Reply • Share ›

**Jason Chen** • 2 years ago

For somehow i tried the codes at my python interpreter and the visited lists it returns for DFS and BFS are not in correct order. I think if you use set(), the order could be screwed up when you print it out. I change all set() with list data structure and the DFS and BFS return visited list with correct order. Anyone tried and saw the same thing?

^ | v • Reply • Share ›

**Jeff LaCoursiere** ➔ Jason Chen • a year ago

Yup the examples don't match reality. In particular when I ran the BFS code above, I got:

```
>>> bfs(graph, 'A')
```

```
set(['A', 'C', 'B', 'E', 'D', 'F'])
```

Which actually makes a lot more sense than the output in the comments. So IMO the code is correct, but the shown output is wrong.

^ | v • Reply • Share ›

**Guest** • 2 years ago

Great post!

I implemented it to solve a maze. I can solve a small maze (~100 nodes), but when I have a big maze (~1000 nodes), the list(myGenerator) doesn't finish. Any idea why?

^ | v • Reply • Share ›

**Santa Jackson** → Guest • a year ago

Should probably use an informed search. Uniformed searches like depth first and breadth first are not the best way to go.

^ | v • Reply • Share ›



**Guest** • 3 years ago

Excellent post!

^ | v • Reply • Share ›

**Ngure Nyaga** • 3 years ago

Great stuff. I've had to deal with a graph at work that is almost identical to your example graph ( only about 300,000 times larger, with some paths being very, very long ). My first implementation was a recursive DFS that simply fell apart when presented with nodes near the root of the hierarchy ( maximum recursion limit exceeded ). I'll try out the iterative DFS and BFS and see what difference they make.

^ | v • Reply • Share ›

**Gabriel Altay** • 3 years ago

nice post. it's worth mentioning that there is an optimized queue object in the collections module called deque for which removing items from the beginning ( or popleft ) takes constant time as opposed to  $O(n)$  time for lists.

(<https://docs.python.org/2/library/collections.html#collections.deque>...

^ | v • Reply • Share ›

**Henry Chang** → Gabriel Altay • 3 years ago

How's the time complexity of "if something in deque"?

^ | v • Reply • Share ›

**Gabriel Altay** → Henry Chang • 3 years ago

the deque object is implemented as a doubly linked list so "if something in deque" is  $O(n)$ . this is the same time it would take for a list object which is implemented as a dynamic array. sets are basically hash tables and have the benefit of  $O(1)$  checking for membership (i.e. "if

something in set") but no concept of order.

^ | v • Reply • Share ›

**Usman Khan** • 3 years ago

helpfull

^ | v • Reply • Share ›

**Fabian Pedregosa** • 3 years ago

This is really useful and beautifully coded. Thanks!

^ | v • Reply • Share ›

ALSO ON EDD MANN

## AVL Trees in Clojure

1 comment • 6 months ago•

**Chiradip Mandal** — Fantastic enhancement over the BST

## Infix Calculator in Clojure

2 comments • a year ago•

**edd\_mann** — This looks awesome! :)

## Implementing Promise.all and Promise.race in JavaScript

1 comment • a year ago•


**Tomasz D** — Inspiring article, thanks! :)I was wondering - can yuo think of any real-life usecase for Promise.race? Or an ...

## Compiling PHP 5.5 with ZTS and pthreads Support

10 comments • 3 years ago•

**JRoze** — Use pecl install pthreads-2.0.10 if you get issues with PHP 7 requirements...

---

✉ Subscribe    Add Disqus to your site   Add Disqus Add    Privacy



Developer at [MyBuilder](#)

[Three Devs and a Maybe](#) podcast co-host

All ramblings can be found in the [Archive](#)