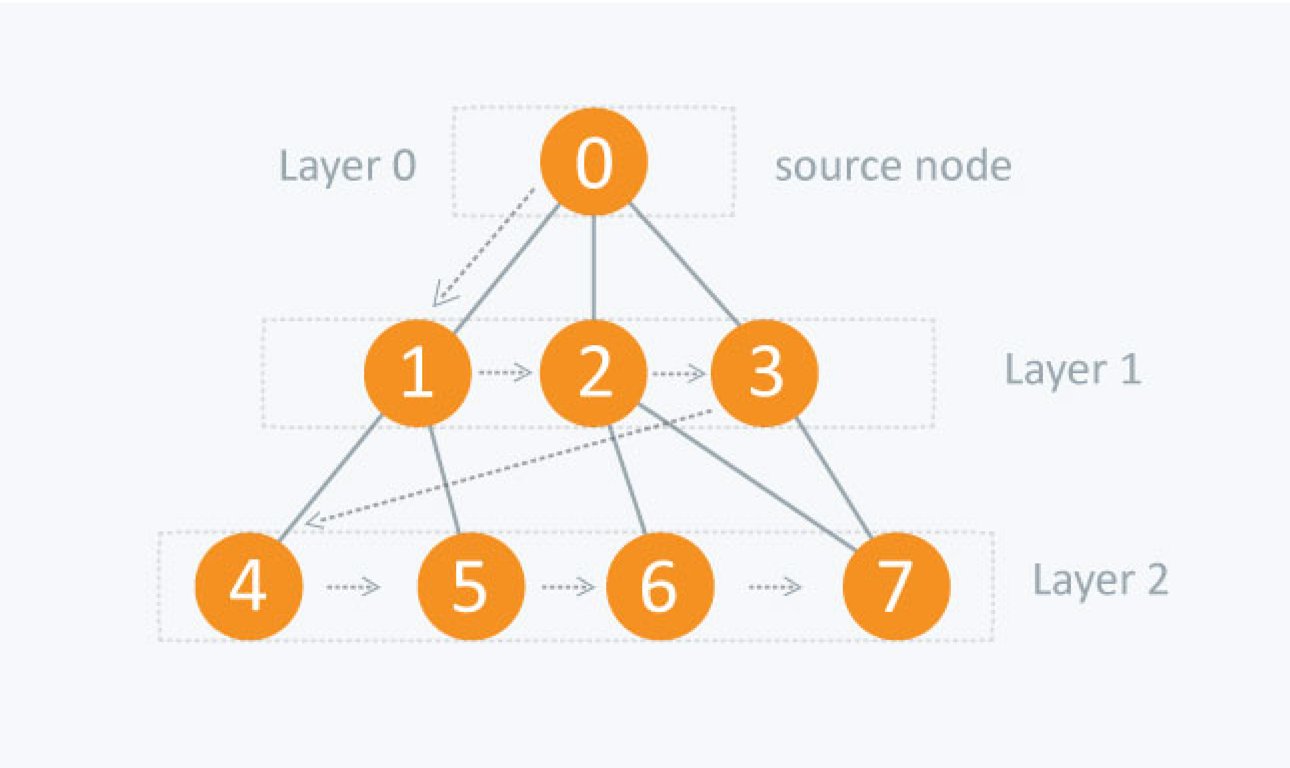
Graph Traversal

#Graphs #Algorithms

BFS

BFS or breadth first search is an algorithm to find the shortest path between points.
BFS traverses through the tree starting at an selected source node and moves entire levels at a time.



Note:
Unlike trees graphs can have cycles BFS implementations have an data structure of some kind to store if the vertex has been visited.

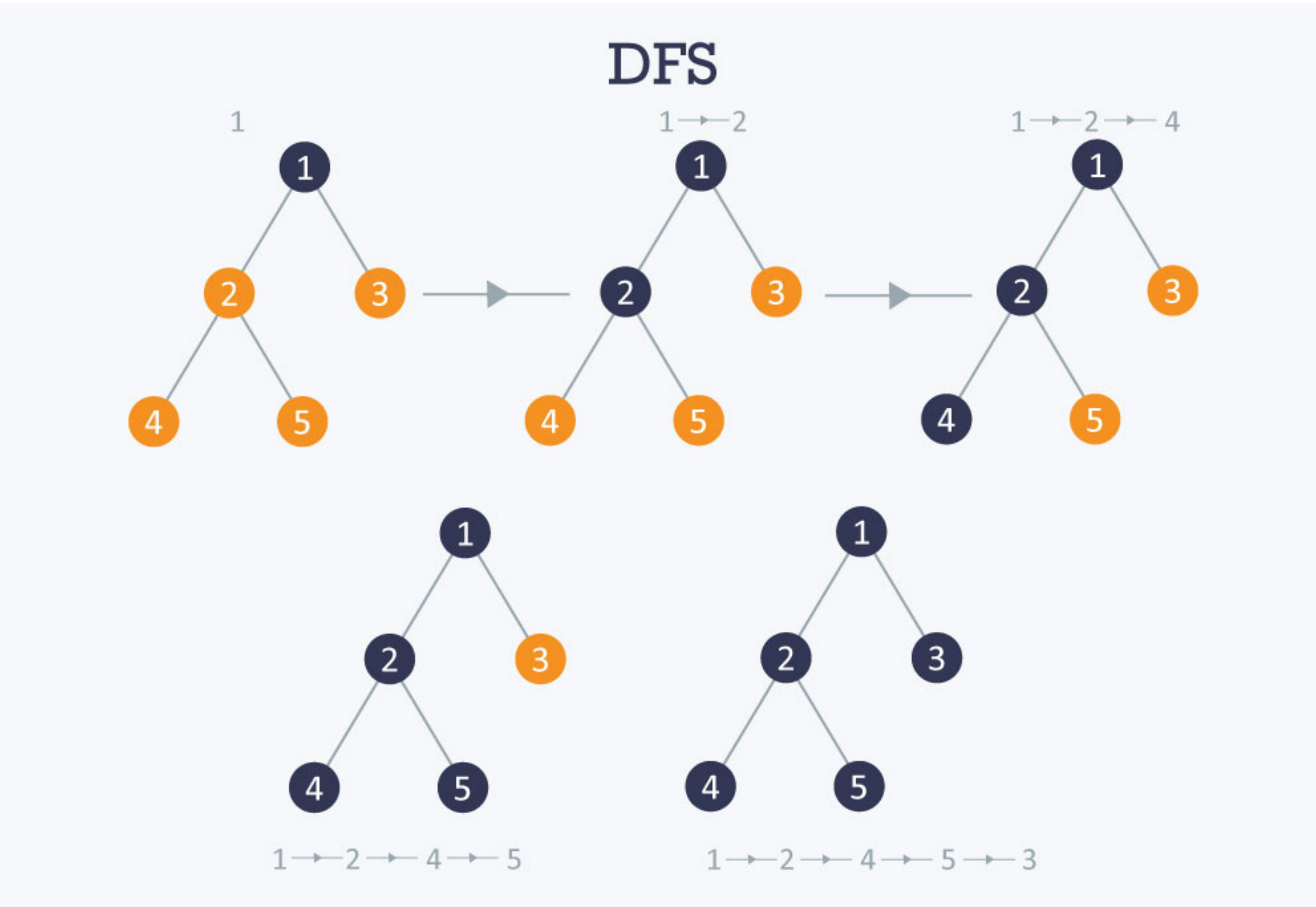
Python implementation:

```
def BFS(graph: List, s: int) -> none:
        q = []
        visited = set()

        # Start at the source
        visited.add(s)
        q.append(s)

        # While there is still nodes to visit
        while not q.empty():
                # We take out the top item
```

```python
        vertex = q.pop()

        # We navigate through it's neighbors
        for neighbors in graph[v]:
            # If they're not visited we add them to the list
            if neighbors not in visited:
                q.append(neighbors)
                visited.add(neighbors)
```

DFS

DFS or depth first search is an algorithm that uses backtracking. DFS travels the full depth of the tree/graph and then navigates backwards traveling the ones that haven't been navigated.



Python implementation:

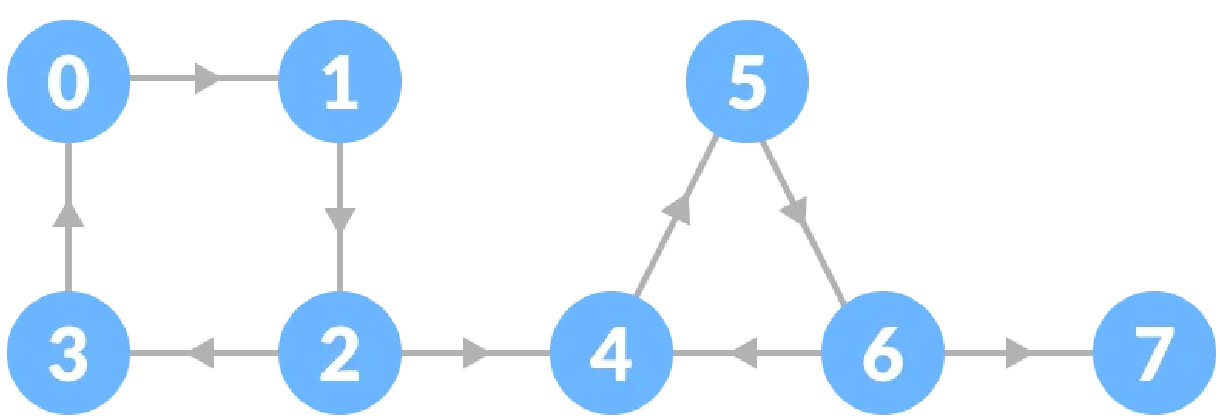
```python
visited = set()
def dfs-recursive(graph: List, node: int) -> none:
    # Visited current node
    visited.add(node)

    # Traverse neighbors
```

```
        for neighbor in graph[node]:
                # if it's not visited dfs more
                if neighbor not in visited:
                        dfs-recursive(graph, neighbor)


# Unsure
def dfs-iterative(graph: List, source: int) -> none:
        stack = []
        visited = set()

        # Start at the source
        stack.push(source)
        visited.add(source)

        # While there is still nodes
        while not stack.empty():
                # Top node
                node = stack.pop()

                # Traverse through node's neighbors
                for neighbor in graph[node]:
                        # If not visited then add to list
                        if neighbor not in visited:
                                stack.push(neighbor)
                                visited.add(neighbor)
```

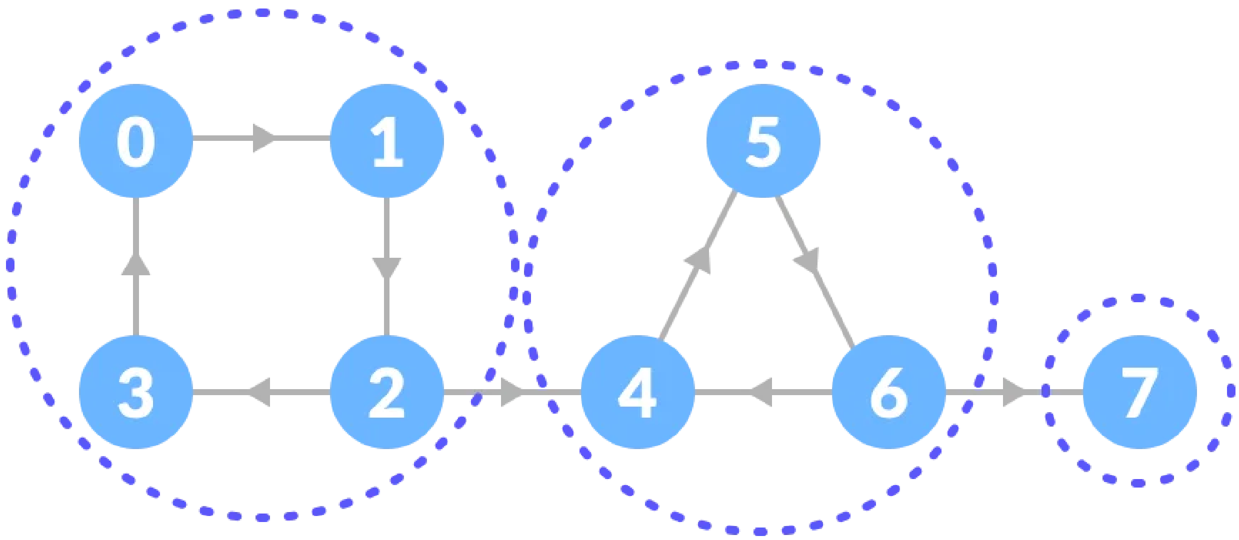
Strong Components

Strongly connected components applies only to directed graphs. A directed graph is strongly connected if there is a directed path from any vertex to every other vertex.

Example initial graph:



Strong connected components of graph:



Kosaruaju's Alogrithm

Kosaruaju's algorithm is a linear time $O(n)$ algorithm that can be used to find the number of strongly connected components.

The algorithm is based on DFS or depth first search that is done twice.

Steps:

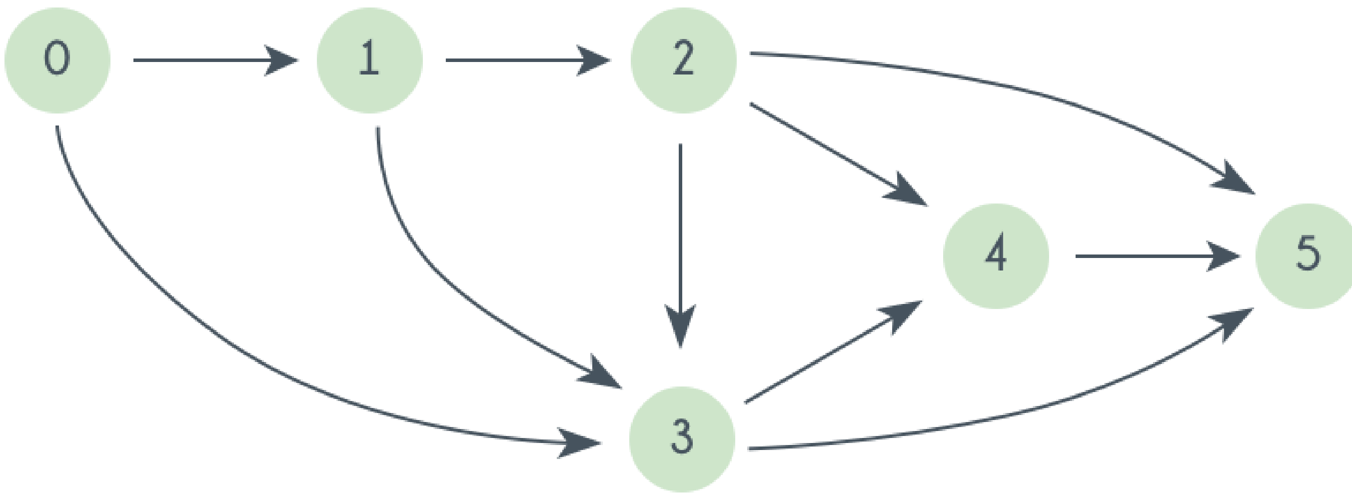
- DFS
  - As they pop we add them to a array `L = []`
  - The first DFS pass will get the finish times and add them to L
  - The second DFS pass will be reversed order of L

Psuedocode:

```
def kosaruaju(graph)
        G_reversed = dfs(graph)
        count = dfs(G_reversed)

        return count
```

Topological Sort

A sorted graph where all of the nodes point in the same direction. We can only run this on a **Directly Asyclic Graph**.



Sorted: $0, 1, 2, 3, 4, 5$

Algorithms:

- Kahn's
- DFS

DFS Python implementation

```
def topologicalSort(graph: List[List[Tuple[int, int]]]) -> List[int]:
        result = []
        visited = set()

        # Create DFS
        # View DFS above for dfs explanation
        def dfs(node: int) -> None:
                visited.add(node)

                # Traverse through node's neighbors
                for neighbor, _ in graph[node]:
                        # DFS neighbors if not visited
                        if neighbor not in visited:
                                dfs(neighbor)
                        # Add node to results
                        result.push(node)

        # Call DFS on all nodes if not visited
        for node in range(len(graph)):
                if node not in visited:
                        dfs(node)

        # Reverse list
        result.reverse()
        return result
```

1129. Shortest Path with Alternating Colors

We are given an list of edges, `redEdges` and `blueEdges`. We want to know the shortest path between 0 and all the nodes.

Example:

```
Input: n = 3, redEdges = [[0,1],[1,2]], blueEdges = []
Output: [0,1,-1]
```

BFS solution:

```
def shortestAlternatingPaths(self, n: int, redEdges: List[List[int]], blueEdges:
List[List[int]]) -> List[int]:

        neighbors = defaultdict(list)

        # Add edges into adjacency list
        for source, neighbor in redEdges:
                neighbors[source].append((neighbor, 'red'))
        for source, neighbor in blueEdges:
                neighbors[source].append((neighbor, 'blue'))

        # Default -1 for not found
        answer = [-1 for _ in range(n)]

        # Add 0 into dequeue default
        q = deque([(0, 'red', 0), (0, 'blue', 0)])
        visited = set()

        # Add 0 as first point
        visited.add((0, 'red'))
        visited.add((0, 'blue'))
        answer[0] = 0

        while q:
                # Grab first item
                node, color, distance = q.popleft()

                # Go through all neighbors
                for neighbor, neighborColor in neighbors[node]:
                        # ignore if neighbor is visited or color is the same we ignore
                        # Color same means not alternating
                        if (neighbor, neighborColor) in visited or color ==
neighborColor:
                                continue

                # If the distance is not filled -> fill
                if answer[neighbor] < 0:
                        answer[neighbor] = distance + 1
```

```python
                # Add item to queue and visited
                q.append((neighbor, neighborColor, distance + 1))
                visited.add((neighbor, neighborColor))

        return answer
```

Citations:

Breadth-first search. (2022, November 22). In *Wikipedia*. https://en.wikipedia.org/wiki/Breadth-first_search

Topological sorting. (2022, November 22). In *Wikipedia*. https://en.wikipedia.org/wiki/Topological_sorting

Kosaraju's algorithm. (2022, June 30). In *Wikipedia*. https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm