Graph

Definition: A graph is a set of vertices/nodes with edges linked them together. Let G denotes graph, then $G(V, E)$ represents a graph, where V represents vertices and E represents edges.

Varioutions of graph

A graph can be categorized using several features, including: directed, weighted, density etc.

With directed:

- Undirected graph
- Directed grah

A directed graphs(digraph) is a graph with directed edges. An Undirected graph don't have any directions. Or in other words edges in digraphs is one-direction while edges in undirected graphs are bi-direction.

With weighted:

- Weighed graph
- Unweighted graph

A weighted graph has weights on each of its edges, while unweighted graph does not.

With density:

- Sparse graph
- Dense graph

If $|E|$ roughly equals $|V|^2$, then the graph is dense. If $|E|$ roughly equals $|V|^2$, then we say the graph is sparse.

Terminology

Degree:

- Undirected graph: A degree of vertex V in an undirected graph is the number of edges associates with V.
- For directed graph, there are two types of degree: in-degree and out-degree. in-degree refers to edges pointing to V while out-degree refers to edges leaving V.

Complete graph: G is said to be complete if there exists an edge between every pair of vertices.

Subgraph: A subset of graph G.

Path: A path is a sequence of vertices between V_1 and V_k such that one can reach V_k from V_1. For directed graph, path is one-way while bi-direction in undirected graph.

Cycle: In a directed graph G, cycle refers to path connect vertex V with itself.

Acyclic: A directed graph is acyclic if there is no cycle in it.

Connected: An undirected is connected if there are edges connect every pair of vertices.

Strongly connected: A directed graph is strongly connected if every two vertices can reach each other.

Representation of graphs

There are two types of representations of graph:

- Adjacency List
- Adjacency Matrix

Implementation

In [1]:
```
import collections

'''Adjacency list'''

class Graph():
    def __init__(self):
        self.adjacency_list = collections.defaultdict(list)


    def add_edge(self, edges):
        for edge in edges:
            src, dest = edge[0], edge[1]
            self.adjacency_list[src].append(dest)
            self.adjacency_list[dest].append(src) #skip this line if you war

#Construct graph
g = Graph()
edges = [(1, 2), (1, 5), (1, 3), (2, 3), (2, 4), (2, 5), (3,4), (3,6), (4,6)
g.add_edge(edges)

print(g.adjacency_list)
```

```
defaultdict(<class 'list'>, {1: [2, 5, 3], 2: [1, 3, 4, 5], 5: [1, 2], 3: [
1, 2, 4, 6], 4: [2, 3, 6], 6: [3, 4]})
```

In [2]:
```
'''Adjacency matrix'''

class Graph():
    def __init__(self, num_vertices):
        self.adjacency_mtx = [[0] * num_vertices for x in range(num_vertices

    def add_edge(self, edges):
        for edge in edges:
            src, dest = edge[0], edge[1]
            self.adjacency_mtx[src][dest] = 1
            self.adjacency_mtx[dest][src] = 1 #skip this line if you want a

#Construct graph
g = Graph(7)
edges = [(1, 2), (1, 5), (1, 3), (2, 3), (2, 4), (2, 5), (3,4), (3,6), (4,6)
g.add_edge(edges)

for row in g.adjacency_mtx:
    print(row)
```

```
[0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 1, 0, 1, 0]
[0, 1, 0, 1, 1, 1, 0]
[0, 1, 1, 0, 1, 0, 1]
[0, 0, 1, 1, 0, 0, 1]
[0, 1, 1, 0, 0, 0, 0]
[0, 0, 0, 1, 1, 0, 0]
```

Adjacency list vs Adjacecny matrix

1. Adjacency list requires memory of $O(|V|+|E|)$ and adjacency matrix requires $O(|V|^2)$
2. Adjacency matrix can quickly determine whether two vertices i, j are connected or not (in $O(1)$) while adjacency list can't (in $O(degree(i))$).
3. Adjacency list can quickly get all vertices adjacent to vertex i (in $O(degree(i))$) while adjacency matrix can't (in $O(V)$).

Overall, adjacency list is preferred when graph is sparse ($|E| << |V|^2$), and adjacecny matrix is preferred when graph is dense($|V|^2$ and $|E|$ are roughly equal)

Breadth-first Search (BFS)

Breadth First Search (BFS) is a fundamental graph traversal algorithm. The idea is to traverse the graph by turning it into a 'tree': given a soure node (root) S, we mark it as visited and set the distance to be 0, then we traverse all its neighors(children), for each of neighor,mark it as visited and compute distance between S to current vertex, then we repeat the same process for the neighbors of current vertex until all vertices are marked as visited.

Note: we only consider adjacency list implementations

Pseudo code:

```
func BFS
    visited = []
    queue = []
    distance = []

    add s to queue
    add s to visited
    distance[s] = 0

    while queue is not empty:
        curr_vertex = queue.deque()
        for neighbor in adjacency_list:
            add neighbor to queue
            add neighbor to visited
            distance[neighbor] = distance[curr_vertex]+1
```

Implementation

```
In [3]: import math

        class Graph():
            def __init__(self) -> None:
                self.adjacency_list = collections.defaultdict(list)


            def add_edge(self, edges) -> None:
                for edge in edges:
                    src, dest = edge[0], edge[1]
                    self.adjacency_list[src].append(dest)
                    self.adjacency_list[dest].append(src) #skip this line if you war

            '''BFS implementation, return a list store distance from source node to
            def BFS(self, s) -> list:
                distance = [math.inf for i in range(len(self.adjacency_list.keys())+
                #print(distance)
                queue = collections.deque()
                visited = set()

                queue.append(s)
                visited.add(s)
                distance[s] = 0

                while(queue):
                    curr_vertex = queue.pop()
                    for node in self.adjacency_list[curr_vertex]:
                        if node in visited:
                            continue
                        queue.append(node)
                        visited.add(node)
                        distance[node] = distance[curr_vertex]+1

                return distance[1:]


        #Construct graph
        g = Graph()
        edges = [(1, 2), (1, 5), (1, 3), (2, 3), (2, 4), (2, 5), (3,4), (3,6), (4,6)
        g.add_edge(edges)
        distance = g.BFS(1)

        print("Distance from '1' to each vertex is:")
        print(distance)
```

```
Distance from '1' to each vertex is:
[0, 1, 1, 2, 1, 2]
```

Time complexity

BFS starts by visiting the source node S, then mark it as 'visited' and compute distance. Then it repeats this procedure to neighbor of S and neighbors of neighbors of S until all vertices has been marked as 'visited'. Total number of 'neighbors' in graph is $|E|$, total number of vertices is $|V|$, then we have:

$$T(n) = |V| + |E| + |E|$$

, where first $|E|$ denotes the number of operation to mark all neighbors of any vertex as 'visited' and the second one denotes number of operation of computing distance. Therefore, the overall time complexity is: $O(|V| + |E|)$

Properties

(1). BFS calculates the shortest distance(minimum steps) from source node to other node.

(2). BFS builds a Breadth-First tree, with S as root node, L_i as the each level of the tree, where i denotes the distance between any vertex V_i to S.

(3). By (2), shortest distance from S to V_i in the tree is the shortest distance between S and V_i in the entire graph.

(4). By (2) and (3), we can compute shortest distance between vertex a to b in $O(|V| + |E|)$.

Depth-first Search (DFS)

DFS is another way of traversing the graph. Unlike BFS, it traverse the graph as 'deeper' as possible. The algorithm starts at the root/source vertex(selecting some arbitrary vertex as the root/source in the case of a graph) and explores as far/deeper as possible along each branch before backtracking. Consider a tree structure, DFS will first explore the leaf node associate with the branch while BFS will first explore nodes in the next level of the current node. There are two types of DFS: recursive and non-recursive. For more info on difference between BFS and DFS, see this post:

https://stackoverflow.com/questions/687731/breadth-first-vs-depth-first

Pseudo code(Recursive):

```
func DFS:
    visited = []
    degree = []
    time = 0
    for v in G:
        if visited[v] = False:
            DFS_until(v, visited, degree, time)


func DFS_until:
    visited[v] = True
    degree[v] = ++time
    for neighbor in adjacency_list[v]:
        if visited[neighbor] = False:
            DFS_until(neighbor, visited, degree, time)
```

Pseudo code (Non-recursive):

```
func DFS:
    stack = []
    visited = []
    stack.push(s)
    while stack is not empty:
        curr_vertex = stack.pop()
        if visited[curr_vertex] = False:
            visited[curr_vertex] = True
            for neighbor in adjacency_list[v]:
                if visited[neighbor] = False:
                    stack.push(neighbor)
```

Implementation

In [4]:
```
import collections

class Graph():
    def __init__(self) -> None:
        self.adjacency_list = collections.defaultdict(list)


    def add_edge(self, edges) -> None:
        for edge in edges:
            src, dest = edge[0], edge[1]
            self.adjacency_list[src].append(dest)
            self.adjacency_list[dest].append(src) #skip this line if you war

    # Driver method
    def DFS(self) -> None:
        visited = set()
        for v in self.adjacency_list.keys():
            # If v is undiscovered, apply dfs to it
            if v not in visited:
                self.DFS_util(v, visited) #search v and its neighbors


    def DFS_util(self, src, visited) -> None:
        print(f"Marking vertex {src} as visited")
        visited.add(src)
        for neighbor in self.adjacency_list[src]:
            if neighbor not in visited:
                self.DFS_util(neighbor, visited)

        print("Vertices visited:")
        print(visited)

#Construct graph
g = Graph()
edges = [(1, 2), (1, 5), (1, 3), (2, 3), (2, 4), (2, 5), (3,4), (3,6), (4,6)
g.add_edge(edges)

g.DFS()
```

```
Marking vertex 1 as visited
Marking vertex 2 as visited
Marking vertex 3 as visited
Marking vertex 4 as visited
Marking vertex 6 as visited
Vertices visited:
{1, 2, 3, 4, 6}
Vertices visited:
{1, 2, 3, 4, 6}
Vertices visited:
{1, 2, 3, 4, 6}
Marking vertex 5 as visited
Vertices visited:
{1, 2, 3, 4, 5, 6}
Vertices visited:
{1, 2, 3, 4, 5, 6}
Vertices visited:
{1, 2, 3, 4, 5, 6}
```

Time complexity:

- $O(|V| + |E|)$ for adjacency list implementation
- $O(|V|^2)$ for adjacency matrix implementation

Properties:

(1). DFS is a backtrack algorithm.

(2). DFS can detect cycles in a directed graph.

(3). Inorder, postorder and preorder traversal of trees are all DFS.

DFS cont.

Edges

DFS introduces some import distinctions on edges:

- Tree edge: Edge that involves new/undiscovered vertices.
- Foward edge: Edge from ancestor to descendent.
- Back edge: Edge from descendent to ancestor.
- Cross edge: Edge links two nodes that do not have any ancestor/descendant relationship.

Paranthesis theorem

In any DFS of directed or undirected graph, for any two vertices u and v, one of the following three conditions holds:

- Interval $[d]$

-

White-path therorem

Vertex v is a descendant of u in a DFS tree if and only if at time $d(u)$ that u was discovered, vertex v can be reached from u along a path consisting entirely of white/unvisited vertices.

With these in mind, we can dive deep into two import applications of DFS: TOPOLOGICAL SORT and find strongly connected components.

Topological sort

Directed Acyclic Graph (DAG)

Definition: A directed acyclic graph (DAG) is a directed graph that has no cycles.

Theorem: A directed graph is a DAG iff DFS yields no back edge.

Topological Sort is a graph traversal algorithm in which each node v is visited only after all its dependencies are visited.s It's a linear ordering of vertices such that for every directed edge $u - v$, vertex u comes before v in the ordering.

Note:

- Toplogical sort can **only** be applied to DAG.

- Topological sort is a DFS-based approach.

Applications of Topological Sort:

- Task scheduling and project management (e.g. Apache Airflow).
- Dependency resolution in package management systems (e.g. SpringBoot).
- Determining the order of compilation in software build systems (e.g. Docker).
- Deadlock detection in operating systems.
- Course scheduling in universities.

Algorithm:

```
1. Call DFS on graph $G$ to compute finishing times $f(v)$
for each vertex $v$.
2. As each vertex is finished, insert it to the front/back of
the list.
3. return the resulting list(reversed order if you insert
vertices to back in step2).
```

Pseudocode:

```
func topological_sort():
    resultList = []
    visited = []
    for v in G:
        if v not in visited:
            topsort(v, resultList, visited)
    return resultList

func topsort(v, resultList, visited):
    visited.add(v)
    for neighbor in adjacencyList[v]:
        if neighbor not in visited:
            topsort(neighbor, resultList, visited)

    resultList.insertFront(v)
```

Implementation

In [5]:
```python
class DAG():
    def __init__(self) -> None:
        self.adjacency_list = collections.defaultdict(list)


    def add_edge(self, edges) -> None:
        for edge in edges:
            src, dest = edge[0], edge[1]
            self.adjacency_list[src].append(dest)


    # Driver method
    def topological_sort(self):
        topological_order = []
        visited = set()
        for v in list(self.adjacency_list.keys()):
            #print(f"Visiting vertex {v}")
            if v not in visited:
                self.topologicalsort_util(v, topological_order, visited)
        return topological_order


    def topologicalsort_util(self, v, topological_order, visited):
        visited.add(v)
        for neighbor in self.adjacency_list[v]:
            if neighbor not in visited:
                self.topologicalsort_util(neighbor, topological_order, visit

        #print(f"Node {v} finished")
        topological_order.insert(0, v)


    def print_finishtimes(self, topological_order):
        finish_time = 0
        for i in range(len(topological_order)):
            print(f"Finished time of vertex {topological_order[i]}: {finish_
            finish_time = finish_time + 1
            print()


edges = [(0, 1), (1, 2), (3, 1), (3, 2)]
dag = DAG()
dag.add_edge(edges)

topological_order = dag.topological_sort()
print(f"Topological sort of DAG is{topological_order}")
print("\n")

dag.print_finishtimes(topological_order)
```

```
Topological sort of DAG is[3, 0, 1, 2]


Finished time of vertex 3: 0

Finished time of vertex 0: 1

Finished time of vertex 1: 2

Finished time of vertex 2: 3
```

You may notice that finish time of each node is its index in the topological order.

Time complexity

Topological sort can be broken down to 2 steps: 1)Call DFS() to compute finish time of each vertex in graph. 2)Insert finished node to list. Step 1) takes running time of $O(|V|+|E|)$; step 2) takes $O(1)$ for each vertex, $O(|V|)$ in total. Overall, runnning time of topological sort is $O(|V|+|E|)$.

Correctness

Theorem: **TOPOLOGICAL-SORT** produces a topological order of the directed acyclic graph provided as its input.

Proof: Suppose (u, v) is an edge in DAG G, then during DFS call, node v can't be marked visited after v, ie, v is always marked visited before u does. This means finish time of u is always greater than finish time of v ($f(u) > f(v)$). Otherwise, it indicates there exists a back edge in G, which violets the fact that G is a DAG. Hence, TOPOLOGICAL_SORT is correct.

Strongly Connected Component

Reacall a digraph G is **Strongly connected** if every two vertices in G are reachable from each other.

Definition: Subgraph S in directed graph G is a **strongly connected component(SCC)** if it's strongly connected.

Component DAG

Definition: A **component DAG** is a DAG made up by SCCs.

Compute SCC

There are 2 ways to compute SCCs:

- Kosaraju's algorithm
- Tarjan's algorithm

In real world, Kosaraju's algorithm is more commonly used.

Kosaraju's algorithm

Kosaraju's algorithm is a linear time algorithm to find the strongly connected components of a directed graph. It uses to two-pass DFS: one on original graph G, the other on transpose of the graph G^T. The algorithm is based on a very interesting fact: If you reverse all of the edges in a digraph, the resulting graph has the same strongly connected components as the original.

Algorithm:

1. Call DFS to compute finishing times $f(u)$ for each vertex u.
2. Compute G^T
3. Call DFS on g^T to compute finishing times $f(u)$ for each vertex u.
4. Print nodes in each tree formed in step 3.

Pseudo code:

```
func DFS(G, s, finished_time):
    visited = []
    for v in G[s]:
        if v not visited:
            DFS(G, s, stack)
    finished_time.add(s)


func transpose(G):
    G_transposed  =
    for u in G.vertices:
        for v in G[u]:
            G_transposed.add_edge(v, u)

    return G_transposed


func getOneSCC(G, s, visited):
    visited.add(s)
    SCC = []
```

```
        for v in G[s]:
            if v not in visited:
                SCC.add(v)
                getOneScc(G, v)
        return SCC

    func kosaraju_SCC(G):
        finished_time = []
        for v in G:
            DFS(G, v, stack)

        G_transposed = transpose(G)
        visited = []
        SCCs = []

        while finished_time:
            node = finished_time.pop()
            SCC = getOneSCC(G, node, visited)
            SCCs.add(SCC)

        return SCCs
```

In [6]:
```
class DAG():
    def __init__(self) -> None:
        self.adjacency_list = collections.defaultdict(list)


    def add_edge(self, edges) -> None:
        for edge in edges:
            src, dest = edge[0], edge[1]
            self.adjacency_list[src].append(dest)

    #DFS to compute finish times of each node
    def DFS(self, s, finished_times, visited):
        visited.add(s)
        for v in self.adjacency_list[s]:
            if v not in visited:
                self.DFS(v, finished_times, visited)

        finished_times.append(s)


    def transpose(self):
        g_transposed = DAG()
        vertices = self.adjacency_list.keys()
        for v in vertices:
            for neighbor in self.adjacency_list[v]:
                g_transposed.adjacency_list[neighbor].append(v)

        return g_transposed


    #DFS implementation to get SCC in dfs tree
    def find_one_scc(self, s, visited, scc):
        visited.add(s)
        for v in self.adjacency_list[s]:
            if v not in visited:
                scc = self.find_one_scc(v, visited, scc)

        scc.insert(0, s)
        return scc


    #Driver method
    def kosaraju_scc(self):
        SCCs = []

        #1st pass of DFS, compute finish time
        finished_times = []
        visited = set()
        vertices = list(self.adjacency_list.keys())

        for v in vertices:
            if v not in visited:
                self.DFS(v, finished_times, visited)

        #Compute transpose of dag
        g_transpose = self.transpose()

        #2nd pass of DFS
        visited = set()
        while finished_times:
            node = finished_times.pop()
            if node not in visited:
                scc = g_transpose.find_one_scc(node, visited, [])
                SCCs.append(scc)

        return SCCs


dag = DAG()
edges = [(0, 1), (1, 2), (2, 0), (2, 8), (8, 9), (1, 3), (3, 4), (4, 5),
         (5, 6), (6, 3),(5, 7)]

dag.add_edge(edges)

SCCs = dag.kosaraju_scc()
print("The Strongly Connected Components are:")
for scc in SCCs:
    print(scc)
```

```
The Strongly Connected Components are:
[0, 2, 1]
[3, 6, 5, 4]
[7]
[8]
[9]
```

Time complexity

Kosaraju's algorithm can be broken down to 3 parts: 1) First pass DFS, 2) Transpose the Graph, 3)Second pass of DFS. Like we know, two passes of DFS take $O(|V| + |E|)$ each; transpose graph requires running time of $O(|V| + |E|)$ as well.So the total time complexity is $O(|V| + |E|)$

Correctness

Theorem: Kosaraju's algorithm computes SCCs of DAG G correctly.

Proof: Regard G as a component dag called G'. Then component C_i must have a finish time $f(C_1 = i)$, where $f(C_i)$ is the largest finish time of C_i. Likewise, C_j has $f(C_j)$ as its finish time. Assume there is an edge (u, v) starts C_i to C_2, then $f(C_i) > f(C_j)$. By definition of SCC, G' is also the component dag of G^T but with edges reversed. The second pass of DFS starts from connected component with largest finish time, let's use C_i again. For C_i to have the largest finish time, it must have **no edge comming in**, meaning that there is no edge coming out in G^T. By property of DFS, trees generated in the second pass of DFS with source node inside C_i will only include nodes within C_i. And that is one SCC we want. Same applies to other components. Therefore, the claim is correct.

For more rigorous proof, see CLRS chapter 22(page 617).

Tarjan's algorithm

Tarjan's strongly connected components algorithm is an algorithm in graph theory for finding the strongly connected components (SCCs) of a directed graph. It's based on the following facts:

1. Strongly Connected Components form subtrees of the DFS tree.
2. There is no back edge from one SCC to another.

Unlike Kosaraju's algorithm, Tarjan's algorithm only uses one-pass DFS. The critical part for Tarjan's algorithm is assigning the predesessor to each node in the DFS call, and nodes with the same predesessor form a SCC in the DAG.

Psuedo Code:

Source:
https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm

```
algorithm tarjan is
```

```
input: graph G = (V, E)
output: set of strongly connected components (sets of vertices)

index := 0
S := empty stack
for each v in V do
    if v.index is undefined then
        strongconnect(v)

function strongconnect(v)
    // Set the depth index for v to the smallest unused index
    v.index := index
    v.lowlink := index
    index := index + 1
    S.push(v)
    v.onStack := true

    // Consider successors of v
    for each (v, w) in E do
        if w.index is undefined then
            // Successor w has not yet been visited; recurse on it
            strongconnect(w)
            v.lowlink := min(v.lowlink, w.lowlink)
        else if w.onStack then
            // Successor w is in stack S and hence in the current SCC
            // If w is not on stack, then (v, w) is an edge pointing to an SCC already found and must be ignored
            // See below regarding the next line
            v.lowlink := min(v.lowlink, w.index)

    // If v is a root node, pop the stack and generate an SCC
    if v.lowlink = v.index then
        start a new strongly connected component
        repeat
            w := S.pop()
            w.onStack := false
            add w to current strongly connected component
        while w ≠ v
        output the current strongly connected component
```

For detailed implementation, checkout https://rosettacode.org/wiki/Tarjan#Java.

Time complexity

$O(|V| + |E|)$

Applications

1. Social Networks
2.