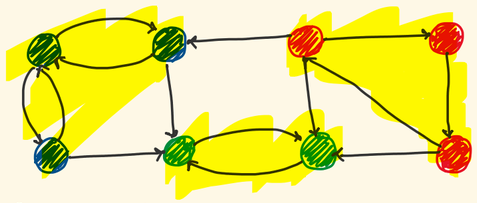
Strongly connected components

Friday, October 27, 2023 5:44 PM

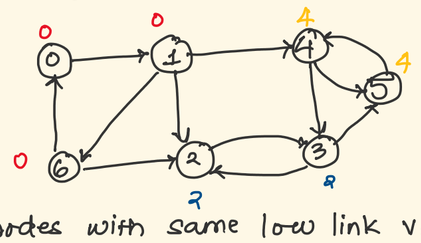
- Self contained cycles within a directed graph.
- Every vertex in the cycle can reach every other vertex in the same cycle.



Once we leave the cycle, there is no coming back

Low link values

- Value of the lowest id reachable from a node.



- All nodes with same low link val belong to the same scc.
- DFS cannot be used to determine Low link vals.
  ↳ Due to randomness of traversal.

Tarjan's Strongly connected components.

- To cope with the randomness of DFS, we use a stack to track connected components.
- Start DFS from any unvisited node.
- Give it a uniq ID & push it to stack.
- Also assign itself as the low link.
- Visit all its unvisited neighbors.
- During the callback get the min low link & update.
  low[node] = min(low[node], low[childI])
- After exploring all child nodes, pop the component from the stack.

This can be only done if the child is being visited ↔ This denotes that it is in the cycle.

```
'''
Approach:
- Maintain a stack, a being_visited array
- Maintain ids and low_link arrays
- While doing a dfs, assign a unique id to each node
- Mark it as being visited
- While exploring its children if they are unvisited dfs into them
- During the callback if the child is in the current cycle (being_visited), update current node's low_link with the minimum of itself and the child's low_link val
- Finally, pop all nodes from the stack until stack.top = node (start of scc)
'''

# O(V + E) Time
def dfs(i):
    stack.append(i)
    being_visited[i] = True   <-- for tracking cycles
    ids[i] = idx
    low[i] = idx
    idx += 1

    for j in adj[i]:
        if ids[j] == -1: dfs(j)
        if being_visited[j]: low[i] = min(low[i], low[j]) # in a cycle

    if ids[i] == low[i]: found the start of an scc
        node = None
        while node != i:
            node = stack.pop()
            being_visited[node] = False
            low[node] = ids[i]
    scc_count += 1
```

This part is purely for storing the components (Not reqd if we just want to count scc.)

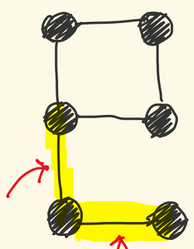
Kosaraju's Algorithm

```
'''
Approach:
- Perform DFS on any unvisited node
- Explore all its unvisited childred
- During the callback, push the node on to the stack
- Reverse the graph (take a transpose)
- pop all visited nodes from the stack
- explore all the unvisited nodes from the stack
- Store the components
'''

# O(V + E)
def dfs_1(i):
    visited[i] = True
    for j in range(n):
        if adj_mat[i][j] and not visited[j]: dfs_1(j)
    stack.append(i)

def transpose():
    for i in range(n):
        for j in range(i):
            adj_mat[i][j], adj_mat[j][i] = adj_mat[j][i], adj_mat[i][j]

def dfs_2(i):
    visited[i] = True
    components[i] = numComponents
    for j in range(n):
        if adj_mat[i][j] and not visited[x]: dfs(j)
```

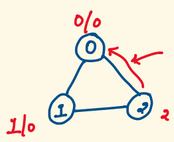
Topological sorting

Bridges:

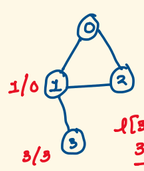
Edges in the graph, which increase the number of components when removed.



Key condition. if low[child] <= tin[node]
(reachable by other path



l[0] <= id[2]
0 <= 2
0 can reach 2 (eventually)



l[3] > id[1]
3 > 1

There is no way of ③ to reach ① if 3—1 is removed

id: is actually dfs discovery time
low: is the lowest discovery time of the entire components.

```
'''
bridges: edges in the graph that increase the number of components when removed.

Approach:
- assign a time of discovery and a low link val to each node.
- on a callback, record the minimum low link value. (represents the lowest discovery time needed to visit the entire component)
- check if the edge can be a bridge during the callback
    - if low_link[child] <= time_of_discovery[node] --> This means that edge cannot be bridge (the child can eevntually reach the node via some other node since its low_link is smaller)
    - if low_link[child] > time_of_discovery[node] --> This means that this edge can be a bridge since there is no way for child to reach node once the edge is removed.
'''

timer = 0
bridges = []

def dfs(i, p):
    visited[i] = 1
    td[i] = timer
    low[i] = timer
    timer += 1

    for j in adj[i]:
        if j == p: continue
        if not visited[j]:
            dfs(j, i)
            low[i] = min(low[i], low[j])   <-- update low link
            if low[j] > td[i]: bridges.append((i, j))
        else:
            low[i] = min(low[i], low[j])
```

Articulation point.

Points in the graph which increase the no. of components when removed.

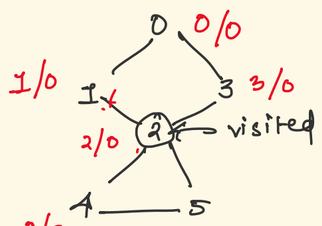
Condition for bridges can be slightly modified

low[child] < tin[node]
↑ Notice it is not '<='.

Because the node is going to be removed, we want to reach a node which was seen before that

Another difference.

if a child is visited, we take its time of discovery for updating low



since 2 is visited

low[4] = min(4, id[2])
Cannot be 0 since it can never reach 0 if 2 is removed.

```
'''
Articulation point: points in the graph that increase the number of components when removed.
'''

articulation = []
idx = 0

def dfs(i, p):
    visited.add(i)

    td[i] = idx
    low[i] = idx
    for j in adj[i]:
        if j == p: continue
        if j not in visited:
            dfs(j, i)
            low[i] = min(low[i], low[j])
            if low[j] >= td[i] and p != -1: articulation.append(i) # can reach its parent
        else:
            low[i] = min(low[i], td[j])
```

Makes sure that a root does not become an articulation pt.