Department of Computer Sciences

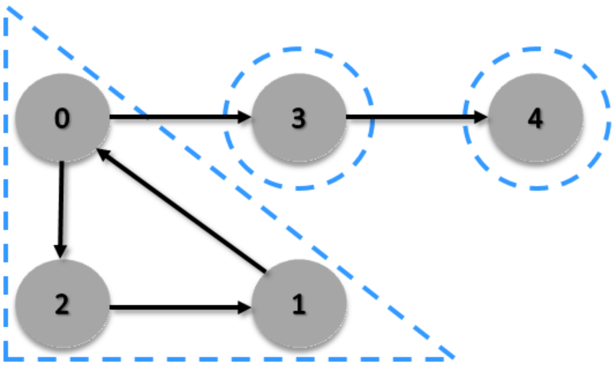
<u>CS-1005- Discrete Structures</u>

Semester: Fall 2022

Course Project

The aim of the project to compute the size of Strongly Connected Component SCC in a given directed graph. A strongly connected component is a partition of a directed graph in which there is a path from each vertex to another vertex in the partition. This is applied only on **Directed graphs**.

For example following graph contains 3 SCCs :



You will be using the dataset given at the links given below.

1. **https://snap.stanford.edu/data/web-Google.html**
2. https://snap.stanford.edu/data/ego-Twitter.html

Problem

Given a directed graph $G = (V, E)$, output <u>all</u> its strong connected components.

Straightforward algorithm:

Mark all vertices in V as not visited.
for each vertex $u \in V$ not visited yet **do**
    find SCC(G, u) the strong component of u:
        Compute rch(G, u) using $DFS(G, u)$
        Compute rch(G^{rev}, u) using $DFS(G^{\mathrm{rev}}, u)$
        SCC$(G, u) \Leftarrow$ rch$(G, u) \cap$ rch(G^{rev}, u)
        $\forall u \in$ SCC(G, u): Mark u as visited.

Running time: $O(n(n + m))$

You will be calculating the SCC as per above algorithms and reporting the size of the largest SCC as output. You can program the solution in any programming language you prefer. You will be submitting the code of the project along with the screenshot obtained results and 3 min video discussing the code along with the results.

The following two functions must be programed by yourself and no ready-made library may be used for these two functions. You must also share the link of the code resources that you have incorporated in your project.

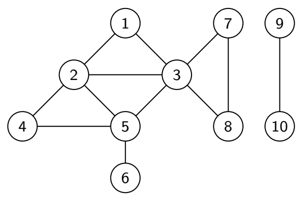
1. Depth First Search DFS
2. Computation of SCC

The description of various components such as *DFS* and *rch* is given below. Additionally, a brief summary of the related project along with relevant theorems and propositions is available below. . These topics are also discussed during the lectures.

Why Graphs?

- Graphs help model networks which are ubiquitous: transportation networks (rail, roads, airways), social networks (interpersonal relationships), information networks (web page links) etc.
- Fundamental objects in Computer Science, Optimization, Combinatorics
- Many important and useful optimization problems are graph problems
- Graph theory: elegant, fun and deep mathematics

Graph



Definition *An undirected (simple) graph* $G = (V, E)$ *is a 2-tuple:*

- *V is a set of vertices (also referred to as nodes/points)*
- *E is a set of edges where each edge* $e \in E$ *is a set of the form* $\{u, v\}$ *with* $u, v \in V$

Example *In figure,* $G = (V, E)$ *where* $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$ *and* $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{3, 8\}, \{4, 5\}, \{5, 6\}, \{7, 8\}\}$.

Notation

An edge in an undirected graph is an *unordered* pair of nodes and hence it is a set.

Conventionally we use (u, v) for $\{u, v\}$ when it is clear from the context that the graph is undirected.

- *u* and *v* are the *end points* of an edge $\{u, v\}$
- *Multi-graphs* allow
  - *loops* which are edges with the same node appearing as both end points
  - *multi-edges*: different edges between same pairs of nodes
- In this course we will assume that a graph is a simple graph unless explicitly stated otherwise.

Common Data Structures for Graphs

Adjacency Matrix *$n \times n$ asymmetric* matrix A. $A[u, v] = 1$ if (u, v) 2 $\in$ and $A[u, v] = 0$ if $(u, v) \notin E$. $A[u, v]$ is not same as $A[v, u]$.

Adjacency Lists for each node u, $Out(u)$ (also referred to as $Adj(u)$) and $In(u)$ store out-going edges and in-coming edges from u.

Default representation is adjacency lists.

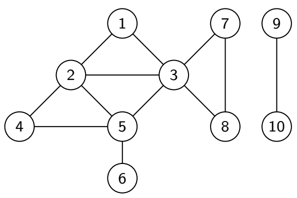
- For each $u \in V$, $\text{Adj}(u) = \{v \mid \{u, v\} \in E\}$, that is neighbors of u. Sometimes $\text{Adj}(u)$ is the list of edges incident to u.

Note: The above given link for dataset assumes that graphs are represented using adjacency lists.

Connectivity

Given a graph $G = (V, E)$:

- A *path* is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \le i \le k - 1$. The length of the path is $k - 1$ and the path is from v_1 to v_k
- A *cycle* is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \le i \le k - 1$ and $\{v_1, v_k\} \in E$.
- A vertex u is *connected* to v if there is a path from u to v.
- The *connected component* of u, con(u), is the set of all vertices connected to u.



Define a relation C on $V \times V$ as uCv if u is connected to v

- In undirected graphs, connectivity is a reflexive, symmetric, and transitive relation. Connected components are the equivalence classes.
- Graph is *connected* if only one connected component

Connectivity Problems/ Algorithmic Problems

- Given graph G and nodes u and v, is u *connected* to v?
- Given G and node u, find all nodes that are connected to u.
- Find all connected components of G.

Basic Graph Search

Given $G = (V, E)$ and vertex $u \in V$:

Explore(u):

Initialize $S = \{u\}$

While there is an edge (x, y) with $x \in S$ and $y \notin S$

add y to S

Proposition *Explore*(u) *terminates with* $S = con(u)$.

Depth First Search

DFS is a very versatile graph exploration strategy. Hopcroft and Tarjan (Turing Award winners) demonstrated the power of **DFS** to understand graph structure. **DFS** can be used for

- Finding cut-edges and cut-vertices of undirected graphs
- Finding strong connected components of directed graphs
- Linear time algorithm for testing whether a graph is planar

DFS in Undirected Graphs

Recursive version.

```
DFS(u)
    Mark u as visited
    for each edge (u,v) in Ajd(u)do if v is not marked
                        DFS(v)
```

Global array Mark for all recursive calls.

DFS Tree/Forest

```
DFS(G)
    Mark all nodes u as unvisited
    T is set to ∅
    While there is an unvisited node u do
        DFS(u)
    Output T
```

```
DFS(u)
    Mark u as visited
    for each edge (u,v) in Ajd(u) do
        if v is not marked
            add edge (u,v) to T
            DFS(v)
```

Edges classified into two types: (u, v) E is a$\in$

- *tree edge:* belongs to T
- *non-tree edge:* does not belong to T

Properties of DFS tree

Proposition *T is a forest and connected components of T are same as those of G.*

- *If (u, v) is a non-tree edge then, in T, either u is an ancestor of v or v is an ancestor of u.*

DFS with Visit Times

Keep track of when nodes are visited.

```
DFS(G)
    Mark all nodes u as unvisited
    T is set to ∅
    time = 0
    While there is an unvisited node u do
        DFS(u)
    Output T
```

Pre and Post numbers

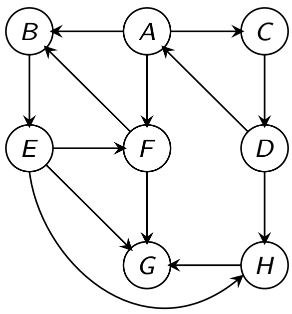
Node u is *active* in time interval [pre(u), post(u)]

Proposition 0.3.5 *For any two nodes u and v, the two intervals* [pre(u), post(u)] *and* [pre(v), post(v)] *are disjoint or one is contained in the other.*

Proof:

- Assume without loss of generality that pre(u) < pre(v). Implies v visited after u.
- If **DFS**(v) invoked before **DFS**(u) finished, post(u) > post(v).
- If **DFS**(v) invoked after **DFS**(u) finished, pre(v) > post(u).

Directed Graphs



Definition 0.4.1 *A directed graph G = (V, E) consists of*

- *set of vertices/nodes V and*
- *a set of edges/arcs $E \subseteq V \times V$.*

An edge is an *ordered* pair of vertices. (u, v) different from (v, u).

Examples of Directed Graphs

In many situations relationship between vertices is asymmetric:

- Road networks with one-way streets. Web-link graph: vertices are web-pages and there is
- an edge from page p to page q if p has a link to q. Web graphs used by Google with PageRank algorithm to rank pages.

- Dependency graphs in variety of applications: link from x to y if y depends on x. Make files for compiling programs.
- Program Analysis: functions/procedures are vertices and there is an edge from x to y if x calls y.

Directed Connectivity

Given a graph $G = (V, E)$:

- A ***(directed) path*** is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$. The length of the path is $k - 1$ and the path is from v_1 to v_k

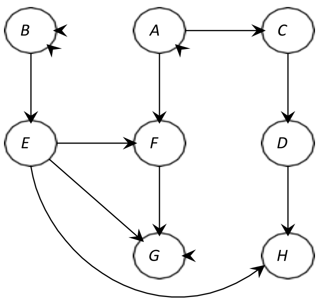
- A ***cycle*** is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$ and (v_k, v_1) 2 E.

- A vertex u can *reach* v if there is a path from u to v. Alternatively v can be reached from u

- Let **rch**(u) be the set of all vertices reachable from u.

Connectivity contd

Asymmetricity: A can reach *B* but *B* cannot reach *A*



Connectivity and Strong Connected Components

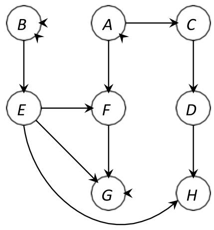
Definition *Given a directed graph G, u is strongly connected to v if u can reach v* and *v can reach u. In other words, $v \in rch(u)$ and $u \in rch(v)$.*

Define relation C where uCv if u is (strongly) connected to v.

Proposition *C is an equivalence relation, that is reflexive, symmetric andtransitive.*

Equivalence classes of C: *strong connected components* of G. They partition the vertices of G SCC(u): strongly connected component containing u.

Strongly Connected Components: Example



Directed Graph Connectivity Problems

- Given G and nodes u and v, can u reach v?
- Given G and u, compute rch(u).
- Given G and u, compute all v that can reach u, that is all v such that u 2 rch(v).
- Find the strongly connected component containing node u, that is SCC(u).
- Is G strongly connected (a single strong component)?
- Compute *all* strongly connected components of G.

First four problems can be solve in $O(n + m)$ time by adapting **BFS**/**DFS** to directed graphs. The last one requires a clever **DFS** based algorithm.

DFS in Directed Graphs

```
DFS(G)
    Mark all nodes u as unvisited
    T is set to ∅
    time = 0
    While there is an unvisited node u do
        DFS(u)

    Output T
```

```
DFS(u)
    Mark u as visited
    pre(u) = ++time
    for each edge (u,v) in Out(u) do
        if v is not marked
            add edge (u,v) to T
            DFS(v)
    post(u) = ++time
```

DFS Properties

Generalizing ideas from undirected graphs:

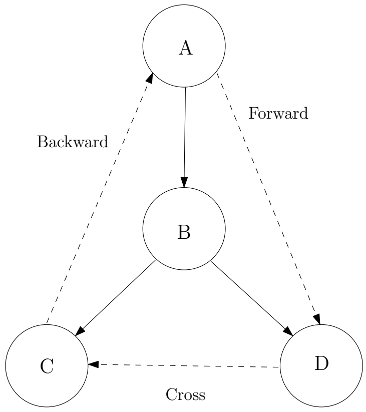
- *DFS*(u) outputs a directed out-tree T rooted at u
- A vertex v is in T if and only if $v \in$ rch(u)
- For any two vertices x, y the intervals [pre(x), post(x)] and [pre(y), post(y)] are either disjoint are one is contained in the other.

DFS Tree

Edges of G can be classified with respect to the **DFS** tree T as:

- ***Tree edges*** that belong to T
- A ***forward edge*** is a non-tree edges (x, y) such that $pre(x) < pre(y) < post(y) < post(x)$.
- A ***backward edge*** is a non-tree edge (x, y) such that $pre(y) < pre(x) < post(x) < post(y)$.
- A ***cross edge*** is a non-tree edges (x, y) such that the intervals $[pre(x), post(x)]$ and $[pre(y), post(y)]$ are disjoint.

Types of Edges



Directed Graph Connectivity Problems

- Given G and nodes u and v, can u reach v?
- Given G and u, compute $rch(u)$.
- Given G and u, compute all v that can reach u, that is all v such that $u \in rch(v)$.
- Find the strongly connected component containing node u, that is $SCC(u)$.
- Is G strongly connected (a single strong component)?
- Compute *all* strongly connected components of G.

Algorithms via DFS- I

- Given G and nodes u and v, can u reach v?
- Given G and u, compute $rch(u)$.

Proposition For any graph G, the graph of SCCs of G^{rev} is the same as the reversal of G^{SCC}.

Proposition For any graph G, the graph G^{SCC} has no directed cycle.

Directed Acyclic Graph
A directed graph G is a directed acyclic graph (DAG) if there is no directed cycle in G

A vertex u is a source if it has no in-coming edges. A vertex u is a sink if it has no out-going edges.