3.1 Eulerian Circuits

Recall the Königsberg bridge problem we discussed in the first class. The problem essentially reduces to whether or not its possible to begin at some vertex, traverse every edge exactly once, and return to that starting vertex. In other words, a closed trail exists on G that contains all $e \in E(G)$.

A graph is **Eulerian** if such a trail exists. A closed trail is a **circuit** when there isn't any specific start/end vertex specified. An **Eulerian circuit** in a graph is the circuit or trail containing all edges. An **Eulerian path** in a graph is a path containing all edges, but isn't closed, i.e., doesn't start or end at the same vertex.

Prove: If every vertex in G has at least a degree of 2, then G has a cycle.

Prove: A graph is Eulerian iff it has at most one nontrivial component and is an even graph.

How might we find an Eulerian circuit?

Fleury's algorithm:

$T \leftarrow \emptyset$ ▷ Initialize Eulerian circuit
$G' \leftarrow G$
Start at any vertex v
while $G' \neq \emptyset$ **do**
    Select at edge e to travel along, where $(G' - e)$ is not disconnected
    $T \leftarrow e$
    $G' \leftarrow (G' - e)$
return T

Hierholzer's algorithm:

$T \leftarrow \emptyset$ ▷ Initialize Eulerian circuit
Select at any vertex v
$T \leftarrow$ randomly traverse unvisited edges until you arrive back at v
$G' \leftarrow G - T$
while $G' \neq \emptyset$ **do**
    Select any vertex u in T that has incident edges remaining in G'
    $P \leftarrow$ randomly traverse unvisited edges in G' until you arrive back at u
    $T \leftarrow P$ ▷ Insert new path into circuit
    $G' \leftarrow G - P$
return T

What is the computational complexity of each approach?

3.2 Degrees

As mentioned previously, we're going to use variables n and m regularly as:

$$n = |V(G)|, m = |E(G)|$$

The **degree** of a vertex is the number of incident edges. We write degree of vertex v_i as $d(v_i)$ or sometimes d_i. For a graph G, the maximum degree is $\Delta(G)$ and the minimum degree is $\delta(G)$. A graph is **regular** if $\Delta(G) = \delta(G)$. A graph is **k-regular** if $k = \Delta(G) = \delta(G)$.

The degree sum formula shows that the sum of the degrees of all vertices in a graph is always even:

$$\sum_{v \in V(G)} d(v) = 2m$$

So it follows that there can only be an even number of vertices of odd degree in G.

The average degree of a graph G is $\frac{2m}{n}$. Therefore:

$$\delta(G) \leq \frac{2m}{n} \leq \Delta(G)$$

A **hypercube**, or **k-dimensional hypercube** Q_k is a simple graph whose vertices are k-tuples of $\{0, 1\}$ and whose edges are the pairs of k-tuples that differ by one.

Prove a hypercube is a (regular) bipartite graph.

Prove that any k-regular bipartite graph has the same number of vertices in each partite set.

3.3 Extremal Problems

An **extremal problem** asks for the maximum or minimum value of a function over a class of objects. We'll do a couple extremal proofs related to degrees and connectivity.

Prove the minimum number of edges in a connected graph is $(n - 1)$.

Prove a graph must be connected if $\delta(G) \geq \frac{(n-1)}{2}$.

We'll often use *extremal arguments* (commonly called the **extremal principle**) as a proof technique through the course. Recall from the first proof we worked through today – "Let P be a *maximal* path in G". We'll consider many other minimal or maximal graphs, subgraphs, and properties as methods to solve various proofs.

3.4 Graphic Sequences

The **degree sequence** of a graph is the list of vertex degrees, usually in non-increasing order: $d_1 \geq d_2 \geq \ldots \geq d_n$.

A **graphic sequence** is a list of nonnegative numbers that is the degree sequence of a simple graph. A simple graph G with degree sequence S *realizes* S.

A sequence $S = \{d_1, d_2, \ldots, d_n\}$ is a graphic sequence iff sequence $S' = \{d_2 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n\}$ is a graphic sequence, where $d_1 \geq d_2 \geq \ldots \geq d_n$ and $n \geq 2$ and $d_1 \geq 1$. This is called the **Havel-Hakimi Theorem**. We can use this general idea to also create (*realize*) a graph using a given graphic sequence.

For time consideration, we're not going to go over the proof in class, so go through the book or use other online resources to understand it. A couple relevant youtube videos are also listed below. They're also linked to on the course website:

`https://www.youtube.com/watch?v=aNKO4ttWmcU`
`https://www.youtube.com/watch?v=iQJ1PFZ4gh0`

3.5 Directed Graphs

Before we were only considering graphs with symmetric relations in the edges. Now we're considering **directed graphs** or **digraphs**, where the edges have a defined directionality. The vertex where an edge starts is the **tail** and the vertex that is pointed to is the **head**. These together are the **endpoints**. We also term the tail as the **predecessor** of the head and the head as the **successor** of the tail.

Like with our undirected graphs. We can consider digraphs as **simple digraphs** if they don't have repeated edges or loops. Note that a simple digraph can have two edges between the same two vertices as long as they point in opposite directions.

We have similar definitions in directed graphs for **walks**, **paths**, **trails**, and **cycles**. Likewise, we have the same concepts of **subgraphs**, **isomorphism**. The **adjacency** matrix is created in a similar row-wise fashion, except now it is now longer symmetric.

Instead of just degree, digraphs consider both **out degree** ($d^+(v)$) or **in degree** ($d^-(v)$).

We also have the out neighborhood ($N^+(v)$) or successor set and the in neighborhood ($N^-(v)$) or predecessor set. Likewise minimum and maximum out and in degrees:

$$\delta^-(v), \delta^+(v), \Delta^+(v), \Delta^-(v)$$

And our degree sum formula:

$$\sum_{v \in V(G)} d^+(v) = |E(G)| = \sum_{v \in V(G)} d^-(v)$$

3.6 Directed Connectivity

For digraphs, we have the concepts of **strong connectivity** and **weak connectivity**. The definition of strong connectivity is similar to connectivity in undirected graphs: for any u, v is a strongly connected component, there exists a directed path from u to v. Weak connectivity of a directed graph is equivalent to connectivity of its underlying graph, where the **underlying graph** of a digraph is the undirected representation created by removing directionality from the directed edges.

3.6.1 Connectivity Algorithms

The optimal (serial) algorithm for determining all maximal strongly connected components in a graph is Tarjan's Algorithm. The algorithm completes one DFS of the graph, labeling each vertex with two labels. The first label is the *index*, or the DFS order. The second label, *lowlink* is the *index* of the lowest index vertex that a given vertex can reach following its out edges. As the algorithm completes one full DFS traversal where we visit every vertex and every edge exactly once, we can state that the work complexity is $O(|V| + |E|)$. The algorithm is given below (note the similarity of the outer loop to our connectivity algorithms).

There is also **Kosaraju's Algorithm**, which is also optimal in terms of work complexity. However, this algorithm requires two traversals of all edges (or one of both out edges and in edges) so it isn't as efficient in practice. For weak connectivity, undirected connectivity algorithms can be used but with the edges mirrored (an out edge becomes out and in edges between same vertex endpoints) to create the underlying graph.

```
for all v ∈ V(G) do
    index(v)← −1                ▷ Assume all arrays and variables are globally accessible
    lowlink(v)← −1
    onstack(v)←false
curIndex ← 1                                                    ▷ DFS order counter
Stack ← ∅                                                       ▷ DFS stack
SCC ← ∅                                                         ▷ Sets of SCCs
for all v ∈ V(G) do
    if index(v)< 0 then
        tarjan(v)
return SCC
```

```
procedure TARJAN(v)
    index(v)← curIndex
    lowlink(v)← curIndex
    onstack(v)←true
    curIndex ← curIndex + 1
    Stack.push(v)
    for all u ∈ N+(v) do
        if index(u)= −1 then
            tarjan(u)
            lowlink(v)=min(lowlink(u), lowlink(v))
        else if onstack(u)=true then
            lowlink(v)=min(index(u), lowlink(v))
    if lowlink(v)=index(v) then                                 ▷ v is root of new SCC
        while (w = Stack.pop(v))≠ v do
            onstack(w)←false
            SCC(index(v))← w
    return
```