Graph Terminology (I)

- A graph is defined by a set of vertices $V$ and a set of edges $E$.
- The edge set must work over the defined vertices in the vertex set.
- Many different types of relationships can be represented as graphs.
- Graphs (specifically edges) can be either directed (eg: driving on a street) or undirected (eg: walking on a street).
- If two vertices are connected by an edge, we say those vertices are adjacent to each other.
- Edges can have values associated with them, in which case we call the graph a weighted graph.
Graph Terminology (II)

- A **path** is a list of edges that are sequentially connected. The **length** of a path is the number of edges. We will say that vertex $b$ is **reachable** from $a$ if there is a path from $a$ to $b$.
- A **cycle** is a path where the starting vertex is also the ending vertex.
- A **Hamiltonian Path** is a path that visits every vertex in a graph **exactly once**.
- An **Eulerian Path** is a path that visits every edge in a graph **exactly once**.

Graph Representation (Directed)

- $V = \{A, B, C, D, E\}$
- $E = \{(A, B), (A, D), (B, C), (C, D), (D, A), (E, D), (E, E)\}$
A Proof on Graphs

Definitions: In a directed graph, the in-degree of a vertex is the number of edges going into it and the out-degree of a vertex is the number of edges coming out of it.

Theorem:
\[ \sum_{v \in V} \text{in-degree}(v) = \sum_{v \in V} \text{out-degree}(v) \]

Proof will be by induction. Start with a base case of a graph with a single edge. For the inductive hypothesis, say that for any graph with an edge set of size \( k \) that the theorem holds. Then show that it holds for any graph with an edge set of size \( k+1 \).
**Inductive Hypothesis**

For any graph with edge set of size $k$ ($k \geq 1$)

\[
\sum_{v \in V} \text{in-deg}(v) = \sum_{v \in V} \text{out-deg}(v)
\]

**Inductive Step**

Show that for any graph $G$ with edge set of size $k+1$ that

\[
\sum_{v \in G,V} \text{in-deg}(v) = \sum_{v \in G,V} \text{out-deg}(v)
\]

Let $H$ be a generic particular graph with $k+1$ edges.

Select an edge (call it $e_1$) and remove it from the edge set to create a new graph $H'$.

By our definitions,

\[
\sum_{v \in H} \text{in-deg}(v) = \sum_{v \in H} \text{in-deg}(v) + 1
\]

\[
\sum_{v \in H} \text{out-deg}(v) = \sum_{v \in H} \text{out-deg}(v) + 1
\]

Can now ask:

\[
\sum_{v \in H'} \text{in-deg}(v) + 1 = \sum_{v \in H'} \text{out-deg}(v) + 1
\]

---

**Classical Graph Problem**

**Königsberg Bridge Problem**

**Seven Bridges. How can you cross all seven without re-using any?**
How can we solve Eulerian path?

If all we want is a yes/no answer, it’s fairly easy.

If we also want to find the actual path if it exists, that becomes a much more involved question…

For one point, we need to think about algorithms that are able to traverse graphs. So, let’s look at one…

Breadth-First Search

Given a graph, one way to have an algorithm try to visit every vertex in that graph is via a breadth-first search.

– Select a starting point.
– Visit all vertices that are “one jump” away from it.
– Visit all vertices that are “two jumps” away from it.
– etc.

What if the graph is directed?

If the graph is not connected, what ends up happening?

A simple problem that can be solved using this general technique is that of finding the shortest path between two vertices in an undirected and unweighted graph.
**Shortest Path via BFS**

Starting at vertex \( s \in V \) generate an array of distances from \( s \) called \( \text{dist[]} \) such that \( \forall v \in V \),
\[
\text{dist}[v] = \text{length of shortest path from } s \text{ to } v.
\]
\[
\text{dist}[s] = 0
\]
We will also create a predecessor array of the last vertex we were at before getting to the end of the path from \( s \) to \( v \)
\[
\forall v \in V, \ \text{pred}[v] = \text{"one step back"}
\]
\[
\text{pred}[s] = \text{none}
\]

With just these two arrays, we will be able to reconstruct any shortest path request from \( s \) to some vertex.

This is because any sub-path of the optimal path must also be an optimal path between its own endpoints.

If it weren’t, then we could have replaced it and gotten a shorter overall path.

**Basic Pseudocode**

Start at \( s \).

For each neighbor \( v \) of \( s \)
\[
\text{dist}[v] = 1
\]
\[
\text{pred}[v] = s.
\]

Move outwards from each neighbor you’ve seen and set the next “ripple” out as “+1” of the current distance, and set \( \text{pred[]} \) appropriately.

Need a way to make sure we don’t end up in cycles!
Avoiding Cycles

We will assign a color to each vertex based on the following rules:
- white = not seen yet at all
- gray = seen but not processed yet
- black = processed

We will create a queue of gray vertices, and will never add any vertex to the queue more than once.

When we are done processing a vertex (ie: we have touched all its neighbors) we go back to the queue to get the next vertex to process.

More Detailed Pseudocode

BFS (Graph G, vertex s) {
    int size = G.getVertexCount;
    int dist = new int[size];
    vertex pred = new int[size];
    Queue Q= new Queue<vertex>;
    Colors state = new Colors[size];
    for each v in G.V {
        state[v]=white; dist[v]=infinity; pred[v]=none;
    }
    state[s]=gray;  dist[s]=0;  pred[s]=none; Q.add(s);
    while (!Q.empty()){  
        u=Q.remove();
        for each unvisited v in G.Adj(u) {
            state[v]=gray;  
            dist[v]=dist[u]+1;  
            pred[v]=u;
            Q.add(v);
        }
        state[u]=black;
    }
}
What’s the runtime?

Each vertex gets enqueued at most one time, so each is processed at most one time.
  – Write this up using a summation to represent the processing of all of the vertices…

Our runtime will be order:
  |V| for all of the *initializations*

The *while* loop’s cost can be seen as the sum across all vertices $u$ in $V$ of:
  - the degree($u$) for work inside the *for* loop
  - “+1” for the work outside of the *for* loop

We can split the summation into two simpler ones and if you work it through, the runtime is $O(|V|+|E|)$.

What else does BFS give us?

It allows us to organize the entire graph as “ripples” away from a central point.
  – This could be useful if we could restate other questions within this framework.

Our predecessor array could be used to create a tree rooted at source $s$ of vertices that can be reached from $s$.
  – This is often called a breadth-first tree.
  – If we could phrase a problem as a traversal of this tree…
Depth-First Search

You could basically just change the Queue in the BFS code into a Stack.

You could also just write it out as a recursive algorithm.

This approach can also be used to determine what vertices are reachable in $O(|E|+|V|)$ time.

DFS on a Directed Graph with “Timing” Info

We can add more arrays and store information such as when (in terms of a continuously advancing ticker) each vertex is first visited (enter) and finally processed (exit).

Even in a connected graph, we might end up having to build a forest of trees to give every vertex a set of times.

– After doing a DFS from a given starting point, if there are vertices with no times, choose one of them, and continue.
Topological Sort of a Digraph

NOTE: This only works if there are no cycles, since if there are cycles there isn’t the notion of a sorted order.

Imagine a graph as beads where the edges are strings of equal length connecting ordered pairs of beads.

You want to arrange the beads so that all edges point left-to-right.

How can you use a DFS with “timing” info to accomplish this?

– Perform the DFS with timing and then “sort” by listing the nodes in reverse order based on the exit times.
**Strongly Connected Components**

We define “strongly connected” to mean that for every pair of vertices \((u,v)\) in the component, there is a path from \(u\) to \(v\) and from \(v\) to \(u\).

In the following graph, what are the strongly connected components?

```
A -> B -> C <-> D
  \
  \  \         \
   \  \       \
    E <-> F <-> G --> H
```

**Finding the SCCs**

Step 1: Perform a DFS with “timing” on the graph \(G\).

Step 2: Perform a DFS with “timing” on the graph \(G^T\) with the added restriction that when you have a choice of vertices, you choose the one with the largest finish time from Step 1’s search.

Every time your algorithm hits a dead-end, you have finished one strongly connected component and are ready to start finding the next one.

Let’s trace this on the graph from the previous slide…
Could you use a BFS or DFS to…

Detect whether a given graph has any cycles?
   – Yes.

Determine whether every vertex is reachable from a particular vertex in a given graph?
   – Yes.

Find the longest simple path through a graph between two vertices in an unweighted graph that might contain cycles?
   – No!