Graph Notes

## Graphs

Graph Types

* Undirected Graphs
* Directed Acyclic Graphs (DAGs)
* Complete Graphs
  + Every node has an edge to all other nodes in the graph.
* Bipartite Graphs
  + Even length cycles.

Representing Graphs

Representing Graphs

* Adjacency Matrix
  + Space Complexity: O(V^2)
* Adjacency List
  + Space Complexity: O(E) -> O(E^2)
* Edge List
  + (u, v, w)
  + <https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html>
* Grid(s)

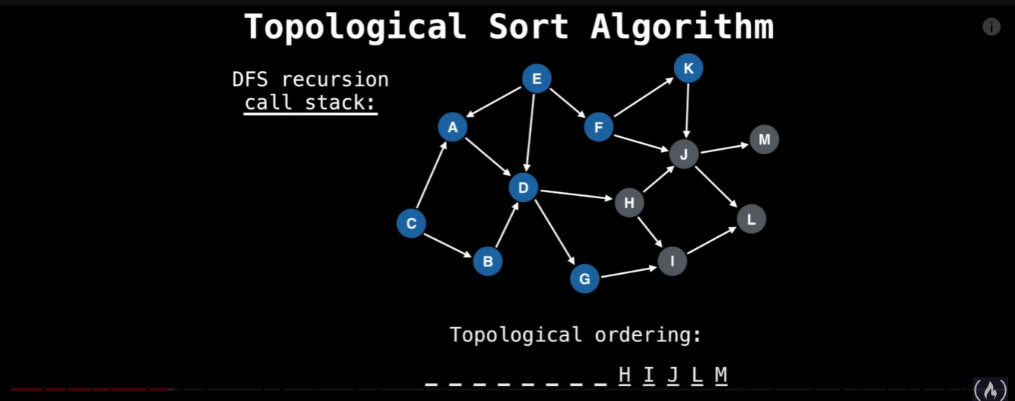
Common Graph Problems

* Shortest Path
  + Dijkstra’s
  + Bellman-Ford
* Connectivity / Strongly Connected Components
  + Tarjan
* Negative Cycle Detection
  + Bellman-Ford
  + Floyd-Warshall
* Minimum Spanning Tree (MST)
  + Kruskal’s
  + Prim’s
* Network Flow / Max Flow – Min Cut
  + Least Cost Network, Circuit Design, Transportation Networks
  + Ford-Fulkerson
  + Edmonds-Karp
* Bridge Detection
* Traveling Salesperson Problem

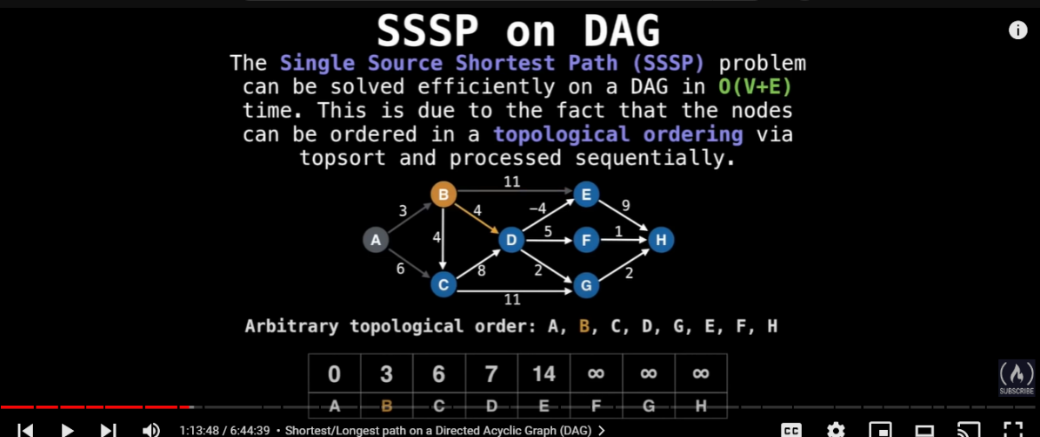
Graph Traversal Algorithms

* Depth First Search O(V + E)
  + Connected Components
  + Connectivity
  + Bridge Detection
  + Topological Sort (DAGs)
* Breadth First Search
  + Shortest Path
  + Maze Solving

## Topological Sort Algorithms

* Trees: Iteratively pick off leaf nodes.
  + Once a node has had all its leaves removed, it becomes eligible to remove.
* DAGs: Perform DFS until all nodes are visited and return the order they were visited in reverse order.
  + Pick an unvisited Node A.
  + Perform DFS on all unvisited nodes reachable from Node A.
  + On the recursive callback, add the current Node X to the topological ordering in reverse order.
  + 

## Shortest Paths Algorithms (SSSP & APSP)

* Single Source Shortest Path (DAGs)
  + Runtime Analysis: O(V + E)
  + Space Complexity: O(V)
  + Steps
    - Processes nodes by relaxing edges => Update to a better value if there’s a better path to this node.
      * Run Topological Sort
      * Use the Topological Order to come up with an order to process the nodes.
      * Process ALL edges outgoing from current node.
      * If a node that has been processed before has an edge going to it, compare the updated value with the existing value and replace if it’s better.
      * Repeat.
      * RelaxEdge() guarantees that if there is an edge to a visited node, then the edge weight will be updated, but the visited vertex will not be revisited.
      * 

1. Longest Path in a Graph (Scheduling)
   1. NP-Hard on ALL except DAGs O(V + E)
      1. Part 1: Multiply all edge weights by -1
      2. Part 2: Run DFS-SSSP
      3. Part 3: Multiple Edge weights by -1 again and returns the shortest path.
2. Dijkstra’s
   1. Runtime: O((V + E)log(V))
   2. Space Complexity: O(V)
   3. Non-negative edge weights (directed, undirected, and cyclic)
      1. Step 1: Create a distance array (initialized to infinity, starting node to 0)
      2. Step 2: Maintain a PQ of nodes ordered by shortest edge weight.
      3. Step 3: Insert Starting Node, add all edges going out of s to PQ, and relax edges in order of shortest edge weight next until we find the Target Node.
   4. Priority Queue Guarantees that if a vertex is processed, then its shortest path has already been discovered (no negative edge weights)
   5. The Priority Queue is always sorting vertexes based on their current shortest path.
3. Bellman-Ford
   1. Runtime: O(V \* E)
   2. Space Complexity: O(V)
   3. Use when Dijkstra’s fails (Only on negative edge weights)
      1. Step 1: Loop Through all nodes in the graph
      2. Step 2: Loop through all edges in the graph and update the distance of the endpoint node if the distance is improved.
      3. Step 3: Repeat V – 1 Times.
      4. Step 4: Run the algorithm AGAIN and if the distance array updates, we’ve detected a negative cycle.
4. Floyd-Warshall
   1. Runtime: O(V^3)
   2. Space Complexity: O(V^3) -> O(V^2) – compute k in place
   3. APSP -> All Pairs Shortest Paths Algorithm
   4. Best for graphs with no more than a few hundred nodes
      1. Step 1: Use adjacency matrix representation.
      2. Step 2: Create a 3D/2D dp table such that dp[k][u][v] or d[i][j] is the shortest path from u-> v using k-nodes.
      3. Step 3: When k = 0, memo[k][i][j] = m[i][j] // adjacency matrix
      4. Step 4: Check all intermediate pairs:
         1. dp[k][i][j] = Math.min(dp[k-1][i][j], dp[k-1][i][k] + dp[k - 1][k][j]); // use the best solution from nodes 0->(k-1)
         2. 2D dist(I, j) = Math.min(dist[i][j], dist[i][k] + dist[k][j]);
   5. Visualization: <https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-floyd-warshall/index_en.html>
   6. Wiki: <https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm#Example>

## Bridges and Articulation Points

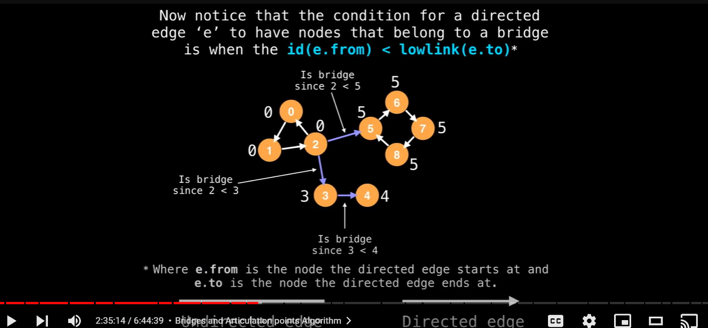
A bridge in a graph is an edge such that its removal from the graph would split the nodes into two or more disconnected components.

Similarly, an articulation vertex (commonly referred to as an articulation point) is a vertex in the graph such that its removal would split the graph into two or more disconnected components.

**Definition**

* Low Link Value: Smallest node UID reachable from the current node, including itself.

Bridges Algorithm

* Runtime: O(V (V + E)) -> O(V + E)
* Space Complexity: O(V)
* Performed on an undirected graph.
  + Algorithm
    - Start at any node and perform DFS.
    - Begin the DFS traversal by assigning each vertex a UID and LLV that is exactly the vertex’s label.
    - Perform DFS on all connected vertices.
    - Return the LLV of the node at the end of DFS.
    - During the loop, update the LLV of the current node if the traversal on the neighboring nodes produces a smaller LLV.
    - Bridge ID: EDGE(U, V), If the UID(U) < LLV(V) 🡺 U is not reachable from V, so removing the edge creates an additional disconnected component.
    - 
    - Maintain a Stack
      * Add vertices to stack
      * After the loop, if the LLV(U) == UID(U), pop all values from stack. Update the LLV of all values on stack to the LLV(U).

Articulation Points Algorithm

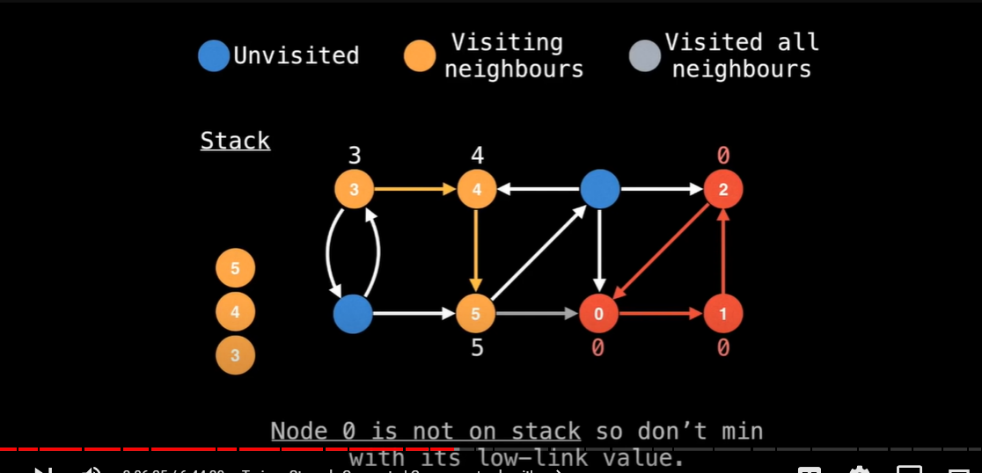
* Runtime: O(V + E)
* Space Complexity: O(V)
  + If (u, v) is a bridge, then u or v is an Articulation Point. But this only captures **SOME** articulation points. It does not capture cycles.
  + Cycles
    - If Low\_Link(v) == UID(v) -> We’ve discovered a cycle. So V is an articulation point.
      * This fails if the starting node has 0 or 1 outgoing and/or incoming edges IN THE UNDIRECTED VERSION ON THE GRAPH!
      * A screenshot of a computer game

        Description automatically generated

## Strongly Connected Components (SCC)

**Strongly Connected Components** are defined as a group of vertices such that for any vertex V in the SCC, there is a path to all other vertices in the SCC.

**Definitions:**

* Low Link Value:
  + The LLV of vertex U is the smallest ID of the vertex V reachable from U when running DFS.
* Tarjan’s Algorithm
  + Runtime: O(V + E)
  + Space Complexity: O(V)
  + Steps:
    - Maintain a stack of nodes to prevent Strongly Connected Components from interfering with other Low Link Values.
    - Run DFS on any node. Update ID, LLV, and add the vertex to the stack.
    - Add nodes to a stack as they are explored for the first time.
    - Remove nodes from the stack when a strongly connected component is found.
    - If the current node is the start of the SCC (i.e. ID == LLV), update all nodes on the stack with it’s LLV / ID.
  + Low Link Value Update Condition:
    - Case 1: LLV(v) < LLV(u)
      * There must be an edge going from u -> v AND v must already be on the Stack.
    - Case 2: The recursive call on a neighbor produced a smaller LLV.
    - 

## Traveling Salesperson Problem

**The Traveling Salesperson Problem is defined as follows:**

Given a set of cities and the distances between them, find the shortest possible route that visits each city exactly once and returns to the starting point.

* The Traveling Salesperson Problem is reduced to the problem of finding a Hamiltonian Cycle in a Weighted Complete Graph.
* A Complete Graph:
  + A graph where ALL vertices have an edge to every other vertex in the graph.
* A Hamiltonian Cycle:
  + A cycle that visits every node once and then returns to the starting vertex, never visiting the same vertex again until the very end.
* Runtime Analysis:
  + Dynamic Programming takes the naïve approach from O(n!) to O(n^2\*2^n) running time.

**Why Greedy and MST Algorithms Don’t Solve the Problem:**

While both Minimum Spanning Trees (MST) and Greedy algorithms seem like good candidates for solving TSP, they fall short for a key reason: They prioritize short edges, which doesn't guarantee the shortest overall tour.

* MSTs:
  + An MST finds the edges with the minimum weights that connect all nodes exactly once, forming a tree structure.
  + The problem is that an MST doesn't necessarily create a closed loop visiting all cities efficiently. It might prioritize short connections that don't lead to an optimal overall route.
* Greedy Algorithms:
  + Greedy algorithms make the locally optimal choice at each step, hoping it leads to the global optimum (shortest tour).
  + In TSP, a greedy approach might prioritize visiting the closest unvisited city at each step. However, this can lead to suboptimal solutions where backtracking becomes necessary, potentially creating longer overall distances.

**The Held-Karp Algorithm**

The Held-Karp Algorithm provides a dynamic programming solution to the Traveling Salesperson Problem.

**Pseudocode**

*def tsp(graph, start\_city):*

*"""*

*Finds the minimum cost TSP tour using dynamic programming*

*Args:*

*graph: A 2D matrix representing distances between cities*

*start\_city: The index of the starting city*

*Returns:*

*A tuple (min\_cost, optimal\_tour)*

*"""*

*num\_cities = len(graph)*

*cache = {} # Key = (subset of visited cities as bitmask, current city)*

*# Value = (optimal cost to reach this state, previous city)*

*# Base case: all cities visited except the starting city*

*for city in range(num\_cities):*

*if city != start\_city:*

*cache[((1 << city) | (1 << start\_city), city)] = (graph[start\_city][city], start\_city)*

*# Recursive case*

*for subset\_size in range(3, num\_cities + 1):*

*for subset in combinations(range(num\_cities), subset\_size):*

*# Ensure the 'subset' contains the starting city*

*if not (subset & (1 << start\_city)):*

*continue*

*for current\_city in subset:*

*if current\_city == start\_city:*

*continue*

*subset\_without\_current = subset ^ (1 << current\_city) # Bitmask removal*

*min\_cost = float('inf')*

*for prev\_city in subset\_without\_current:*

*new\_cost = cache[(subset\_without\_current, prev\_city)][0] + graph[prev\_city][current\_city]*

*if new\_cost < min\_cost:*

*min\_cost = new\_cost*

*min\_prev\_city = prev\_city*

*cache[(subset, current\_city)] = (min\_cost, min\_prev\_city)*

*# Find optimal tour and its cost*

*min\_tour\_cost = float('inf')*

*last\_city = start\_city # Starting point is the last city in a tour*

*optimal\_tour = []*

*for city in range(num\_cities):*

*if city != start\_city:*

*new\_cost = cache[((1 << num\_cities) - 1, city)][0] + graph[city][last\_city]*

*if new\_cost < min\_tour\_cost:*

*min\_tour\_cost = new\_cost*

*last\_city = city*

*optimal\_tour.append(last\_city)*

*subset = (1 << num\_cities) - 1 # Visited all cities*

*while True:*

*prev\_city = cache[(subset, last\_city)][1]*

*optimal\_tour.append(prev\_city)*

*subset = subset ^ (1 << last\_city)*

*last\_city = prev\_city*

*if last\_city == start\_city:*

*break*

*optimal\_tour.reverse() # Correct the order*

*return min\_tour\_cost, optimal\_tour*

**Runtime Analysis**

* Time Complexity: O(n^2 \* 2^n)
  + 'n' is the number of cities. There are roughly 2^n different subsets and for each subset, we iterate over 'n' cities.
* Space Complexity: O(n \* 2^n)
  + Comes from storing the 'cache' of partial solutions.

## Eulerian Paths and Circuits

An **Eulerian Path** is a path in a graph such that **every edge is crossed exactly once**. The existence of this path depends on the starting node and the degree of all vertices within the graph.

Alternatively, an **Eulerian Circuit is a type of Eulerian Path** that **starts and ends at the same vertex**. They share the **same goal to cross every edge exactly once**.

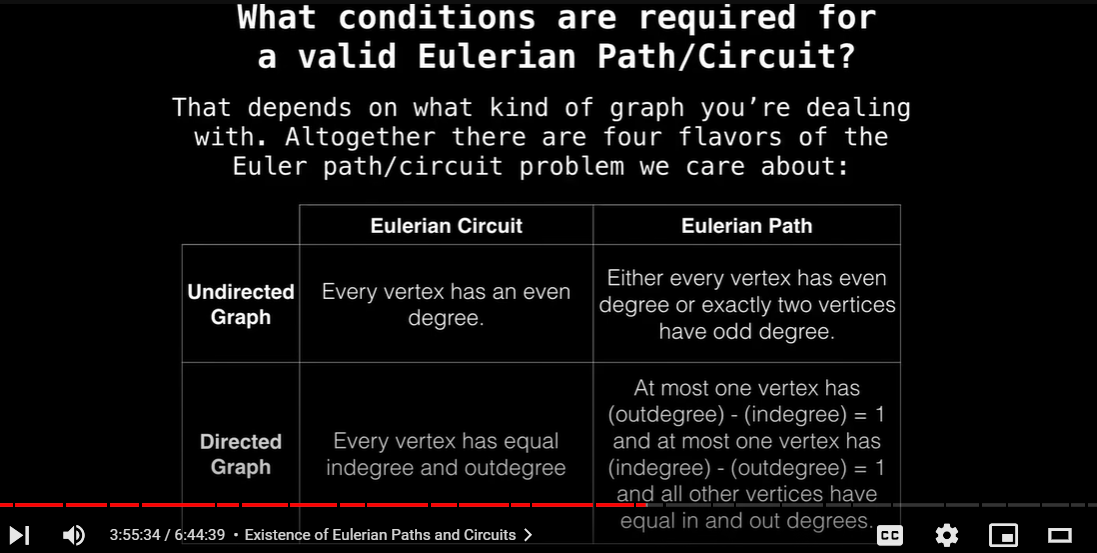
**Notes**

* Not every graph has an Eulerian Path.
* In an Eulerian Path, it is not necessary to return to the start vertex and vertices may be visited more than once.

Below is a chart that defines the set of distinct rules for directed and undirected graphs to determine if an Eulerian Path or an Eulerian Circuit is possible.

Keep in mind that IF an Eulerian Circuit exists 🡺 Eulerian Path exists. **BUT the same is not always true going the other way.**

Also note that **vertices with 0 degrees are vacuously true.**



* Undirected Graph
  + Eulerian Path: Either every vertex has even ways OUT and even ways (EVEN DEGREE) OR exactly two vertices have odd ways IN and OUT.
  + Eulerian Circuit: Every vertex has an even number of ways IN and OUT (EVEN DEGREE).
* Directed Graph
  + Eulerian Path: At most one vertex has 1 more OUT edge than IN edges AND at most one vertex has 1 more IN edge than OUT edges.
  + Eulerian Circuit: Every vertex has EQUAL IN and OUT degrees (ODD or EVEN).

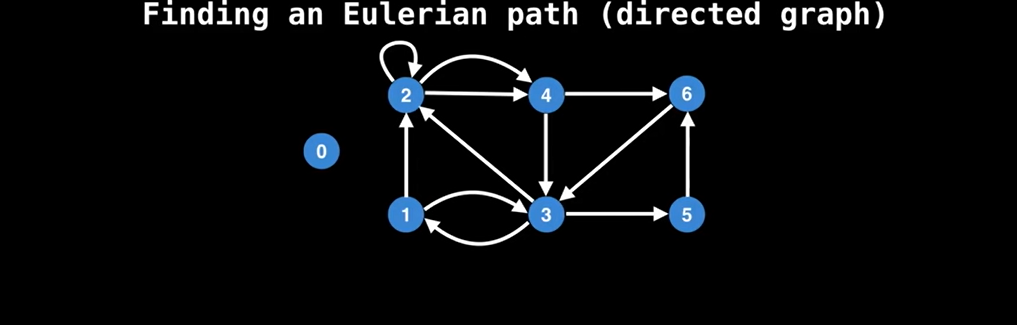
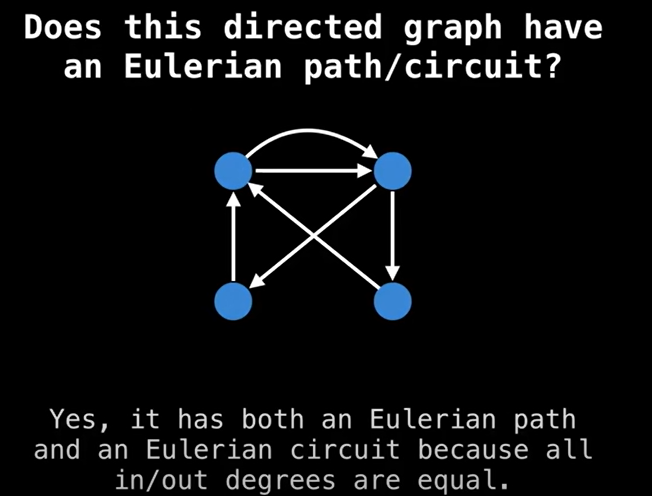
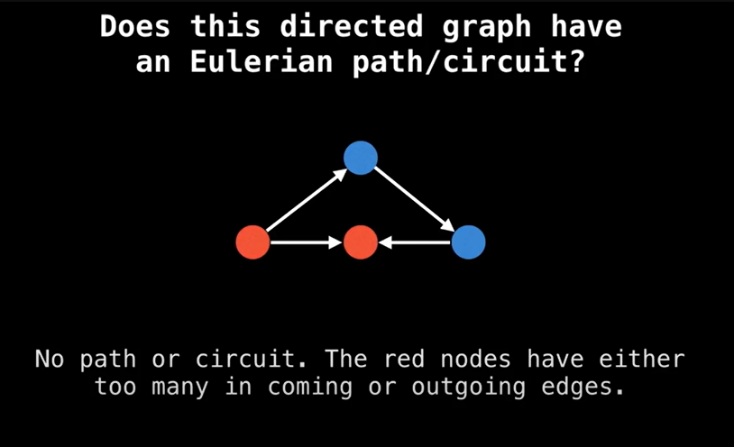
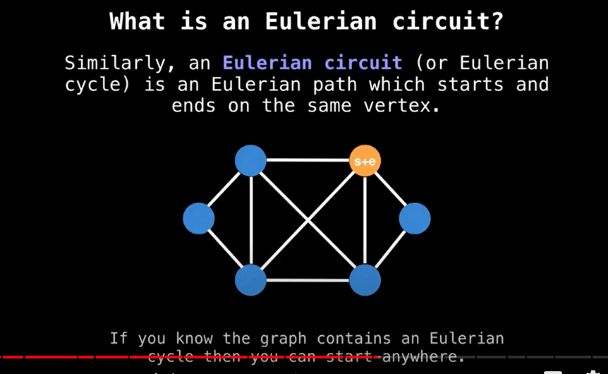
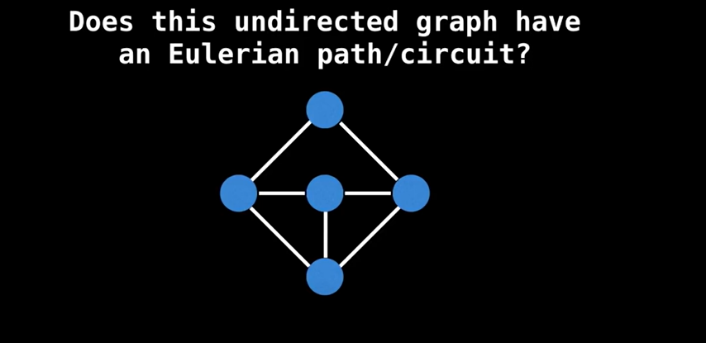
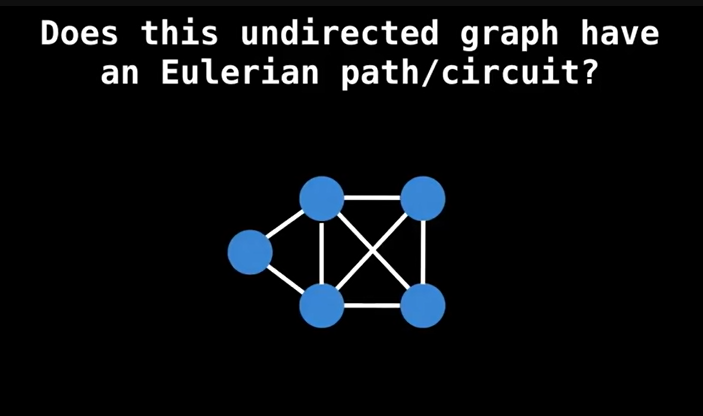
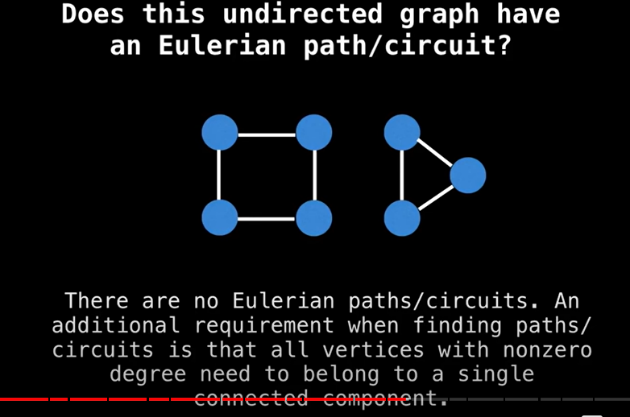
**Eulerian Path Algorithm O(E)**

* 1. Step 1: Check all the constraints in the chart above for a valid path.
     1. Undirected -> Every vertex has EVEN degree, Directed -> All vertices have EQUAL IN and OUT degrees OR 1 node has an additional OUT degree, and 1 node has an additional IN degree.
  2. Step 2: Find a starting node (Node with an additional outgoing edge) and end node (Node with an additional ingoing edge). Otherwise, each vertex has even IN and OUT degree, so start anywhere.
  3. Step 3: Run a modified DFS.
     1. If we finish traversing all the outgoing edges from the current node, add the node we’re to the front of the path, mark as visited, and continue.

**Eulerian Circuit Algorithm O(E)**

* 1. Step 1: Check the constraints in the above chart for a valid circuit.
     1. Undirected -> All EVEN degrees, Directed -> Equal In and OUT degrees PER NODE.
  2. Step 2: If the circuit exists, the algorithm can start anywhere.

**Practice Graphs**

* 
* 
* 
* 
* 
* 
* 

## Minimum Spanning Trees (MST)

**Note:** Log(V) == Log(E)

**Prim’s Algorithm**

* Runtime Analysis: O(E \* log(V)) -> O(V^2)
  + Log(V): Time it takes to extract the minimum edge from the priority queue and restructure the heap. Each vertex in V has at most one edge (PQ should not contain duplicate edges) in the priority queue at a time.
    - There are vertices in the PQ, NOT EDGES!
  + V: Time it takes to search for the next smallest edge in an adjacency matrix. Must be repeated V times.
  + E: Updating the priority queue with a new edge, at most E for every vertex.
* Space Complexity: O(E)
* Data Structure: Priority Queue<Edge>
* Steps
  + Sort edges in a PQ based on the weight of the edges.
  + Start on any Node S, mark S as visited, and add all its edges to the PQ.
  + While the PQ is not empty AND the MST is not complete, traverse the next edge of least cost.
  + If the vertex that the edge is going to is already in the MST, Don’t add it to the PQ! Instead, compare it to the current edge that should be stored in the IPQ and replace it if the edge is smaller.
  + Otherwise, mark the node as visited and select the edge.
  + Repeat until the MST is complete.
  + Size of the MST: (V – 1) Edges

**Kruskal’s Algorithm**

* Runtime Analysis: O(E\*Log(E)) OR O(E\*Log(V))
  + E\*Log(E): Sorting the edges with an efficient sorting algorithm.
  + Log(V): Merging connected components in a Disjoint Set data structure that utilizes path compression.
* Space Complexity: O(E)
* Data Structure: Disjoint Sets
  + Disjoint Sets can be used to efficiently check for connected components.
* Steps
  + Kruskal’s Algorithm sorts the edges by weight and continues to add the edges to the graph if they connect an additional node that was not apart of the expanding MST so far.
  + In contrast to Prim’s Algorithm, Kruskal’s does not process edges based on a starting vertex.

**When to Use Kruskal’s or Prim’s**

* Prim's Algorithm:
  + This algorithm generally performs better for dense graphs (more edges than vertices) due to the efficient use of priority queues.
* Kruskal's Algorithm:
  + This algorithm is typically preferred for sparse graphs (fewer edges than vertices) as it avoids unnecessary processing of irrelevant edges and relies on simpler data structures.

## Max Flow Problem

**The Max Flow Problem is defined as the following:**

Given an infinite input source, how much flow can we push through the network given that each edge has a certain capacity?

**Ford-Fulkerson Max Flow Algorithm**

* Initialization: Start with zero flow in the network.
* Find an Augmenting Path: Search for a path from the source to the sink in the "residual graph" (a graph representing remaining capacities). Common methods include breadth-first search (BFS) or depth-first search (DFS).
* Augment Flow: If an augmenting path is found, increase the flow along that path by the minimum capacity of any edge within the path.
* Repeat: Continue finding augmenting paths and augmenting the flow until no more augmenting paths can be found.

### Ford-Fulkerson Intuition

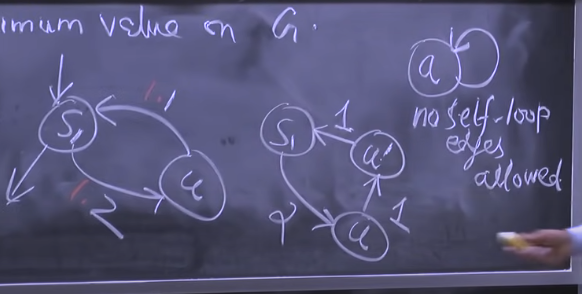
* If no path exists in the residual graph Gf, then the flow is at its maximum flow.
* If there is a path, this is not the maximum flow.
  + Return the edges as edges that need to be updated to get to the maximum flow.

### Max Flow Examples

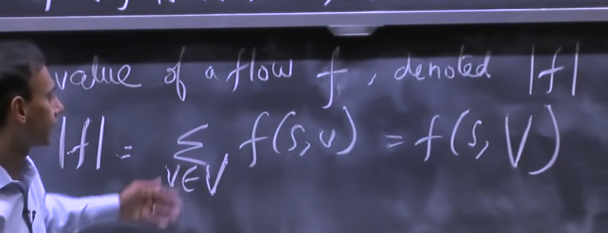
A blackboard with white chalk drawn on it

Description automatically generated

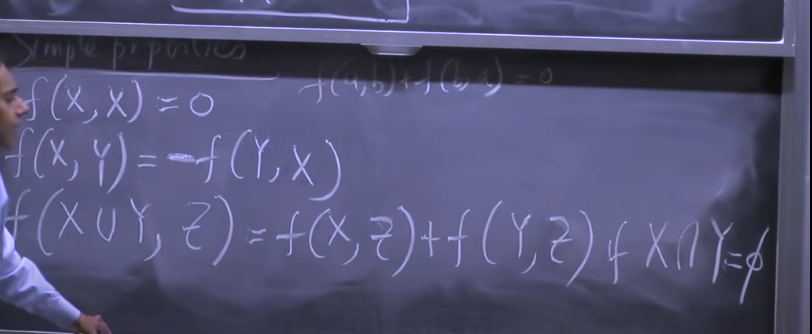
**Assumptions**

* We can disallow simple cycles.
  + Self-Loops
  + Loop from u->v and v->u
  + No cycles of length 1 or 2.
    - Linear Expansion: If cycles of length 1 or 2 exist, an intermediate node can be introduced with the same flow.
    - 
* The flow of an edge cannot exceed the capacity of the edge.
* Vertices are not allowed to accumulate flow over time.

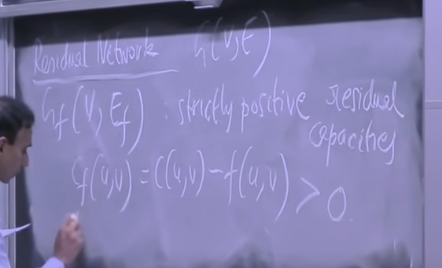
**Constraints**

* There always exists a Source Node and a Sink Node.
  + Source 🡺 No incoming edges.
  + Sink 🡺 No outgoing edges.
* The flow of an edge cannot exceed its capacity.
* Vertices cannot accumulate flow over time.
* Define F(U, V) to be the Max Flow from Node U to Node V.
  + Skew Symmetry: F(U, V) == -F(V, U)
  + 

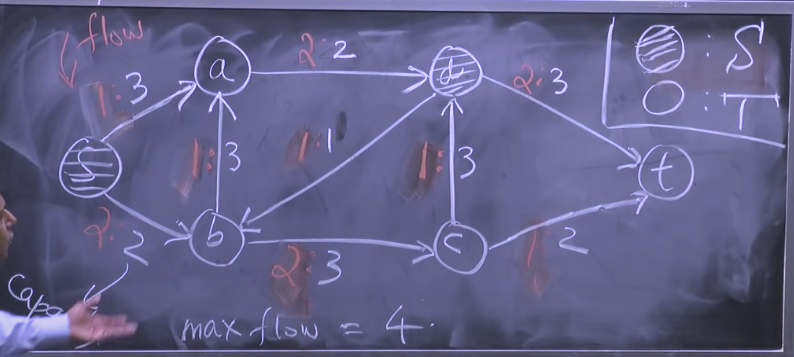
**Definitions**

* F(X, Y) = -F(Y, X)
* F(X, X) = 0
* F(X + Y, Z) = F(X, Z) + F(Y, Z) IFF X + Y == 0 / Null Set
* 

### Residual Network



**Original Graph**



**The Residual Graph**

