University of Pittsburgh

Algorithms and Data Structures 2
CS 1501

Pitt SCI 5 Years

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Lab 6: Monday 10/31 @ 11:59 pm
  - **Assignment 2: ~~Friday 11/4~~ Monday 11/7 @ 11:59 pm**
  - Lab 7: next Monday 11/7 @ 11:59 pm
  - Homework 8: next Friday @ 11:59 pm
- Live Support Session for Assignment 2
  - Recording and slides on the assignment Canvas page
- Weekly Live QA Session on Piazza
  - Friday 4:30-5:30 pm

Previous lecture

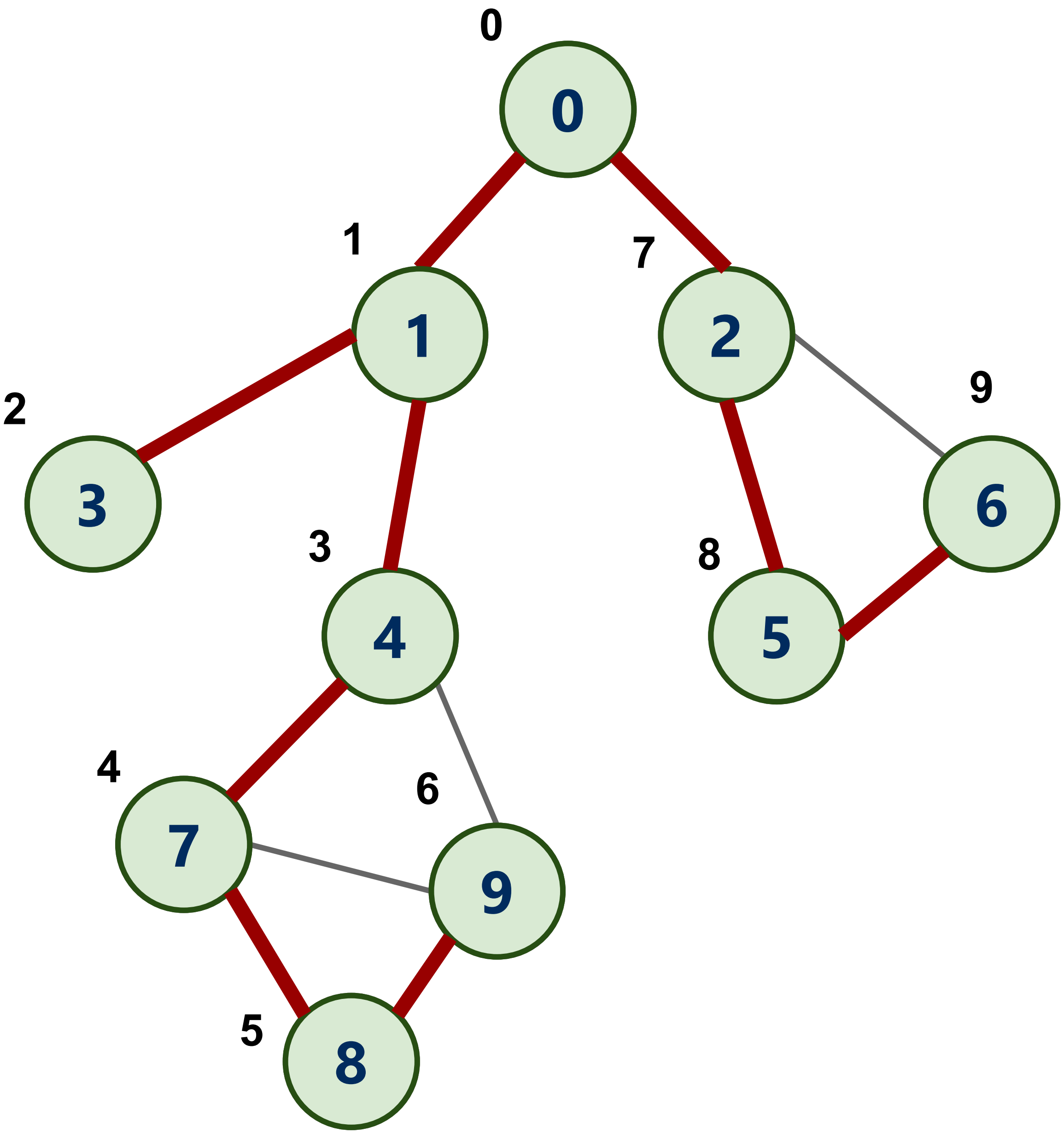
- ADT Graph
  - traversals
    - DFS
      - finding articulation points of a graph

This Lecture

- ADT Graph
  - finding articulation points of a graph
  - Graph compression
  - Graphs with weighted edges
  - Minimum Spanning Tree (MST) problem
    - Prim’s MST algorithm

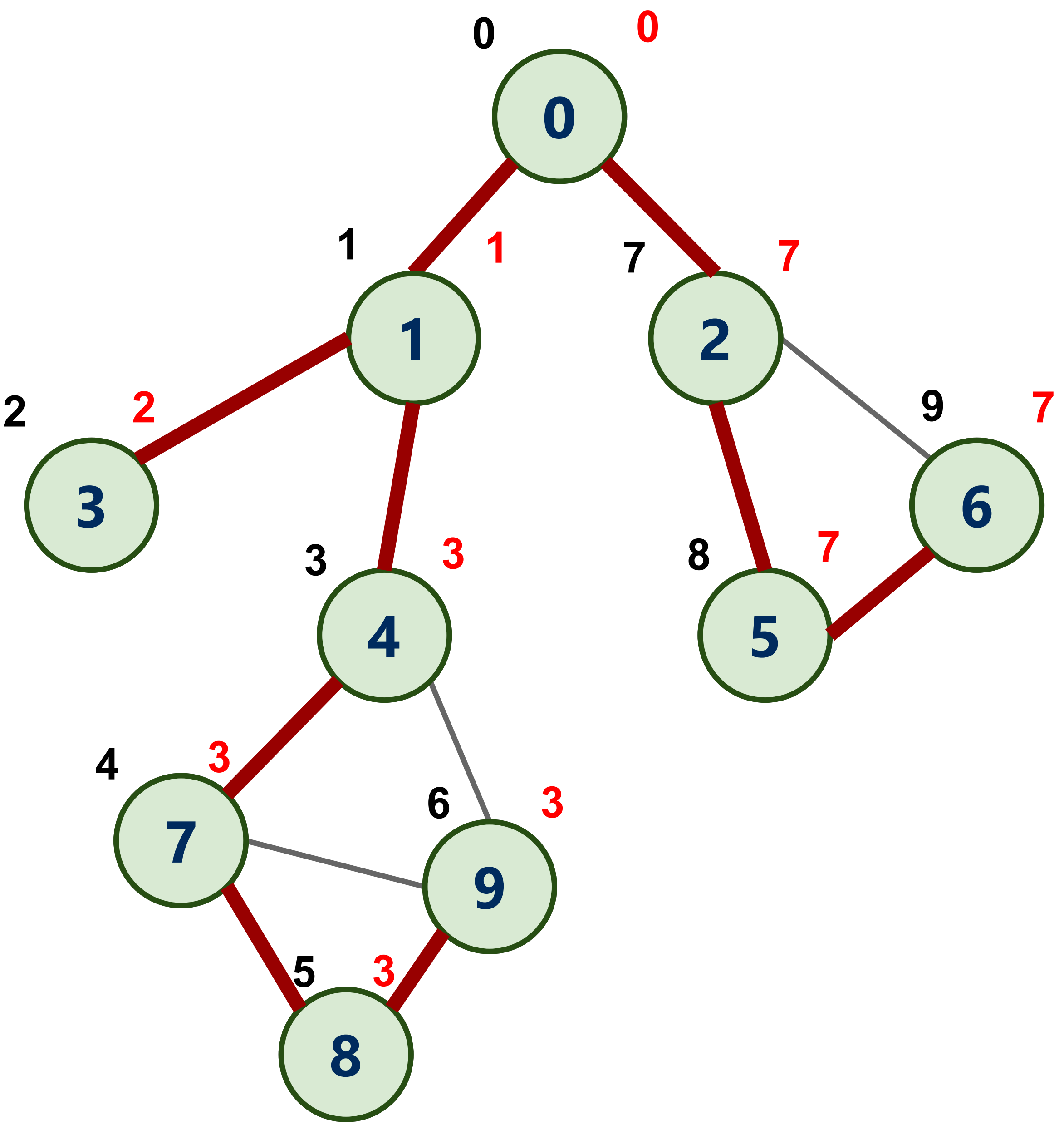
num(v)

- A pre-order DFS traversal visits the vertices in some order
  - let's number the vertices with their traversal order
  - num(v)



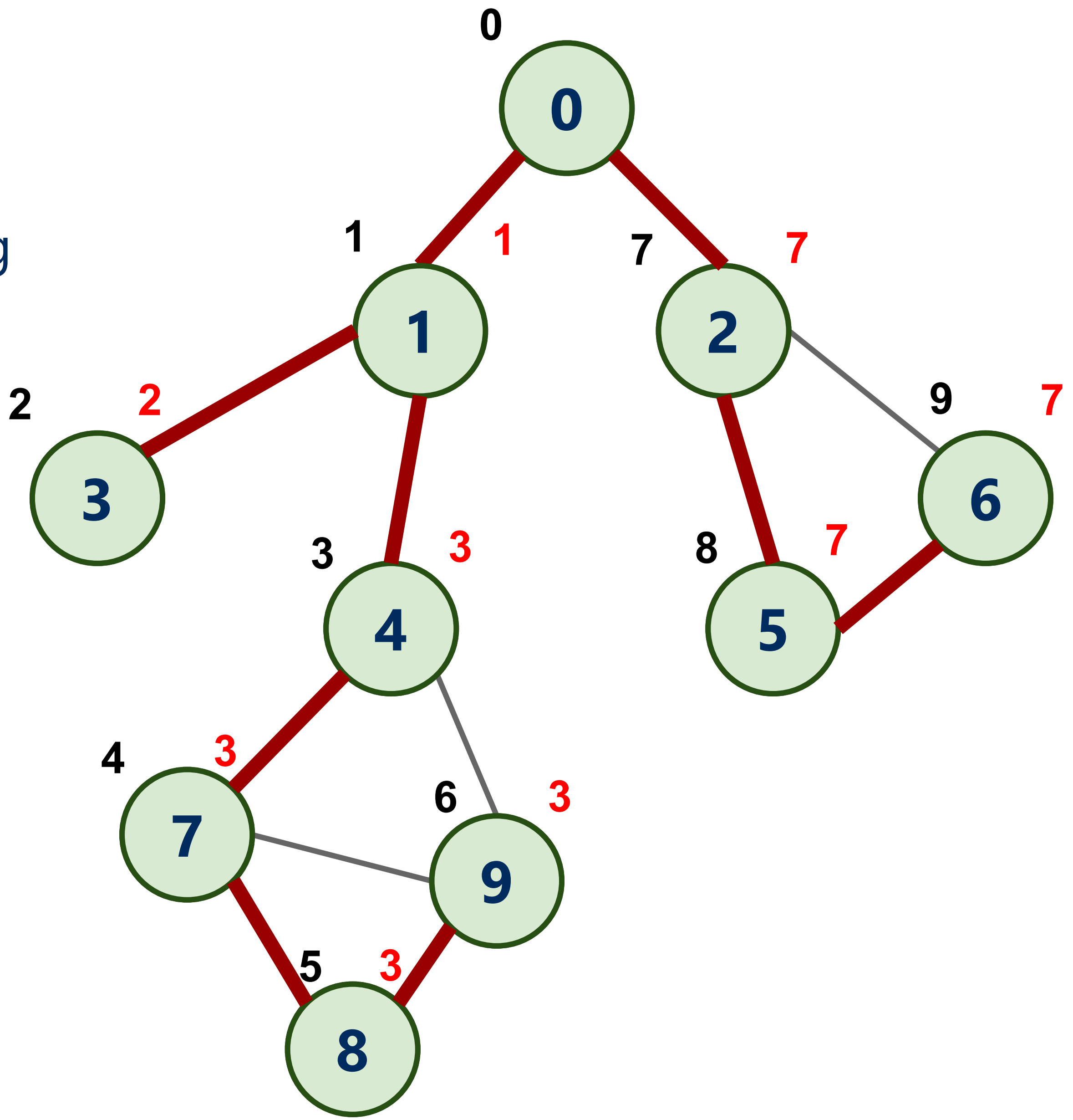
low(v)

- For each vertex v, find the lowest numbered vertex reachable from v through **vertices and edges that have not been traversed yet**
  - the number of such vertex is low(v)



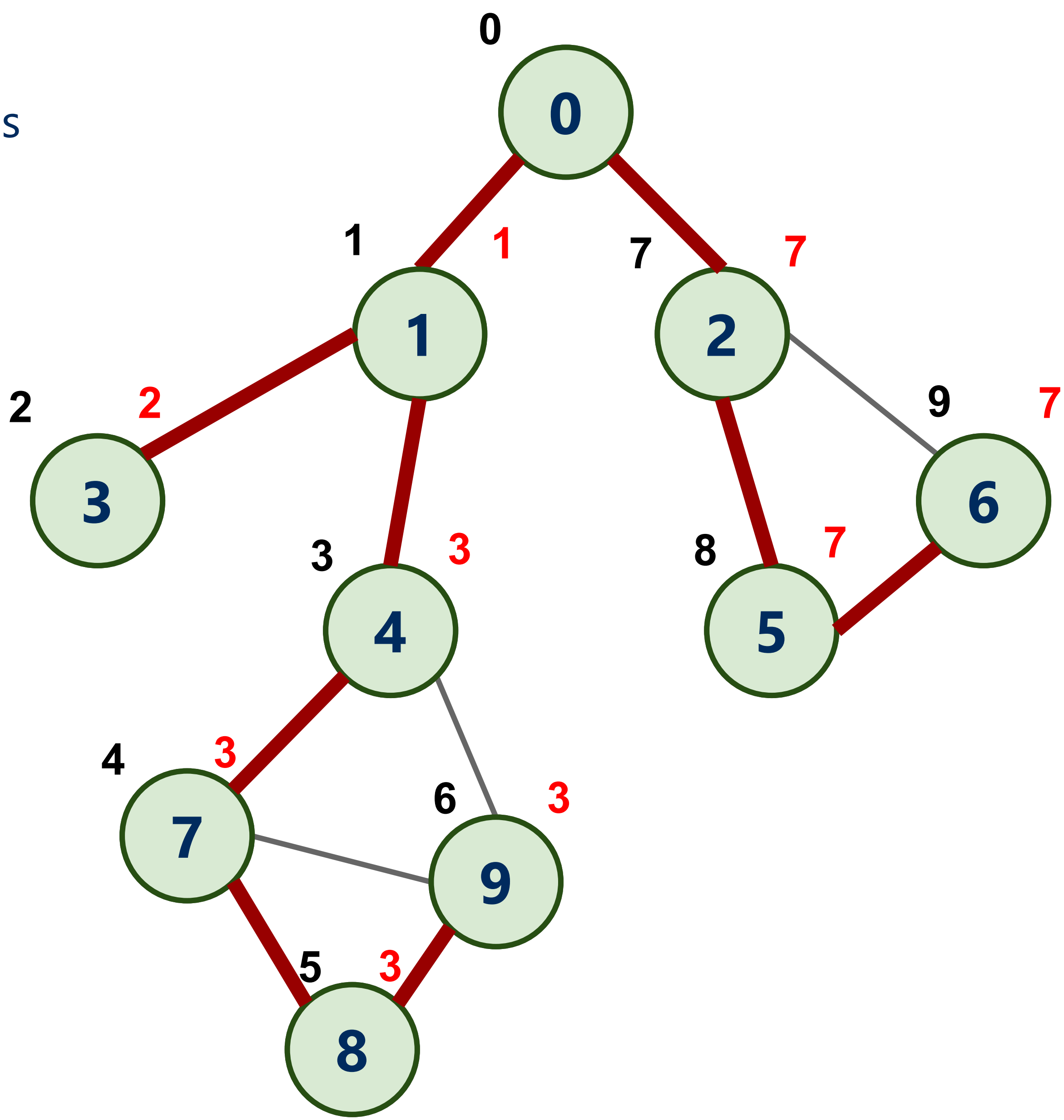
low(v)

- How did we find low(v)?
- Move down the tree looking for a back edge that goes backwards "the furthest"
- **0 or more tree edges going down**
- **then at most one back edge**
    - **why at most one?**



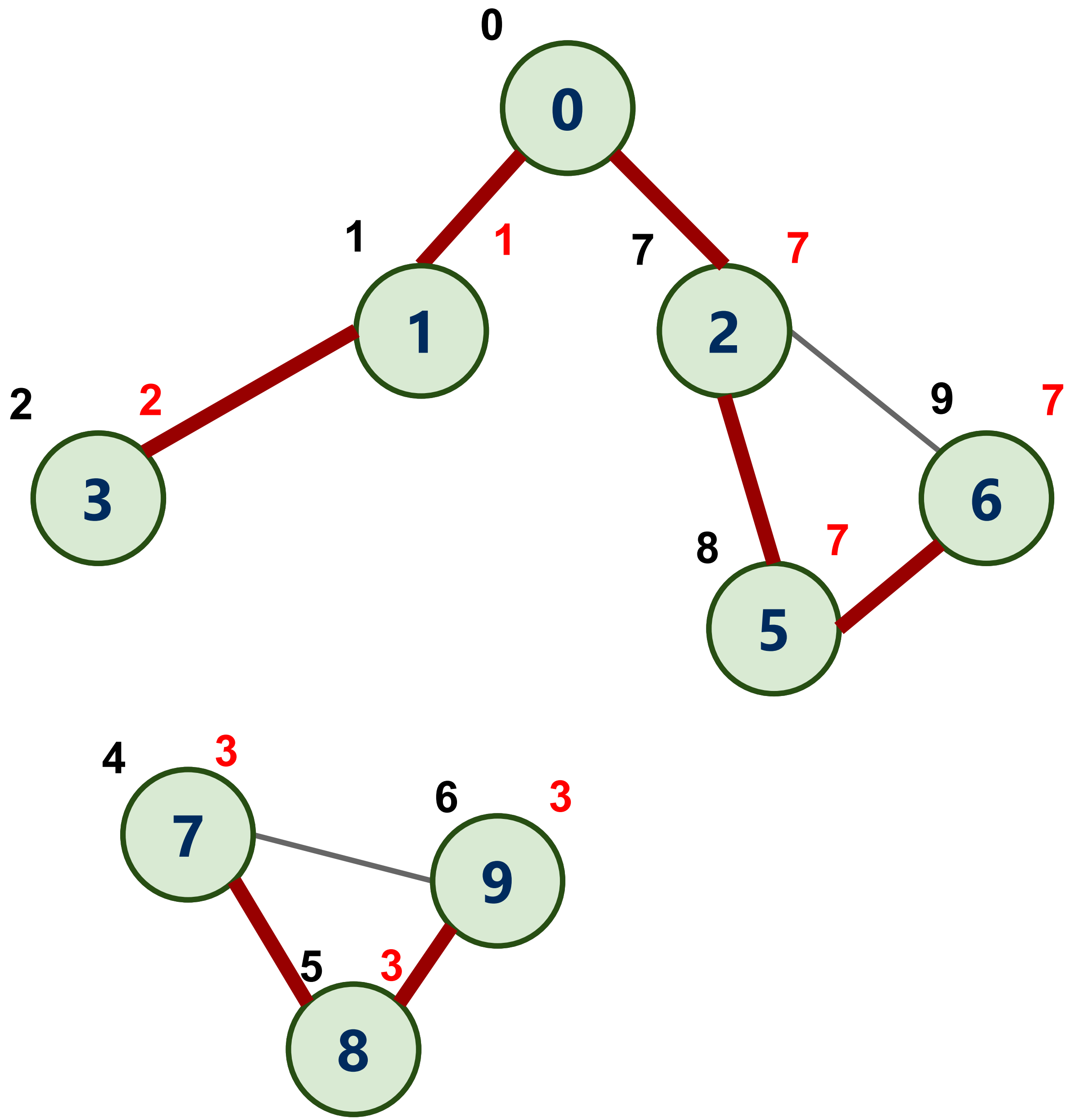
Why are we computing low(v)?

- What does it mean if a vertex has a child such that
  - low(child) >= num(parent)?
- e.g., 4 and 7
- child has no other way except through parent to reach vertices with lower num values than parent
- e.g., 7 cannot reach 0, 1, and 3 except through 4
- So, the parent is an articulation point!
  - e.g., if 4 is removed, the graph becomes disconnected



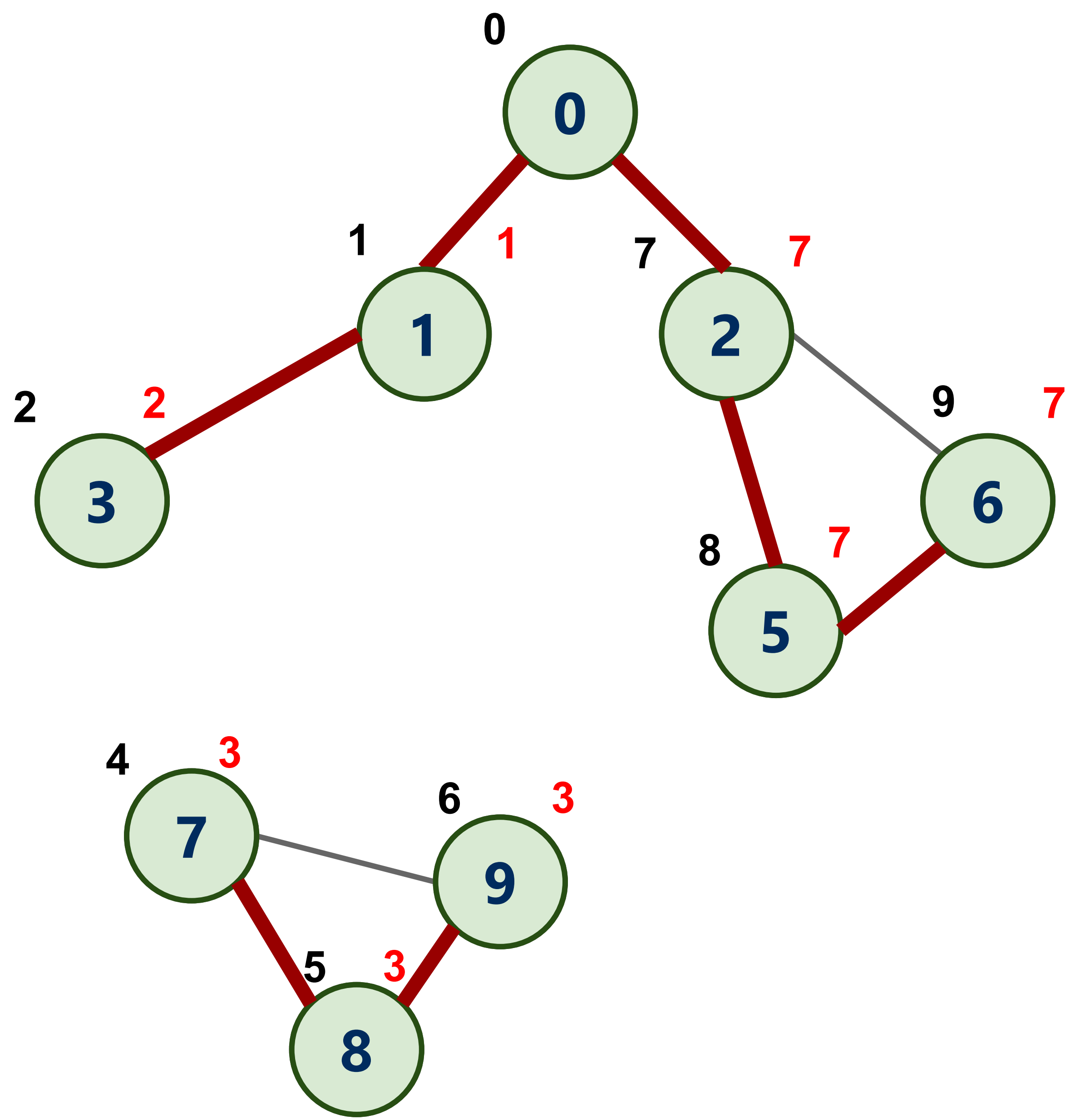
Why are we computing low(v)?

- if 4 is removed, the graph becomes disconnected
- Each non-root vertex v that has a child w such that low(w) >= num(v) is an articulation point



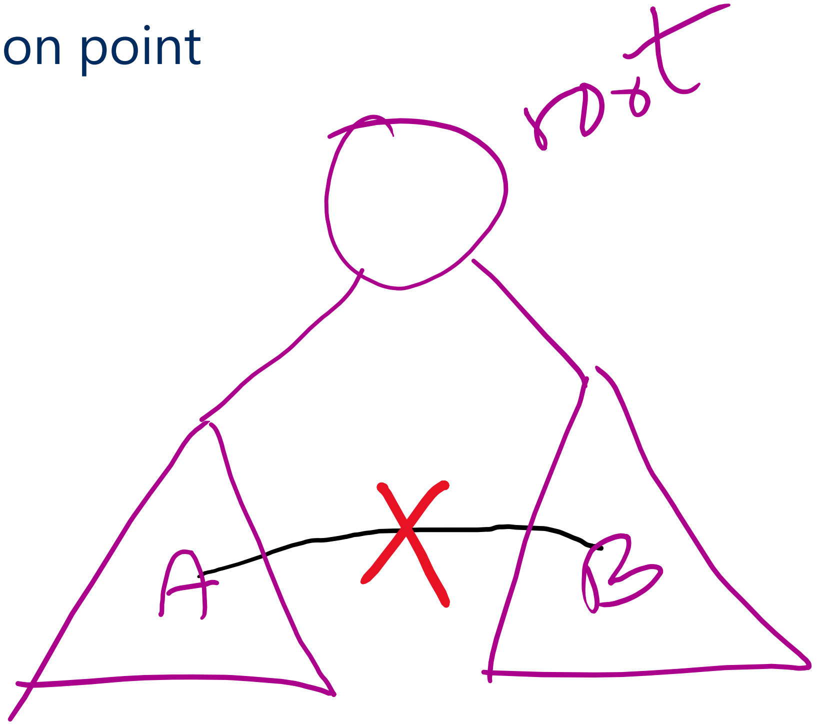
What about the root vertex?

- The root has the smallest num value
  - root's children can't go "further" than root
- Possible that low(child) == num(root) but root is not an articulation point
- need a different condition for root



What about the root of the spanning tree?

- What if we start DFS at an articulation point?
  - The starting vertex becomes the root of the spanning tree
  - If the root of the spanning tree has more than one child, the root is an articulation point



low(v)

- low(v) = the num value of the lowest-numbered vertex reachable from v using 0 or more spanning tree edges and then **at most one** back edge
  - Min of:
    - num(v) (the vertex is reachable from itself)
    - num(w) for all back edges (v, w)
    - low(w) of all children of *v* (the lowest-numbered vertex reachable through a child)

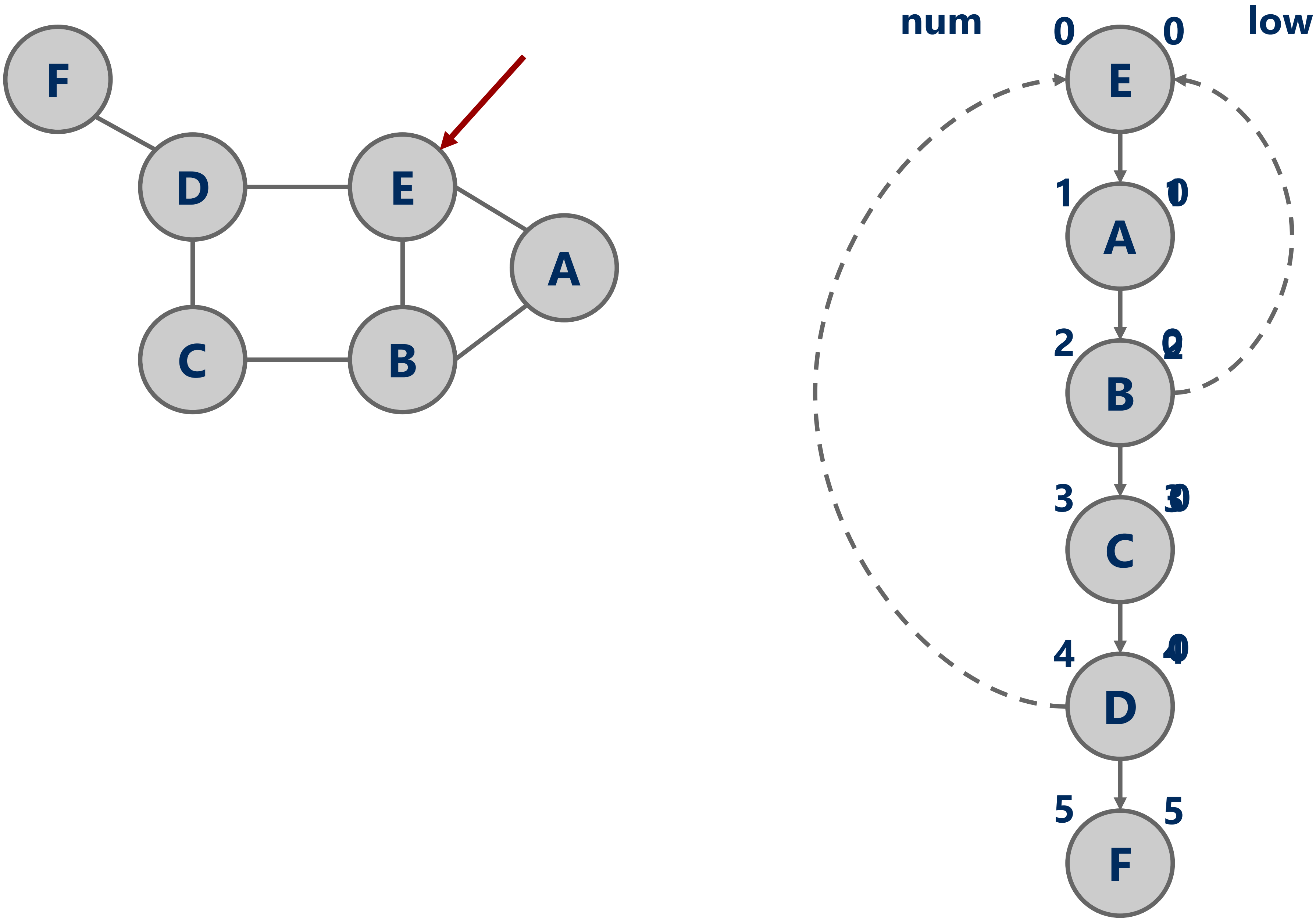
Finding articulation points of a graph: The Algorithm

- As DFS visits each vertex v
  - Label v with with the two numbers:
    - num(v)
    - low(v): initial value is num(v)
  - For each neighbor w
    - if already seen → we have a back edge
      - update low(v) to num(w) if num(w) is less
    - if not seen → we have a child
      - call DFS on the child
      - **after the call returns,**
        - update low(v) to low(w) if low(w) is less

when to compute num(v) and low(v)

- num(v) is computed as we move down the tree
  - pre-order DFS
- low(v) is updated as we move down and up the tree
- Recursive DFS is convenient to compute both
  - why?

Finding articulation points example



Using DFS to find the articulation points of a connected undirected graph

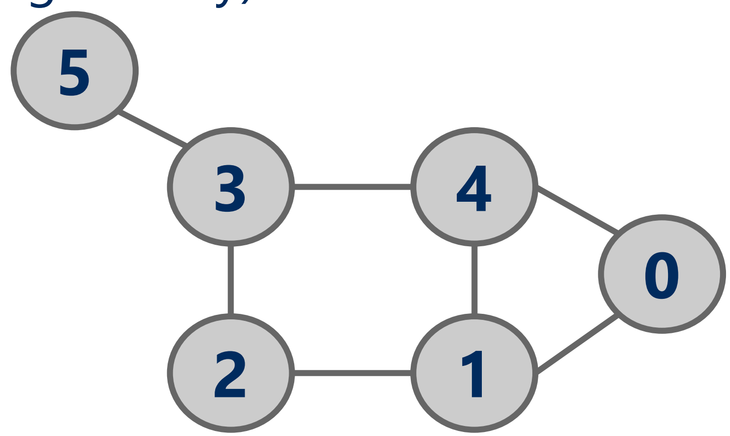
```
int num = 0

DFS(vertex v) {
    num[v] = num++
    low[v] = num[v] //initially
    seen[v] = true //mark v as seen
    for each neighbor w
       if(w unseen){
          parent[w] = v
          DFS(w) //after the call returns low[w] is computed, why?
          low[v] = min(low[v], low[w])
          if(low[w] >= num[v]) v is an articulation point
       } else { //seen neighbor
          if(w != parent[v]) //and not the parent, so back edge
            low[v] = min(low[v], num[w])
       }
```

Graph Compression

- Real-life graphs are huge
  - 100’s if not 1000’s of GBs
  - Facebook graph, Google graph, maps, ...
- Let’s see one (partial) idea for reducing the size of large graphs

Graph Compression

- **Step 1:** Construct a Compressed Sparse Row (CSR) representation of the graph
- CSR
  - Edges array concatenates ***sorted*** neighbor lists of all vertices
  - Offsets array:
    - offsets[v] is the starting index (in the Edges array) for the neighbors of vertex v



Edges

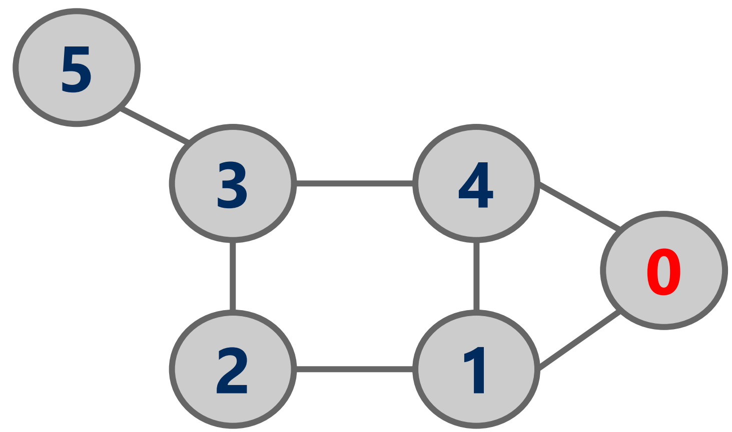
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- Let's start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
  - offsets[v] is the starting index (in the Edges array) for the neighbors of vertex v



Edges

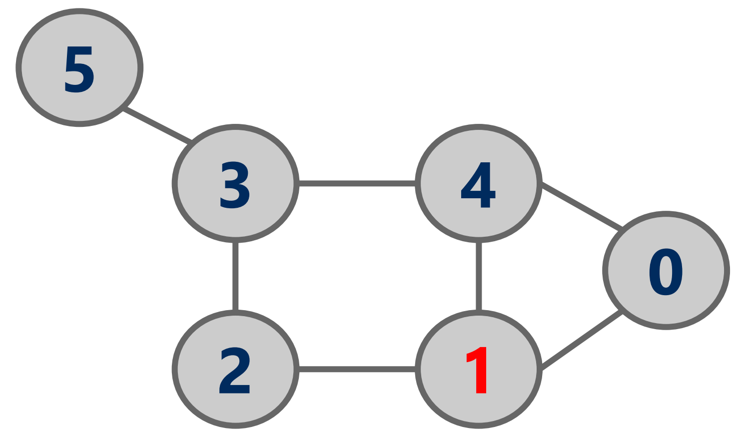
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- Let's start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
  - offsets[v] is the starting index (in the Edges array) for the neighbors of vertex v



Edges

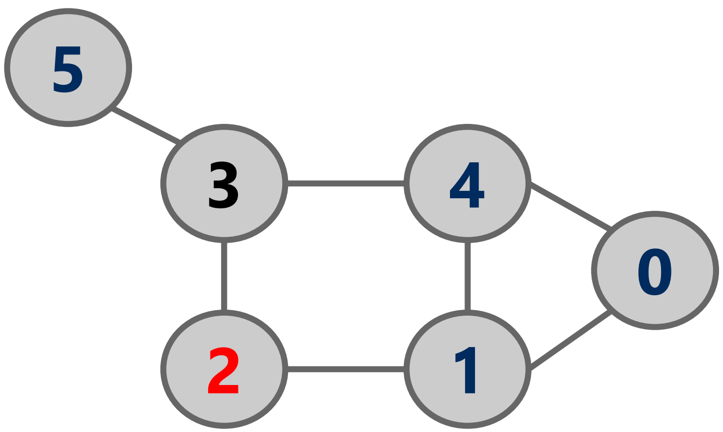
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Edges	1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- Let's start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
    - offsets[v] is the starting index (in the Edges array) for the neighbors of vertex v



Edges

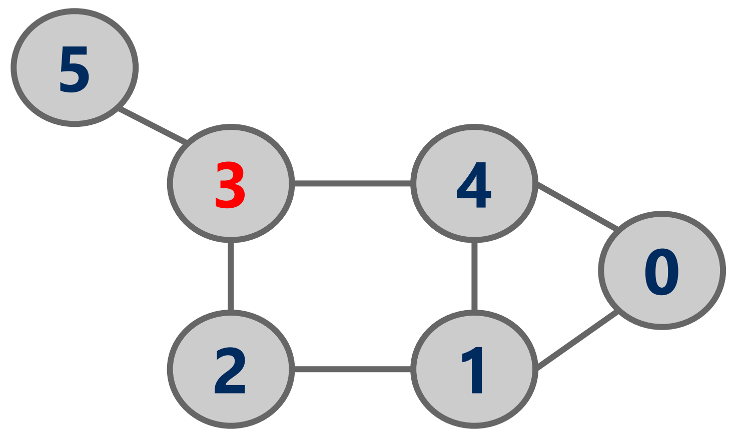
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Edges	1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- Let's start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
  - offsets[v] is the starting index (in the Edges array) for the neighbors of vertex v



Edges

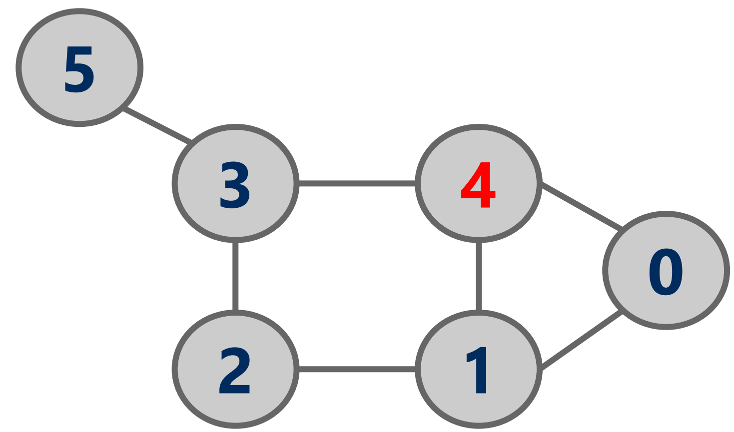
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- Let's start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
  - offsets[v] is the starting index (in the Edges array) for the neighbors of vertex v



Edges

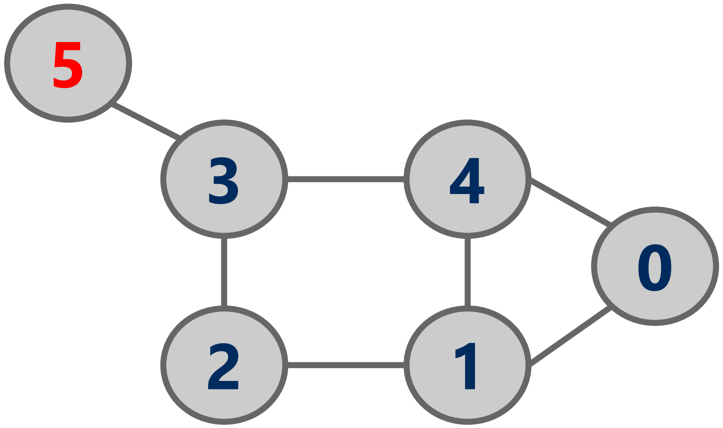
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- Let's start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
    - offsets[v] is the starting index (in the Edges array) for the neighbors of vertex v



Edges

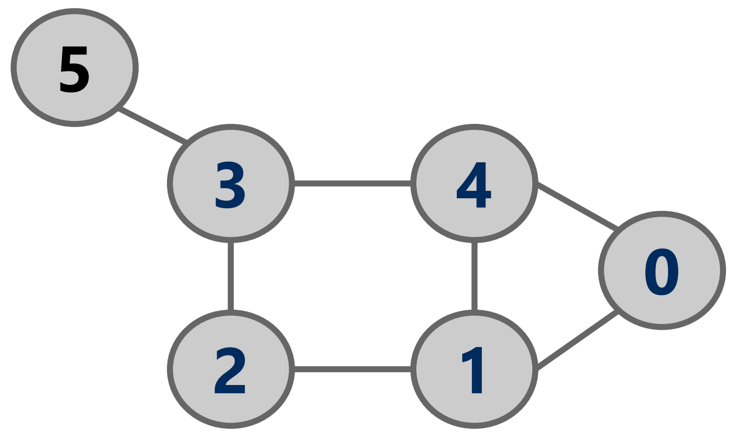
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- Can we compute the degree of a vertex using the offsets array?
  - Running time?
- What is the required space of this representation?
  - Theta(v + e)
  - Assume 4 bytes per vertex and per edge
  - Total size: *4*v* + *8*e* bytes



Edges

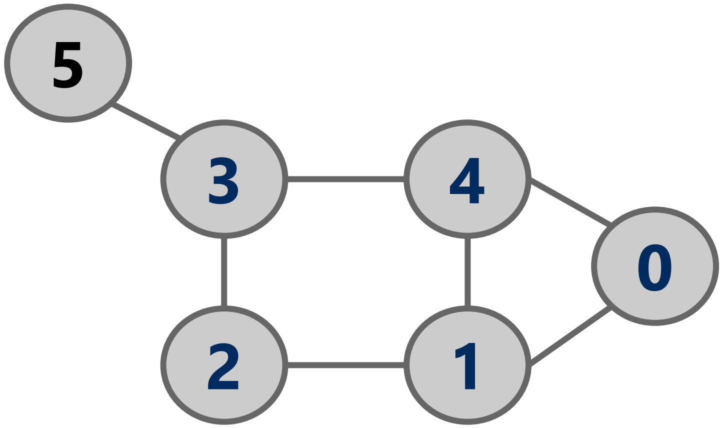
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list v_1, v_2, v_3, …
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \ldots$



Edges

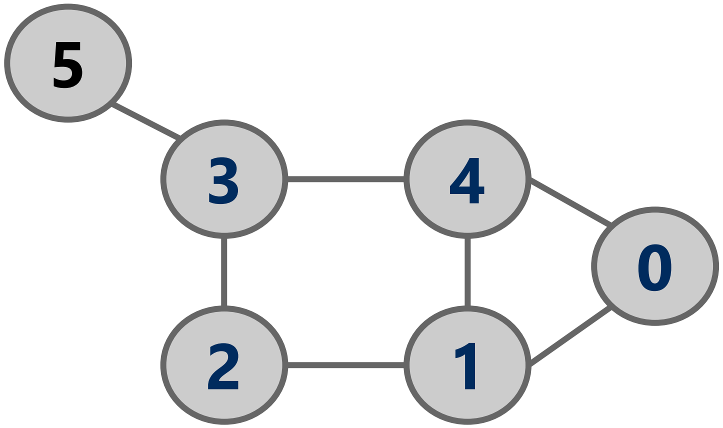
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list v_1, v_2, v_3, ...
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \ldots$



Edges

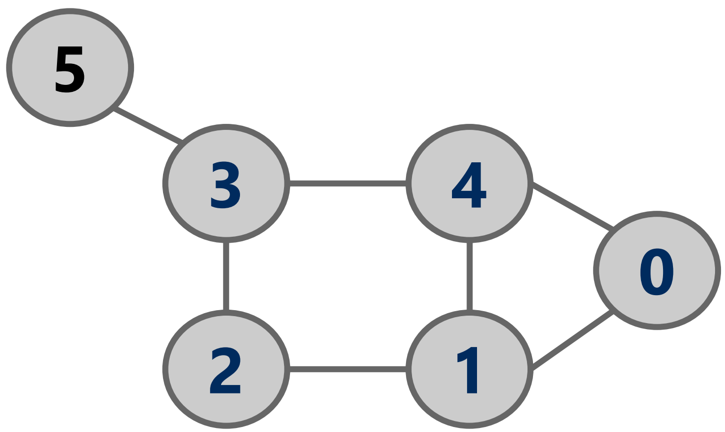
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3
1-0	4-1												

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list v_1, v_2, v_3, ...
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \ldots$



Edges

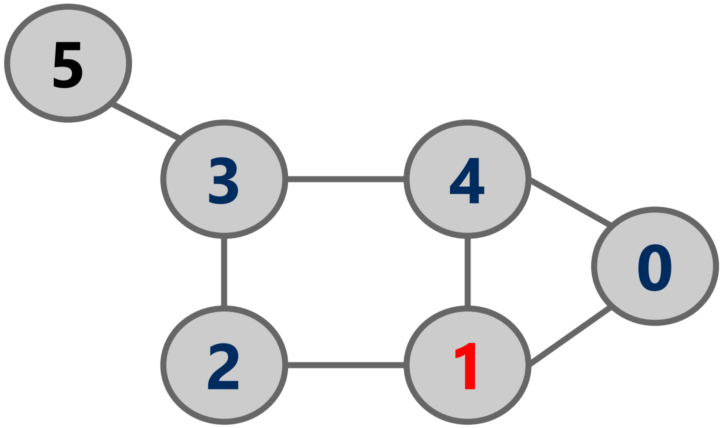
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3
1	3												

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list v_1, v_2, v_3, …
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \ldots$



Edges

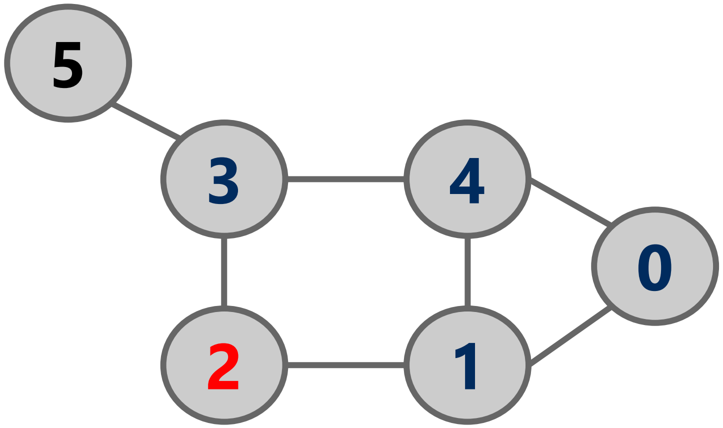
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3
1	3	-1	2	2									

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list v_1, v_2, v_3, …
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \ldots$



Edges

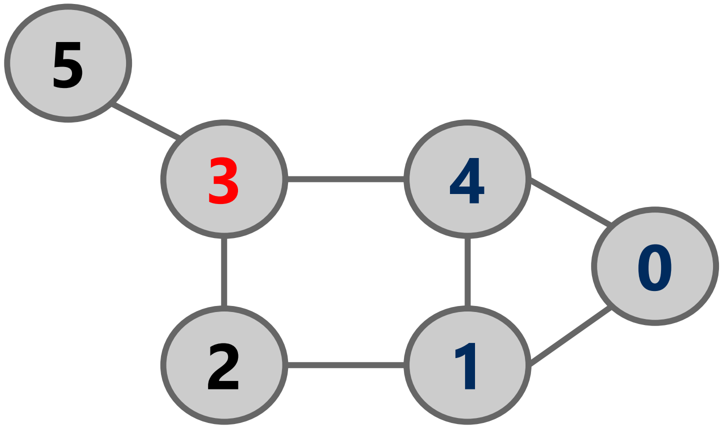
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3
1	3	-1	2	2	-1	2							

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list v_1, v_2, v_3, …
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \ldots$



Edges

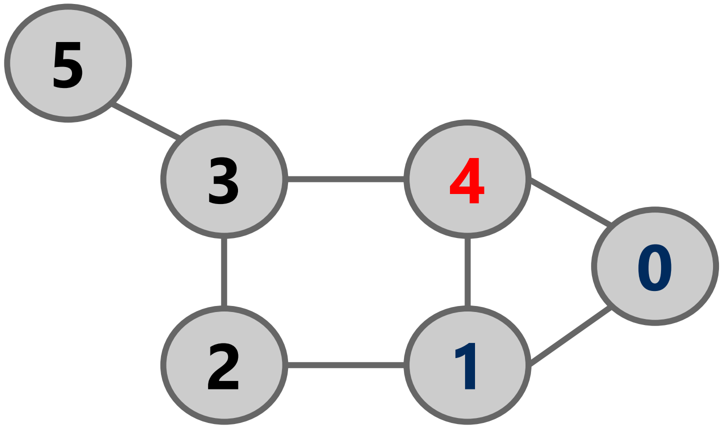
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3
1	3	-1	2	2	-1	2	-1	2	1				

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex v, with a neighbor list v_1, v_2, v_3, ...
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \dots$



Edges

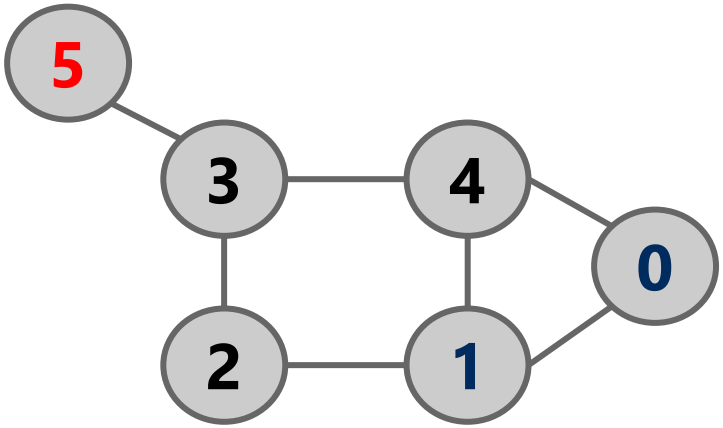
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3
1	3	-1	2	2	-1	2	-1	2	1	-4	1	2	

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list v_1, v_2, v_3, …
  - Store the differences between each two consecutive numbers
    - $(v_1 - v)$, $(v_2 - v_1)$, $(v_3 - v_2)$, …



Edges

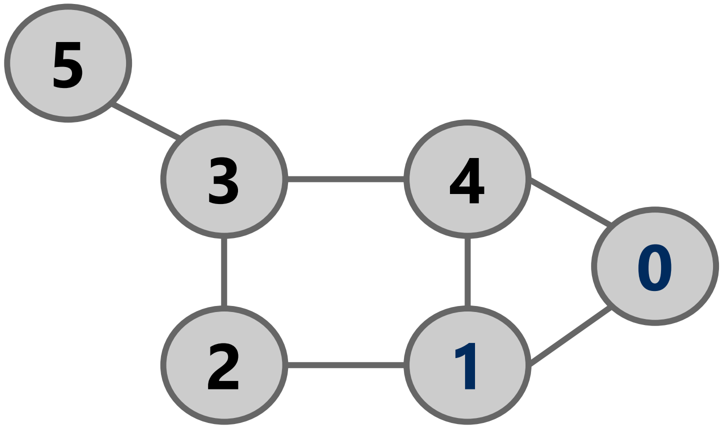
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3
1	3	-1	2	2	-1	2	-1	2	1	-4	1	2	-2

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list v_1, v_2, v_3, …
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \ldots$



Edges

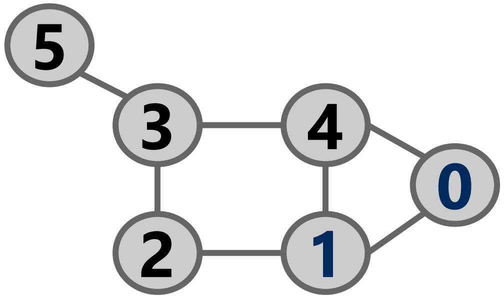
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	3	-1	2	2	-1	2	-1	2	1	-4	1	2	-2

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 3: Use Gamma code to compress the differences**



Edges

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	3	-1	2	2	-1	2	-1	2	1	-4	1	2	-2

Offsets

0	2	5	7	10	13

Gamma Code

- Gamma Code is used to compress data in which small values are much more frequent than large values
- To encode an integer *x*,
  - find T, the largest power of 2 < *x*
  - Encode T as (log T) zeros followed by 1
  - Append the remaining (log T) binary digits of x
- Example: To encode 17: 10001
  - T = 16 = 2^4
  - Gamma code: 0000 1 0001
- 2 floor(log x) + 1
  - much smaller than 32 bits if *x* is small

Graph Compression

- **Goal**: make the differences between vertex labels in each neighbor list small
  - So that their Gamma codes are much less than 32 bits
- For Web Graphs
  - Each vertex is a web page
  - Sort the pages based on their reverse URL (e.g., www.cs.pitt.edu)
  - Most links are local (within the same domain)
    - neighbors will be close to each other in the sorted list
    - Goal achieved
- Other graphs can be relabeled to achieve that goal
  - https://www.cs.cmu.edu/~guyb/papers/BBK03.pdf

Neighborhood connectivity Problem

- We want to keep a set of neighborhoods connected with the minimum cost possible
- **Input:** A set of neighborhoods and a file with the following format:
  - neighborhood i, neighborhood j, cost of connecting the two neighborhoods
  - …
- **Output:** A set of neighborhood pairs to be connected and a total cost such that
  - We can go from any neighborhood to any other **(connected)**
  - The total cost should be minimum (i.e., as small as it can be) **(minimal cost)**

Think Data Structures First!

- How can we structure the input in computer memory?
- Can we use Graphs?
- What about the costs? How can we model that?

We said spatial layouts of graphs were irrelevant

- We define graphs as sets of vertices and edges
- However, we'll certainly want to be able to reason about bandwidth, distance, capacity, etc. of the real world things our graph represents
  - Whether a link is 1 gigabit or 10 megabit will drastically affect our analysis of traffic flowing through a network
  - Having a road between two cities that is a 1 lane country road is very different from having a 4 lane highway
  - If two airports are 2000 miles apart, the number of flights going in and out between them will be drastically different from airports 200 miles apart

We can represent such information with edge weights

- How do we store edge weights?
  - Adjacency matrix?
  - Adjacency list?
  - Do we need a whole new graph representation?
- How do weights affect finding spanning trees/shortest paths?
  - The weighted variants of these problems are called finding the *minimum spanning tree* and the *weighted shortest path*

Minimum spanning trees (MST)

- Graphs can potentially have multiple spanning trees
- MST is the spanning tree that has the minimum sum of the weights of its edges

Prim’s algorithm

- Initialize T to contain the starting vertex
  - T will eventually become the MST
- While there are vertices not in T:
  - Find minimum edge-weight edge that connects a vertex in T to a vertex not yet in T
  - Add the edge with its vertex to T