CS130 Software Engineering

Caltech – Winter 2024 – Lecture 19

Tarjan’s Algorithm for Identifying
Strongly Connected Components in the Dependency Graph

Project 4 Functionality

- Project 4 introduces features that make greater demands on the cycle-detection implementation
- Up through Project 3, a simple DFS traversal of the dependency graph is adequate

	A	B	C
1	=B1 #CIRCREF!	=A1 #CIRCREF!	=A1 / 0 #CIRCREF!

- In Project 4 this is no longer adequate – can't just propagate circular-reference errors

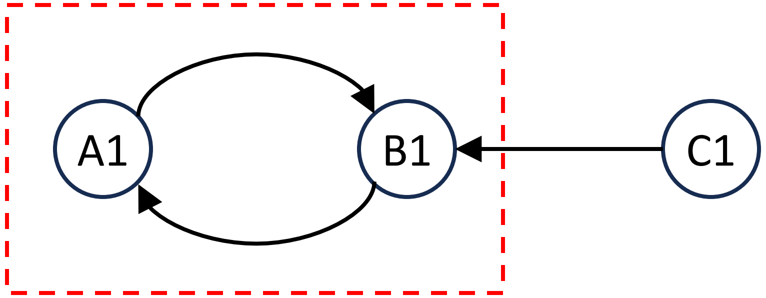
	A	B	C
1	=B1 #CIRCREF!	=A1 #CIRCREF!	=ISERROR(B1) TRUE

Strongly Connected Components

- Dependency graph of this spreadsheet:

	A	B	C
1	=B1 #CIRCREF!	=A1 #CIRCREF!	=ISERROR(B1) TRUE

- Must distinguish between nodes in cycles, versus nodes that reference cycles but are not part of the cycle

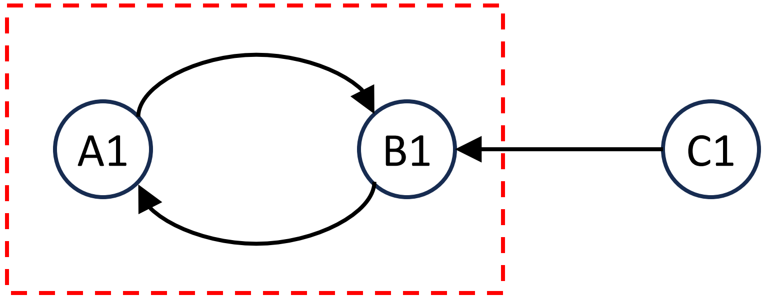


- Nodes in the cycle are called **strongly connected components** (SCCs)
  - Can reach any node in the SCC from any other node in the SCC
  - All nodes in the SCC will be set to #CIRCREF!

Strongly Connected Components (2)

- Dependency graph of this spreadsheet:

	A	B	C
1	=B1 #CIRCREF!	=A1 #CIRCREF!	=ISERROR(B1) TRUE



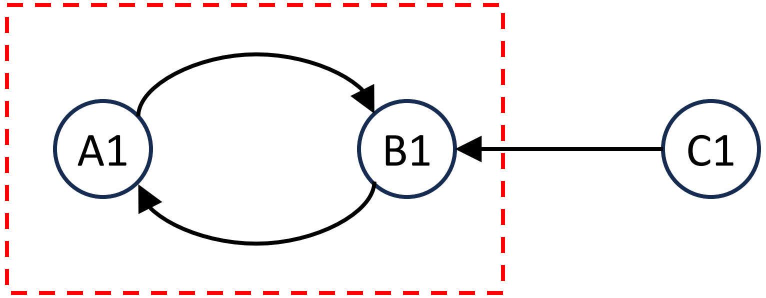
- A single node not in any cycle is a **trivial strongly connected component**
  - e.g. C1 is a trivial SCC
- A multiple-cell SCC is a **non-trivial SCC**
  - e.g. the set [A1, B1] in this example
- A single cell with a dependency on itself is also a non-trivial SCC
  - e.g. if cell D1 was set to the formula “=D1”

Strongly Connected Components (3)

- Dependency graph of this spreadsheet:

	A	B	C
1	=B1 #CIRCREF!	=A1 #CIRCREF!	=ISERROR(B1) TRUE

- In Project 4, correct spreadsheet evaluation requires identifying non-trivial SCCs
- All cells in any non-trivial SCC are set to #CIRCREF!; no other evaluation occurs



- All other cells must be evaluated using normal mechanism
  - Formulas containing e.g. ISERROR(…) function calls will be computed correctly

SCC Algorithms

- Two widely used algorithms for identifying the strongly connected components in a directed graph
  - Tarjan’s algorithm
  - Kosaraju’s algorithm
- Both algorithms are built on top of depth-first search (DFS)
- Both algorithms have been used by CS130 students to identify SCCs
- Tarjan’s algorithm is easier to understand (I think)
- Tarjan’s algorithm also generates a reverse topological sort over the nodes in the graph

Tarjan’s Algorithm: Approach

- As stated, Tarjan’s algorithm operates on an entire graph
  - Can also be used to perform incremental updates, if only one part of the graph has changed
- Iterate over all nodes in graph…
- If a node hasn’t yet been visited by the algorithm, start a DFS traversal from that node
  - All SCCs reachable from that node are identified

```
def tarjan(graph):
    visited = set()
    for node in graph.nodes:
        if not node.id in visited:
            find_sccs(graph, node)


def find_sccs(graph, node):
    # TODO:  Something with DFS?
```

Tarjan's Algorithm: Approach (2)

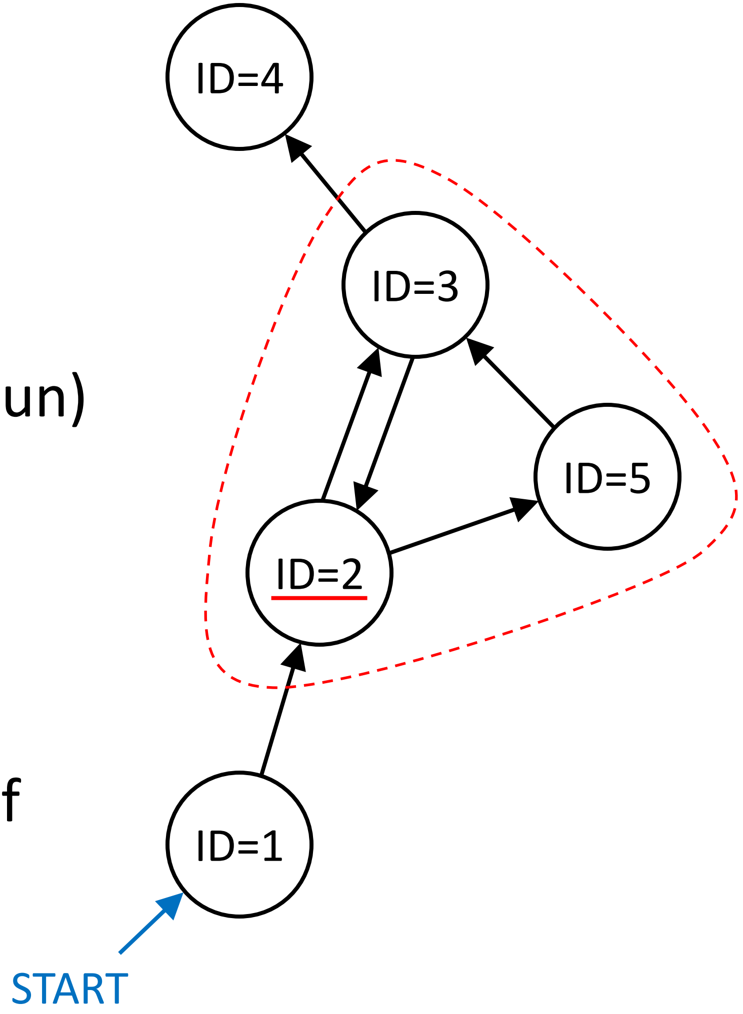
- The DFS traversal from that node will explore a subtree of the graph, rooted at that node
  - All SCCs reachable from that node are identified
- Recall: all nodes in an SCC are reachable from any other node in the SCC
- Thus: a given DFS traversal will never find only a part of an SCC

```
def tarjan(graph):
    visited = set()
    for node in graph.nodes:
        if not node.id in visited:
            find_sccs(graph, node)


def find_sccs(graph, node):
    # TODO:  Something with DFS?
```

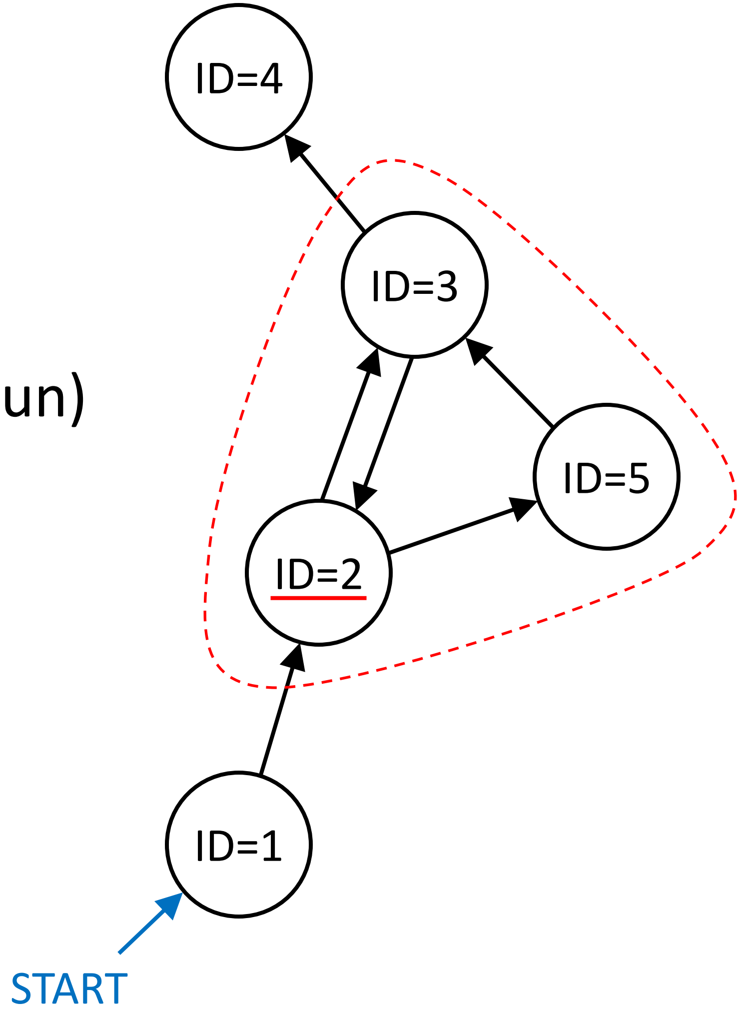
Tarjan's Algorithm: Node IDs and Lowlinks

- Tarjan's algorithm assigns increasing numeric IDs to nodes as it visits them
  - These IDs are used solely by the algorithm, and are independent of any other node-IDs in the program
  - (The specific IDs assigned will likely vary from run to run)
- Each SCC is identified by the node with the lowest ID in that SCC
- Tarjan's algorithm calls this the "lowlink" value
  - A node's **lowlink** value is the lowest ID of any node in the strongly connected component the node is part of



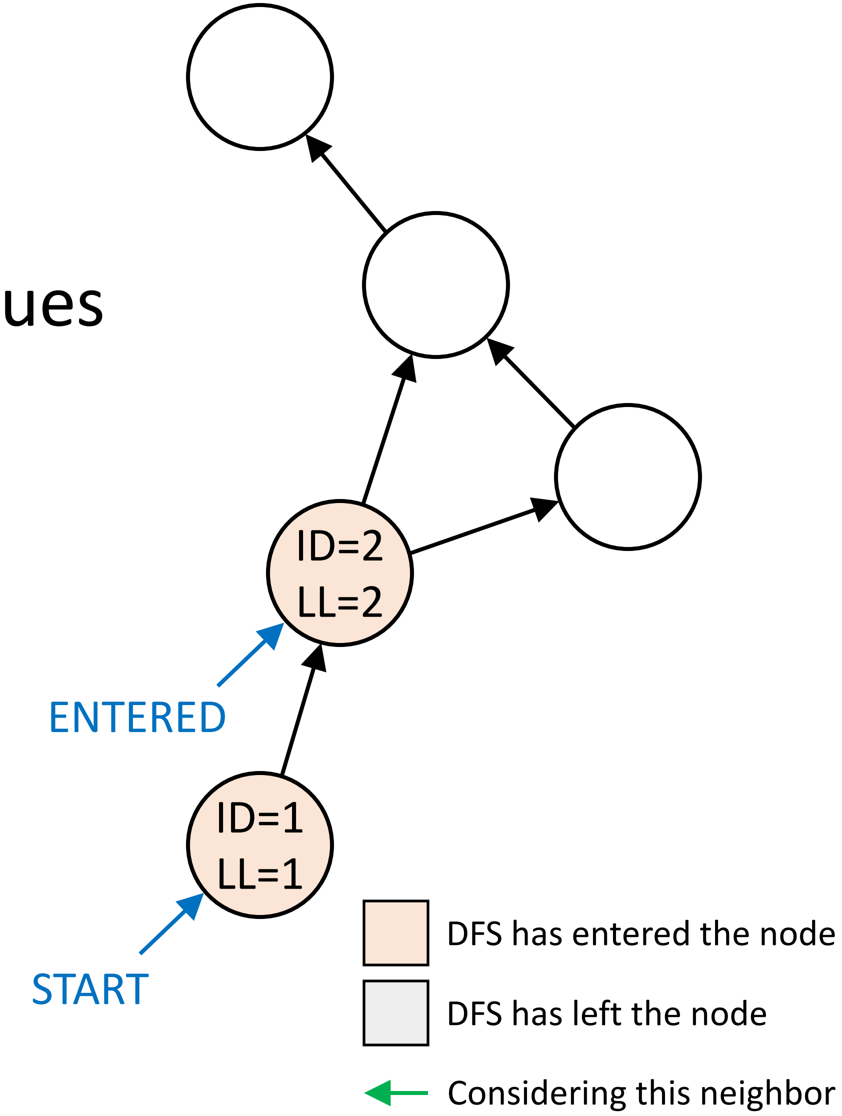
Tarjan’s Algorithm: Node IDs and Lowlinks (2)

- Tarjan’s algorithm assigns increasing numeric IDs to nodes as it visits them
  - These IDs are used solely by the algorithm, and are independent of any other node-IDs in the program
  - (The specific IDs assigned will likely vary from run to run)
- Each SCC is identified by the node with the lowest ID in that SCC
- If each SCC in the graph is **condensed** down to a single node with an ID of the SCC’s lowlink value:
  - The graph becomes a directed acyclic graph
  - A node *i* is a predecessor of node *j* if *i* < *j*



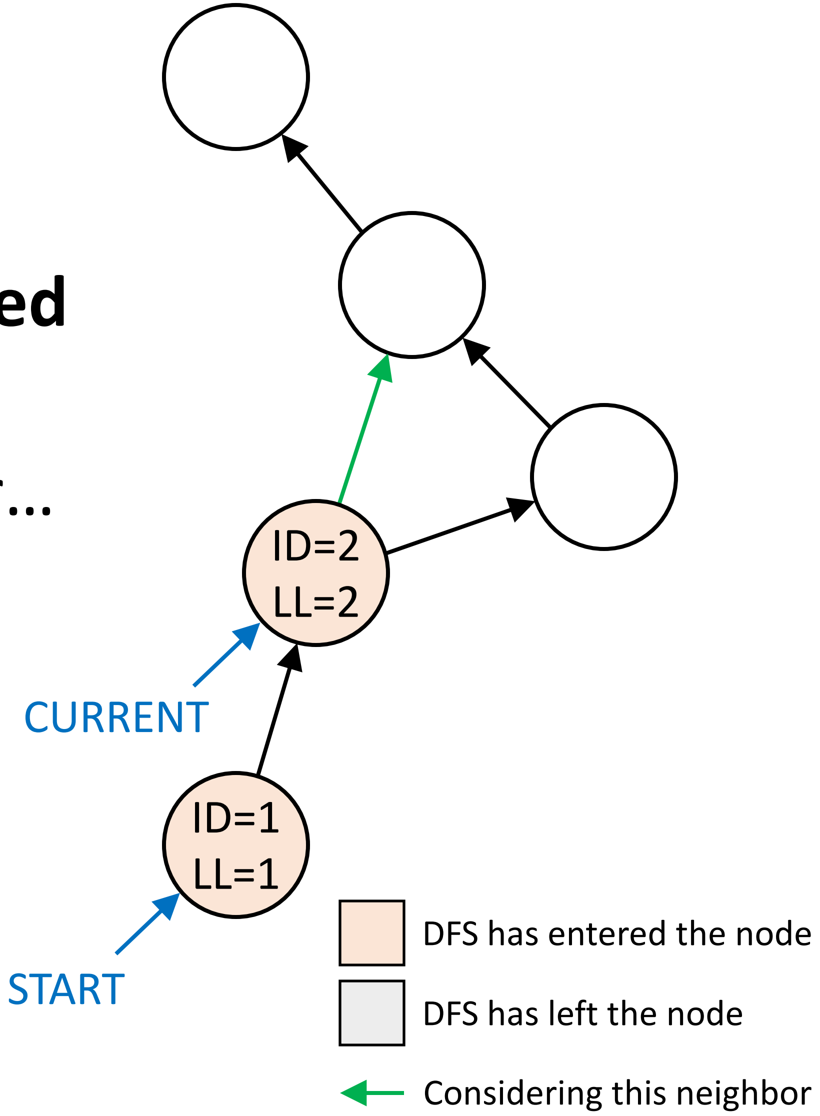
Tarjan's Algorithm: Computing Lowlinks

- Of course, the algorithm doesn't automatically know each SCC's lowlink value…
- As the algorithm traverses the graph, lowlink values are propagated according to specific rules
- When Tarjan's algorithm visits a node for the first time:
  - (i.e. the node doesn't yet have an ID)
  - The next available ID value is assigned to the node
  - The node's lowlink value is set to its own ID value
  - (Every node is in a trivial SCC with itself)



Tarjan’s Algorithm: Computing Lowlinks (2)

- Once the algorithm has entered a node, neighbor nodes fall into two categories
- **Category 1: The neighbor has not yet been visited**
  - It has neither an ID nor a lowlink value
- Recursively invoke the algorithm on the neighbor…
  - This will set the neighbor node’s ID, and compute its lowlink value
  - Once the recursive invocation completes, update the current node’s lowlink value
  - node.lowlink = min(node.lowlink, neighbor.lowlink)



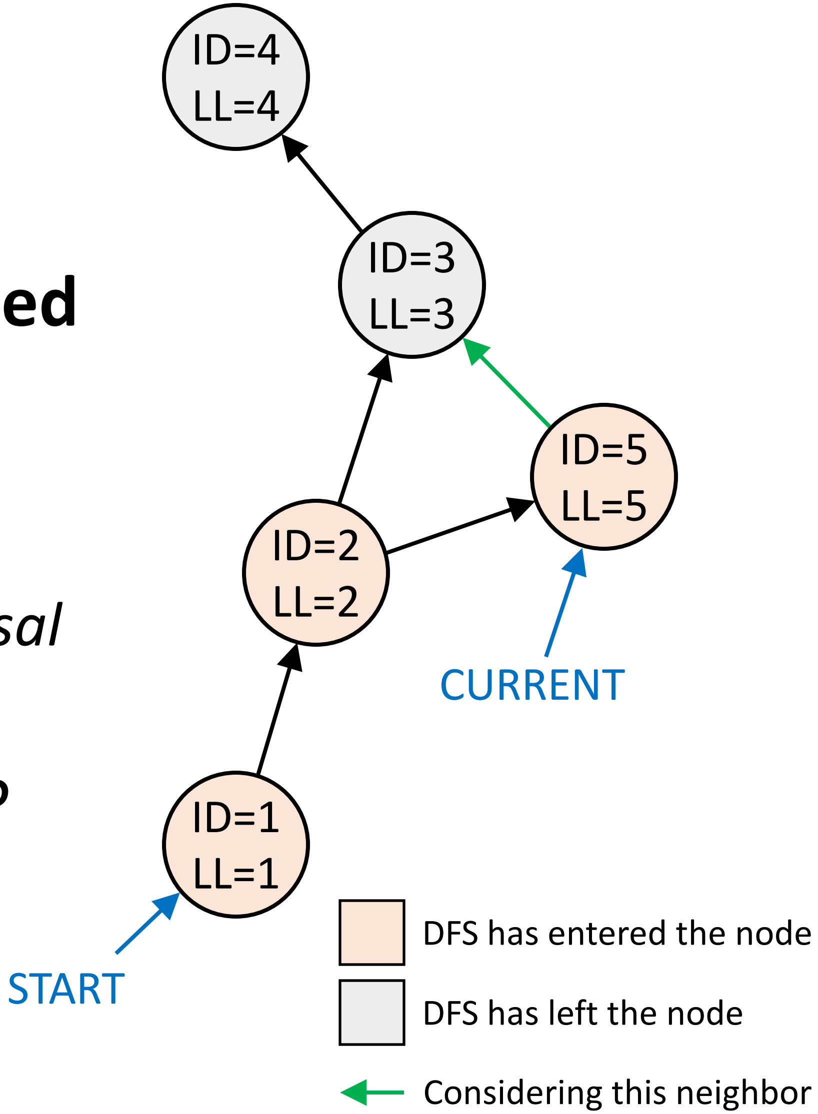
🟧 DFS has entered the node

⬜ DFS has left the node

← Considering this neighbor

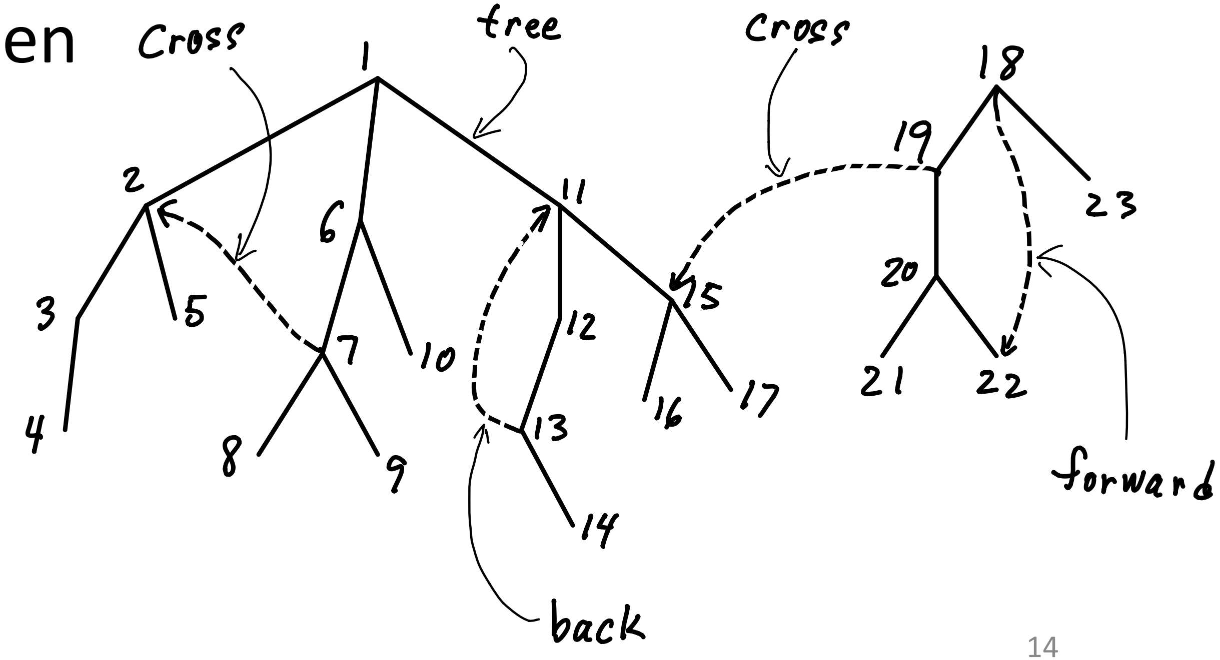
Tarjan's Algorithm: Computing Lowlinks (3)

- Once the algorithm has entered a node, neighbor nodes fall into two categories
- **Category 2: The neighbor has already been visited**
  - It has both an ID and a lowlink value
  - The neighbor's lowlink may not yet be its final value, if we have entered but not yet left the neighbor
  - *The neighbor may also be from a different DFS traversal*
- The algorithm has already visited the neighbor before… *am I inside a cycle (i.e. a nontrivial SCC)?*
- **We need more info to answer this question**



Tarjan’s Algorithm: Computing Lowlinks (4)

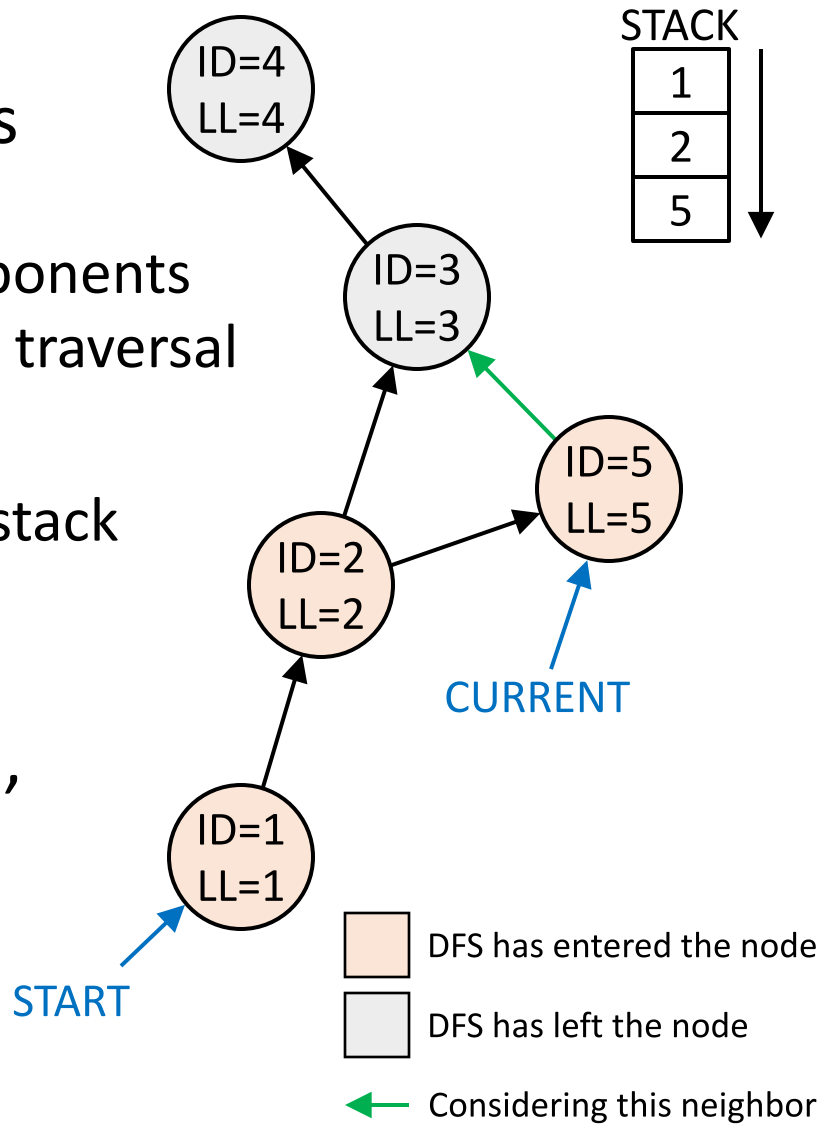
- Tarjan’s algorithm must discern between different kinds of links between nodes in the graph
- When traversing the graph via DFS, non-trivial SCCs will include at least one **back-link**, pointing to some node entered earlier in the DFS
- We may also find **cross-links** between subtrees within the graph
  - Either part of this DFS traversal, or part of some previous DFS traversal
  - We don’t care about cross-links
- **How to distinguish between back-links and cross-links?**



(diagram stolen from CMU lecture)

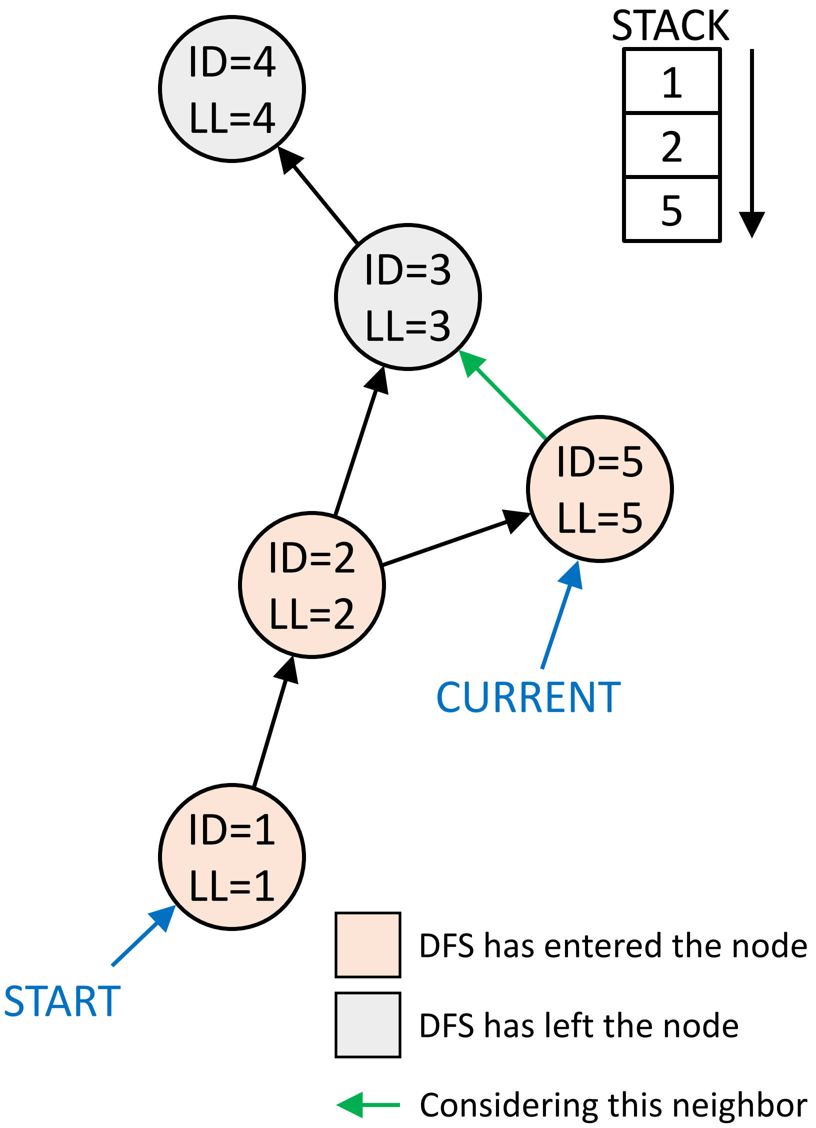
Tarjan's Algorithm: Computing Lowlinks (5)

- Tarjan's algorithm also maintains a stack of nodes that it has entered during DFS traversal
  - This stack is used to identify strongly connected components
  - NOTE: This is separate from whatever is used for DFS traversal
- The stack is governed by special rules:
  - When we enter a node, it is always pushed onto this stack
  - Nodes are only popped off when we identify SCCs
- If we are in a non-trivial SCC, we will eventually reach a node in the SCC we have already entered, but have not yet left
  - Use our stack of nodes to see if this is the case



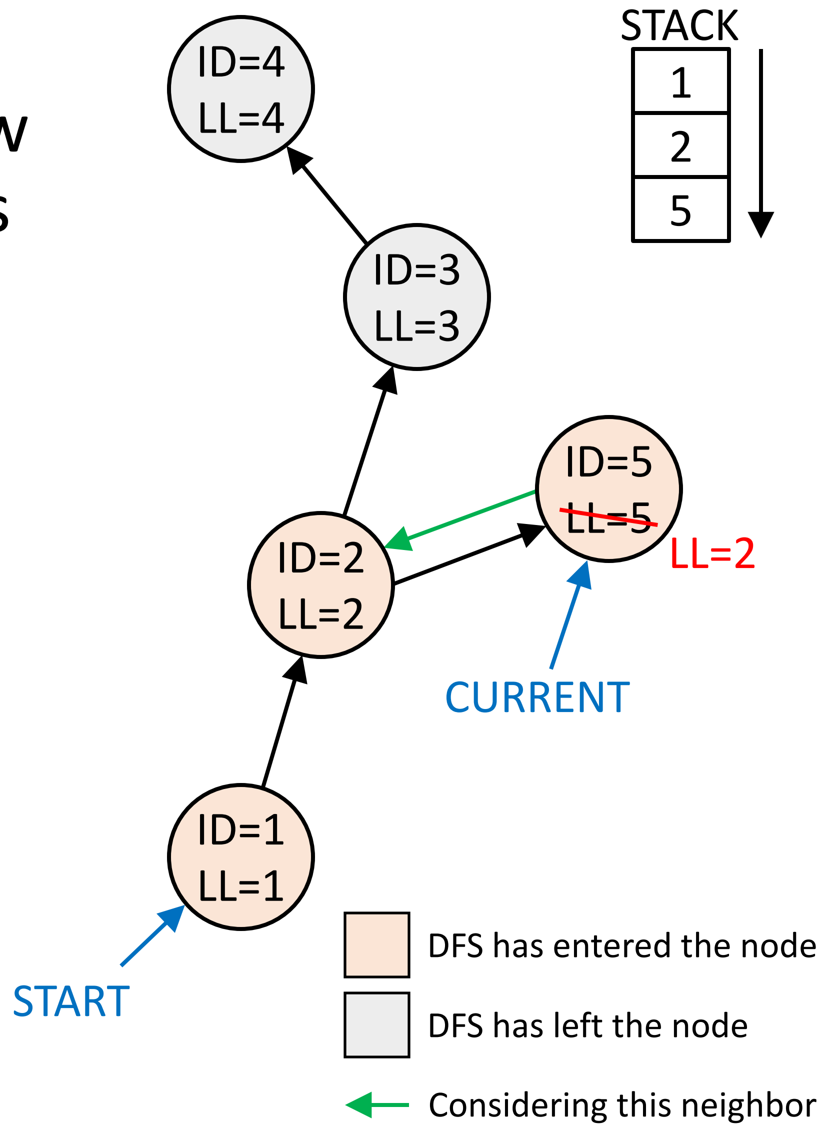
Tarjan’s Algorithm: Computing Lowlinks (6)

- In the example to the right, node 5’s neighbor (ID=3) has already been visited, but is not currently on the stack
- Don’t need to make any changes to node 5’s lowlink value
- This example only contains trivial SCCs
- (TODO: How *exactly* to update our stack?)



Tarjan's Algorithm: Computing Lowlinks (7)

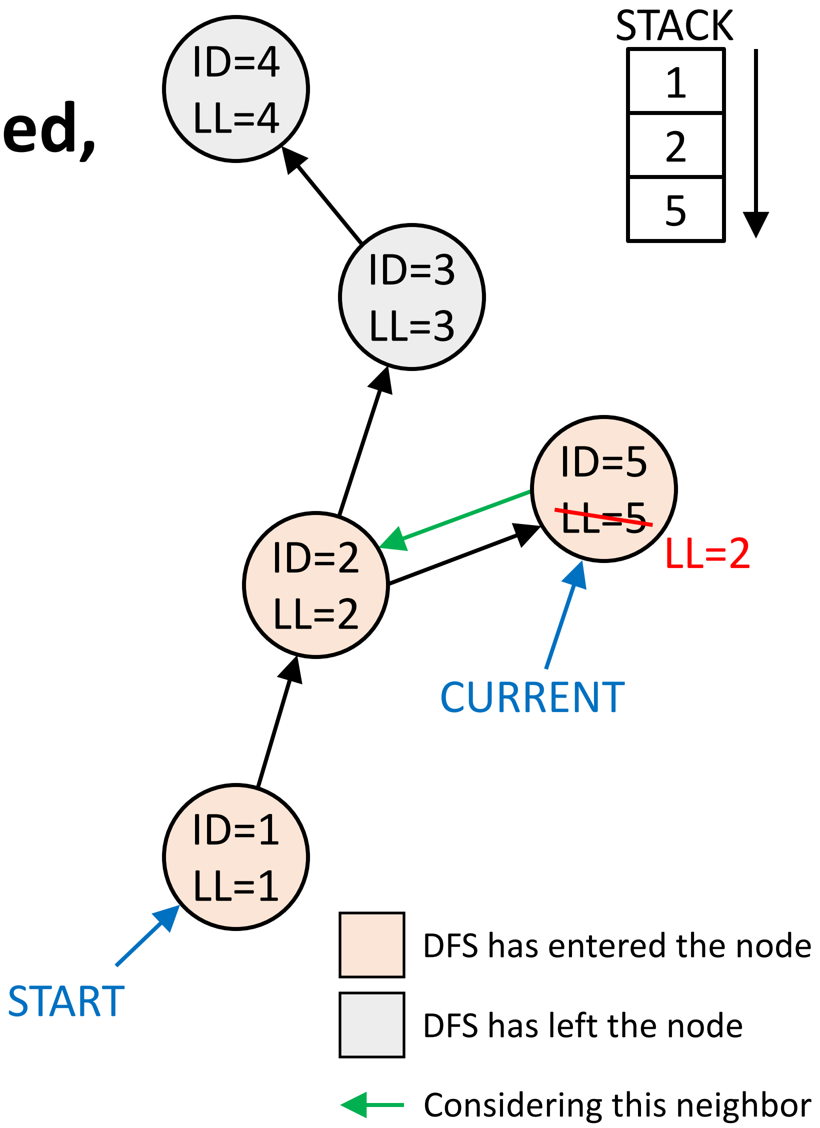
- Now a slightly modified graph, where node 5 now references node 2 in a cycle, same stack contents
- Node 2 has already been visited, and it also appears in the stack
- Node 5 is in the same SCC as node 2. Need to update node 5's lowlink based on node 2
- (TODO: How *exactly* to update our stack?)



Tarjan’s Algorithm: Computing Lowlinks (8)

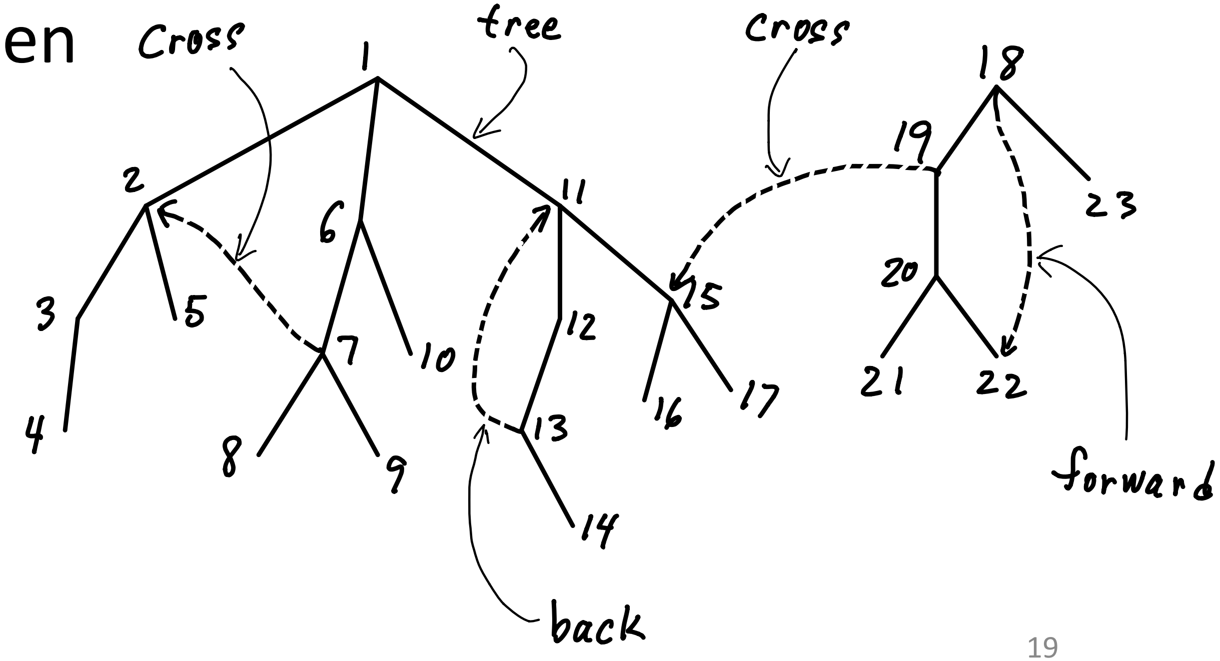
- **Category 2: The neighbor has already been visited, <u>and</u> the neighbor also appears on the stack**
  - if neighbor.id in stack:
  - my.lowlink = min(my.lowlink, neighbor.id)

- In this formulation it’s important to use neighbor.id and not neighbor.lowlink
  - This is the approach of the original paper
  - (See references at end for more detailed explanation, and an alternate approach)



Tarjan’s Algorithm: Computing Lowlinks (9)

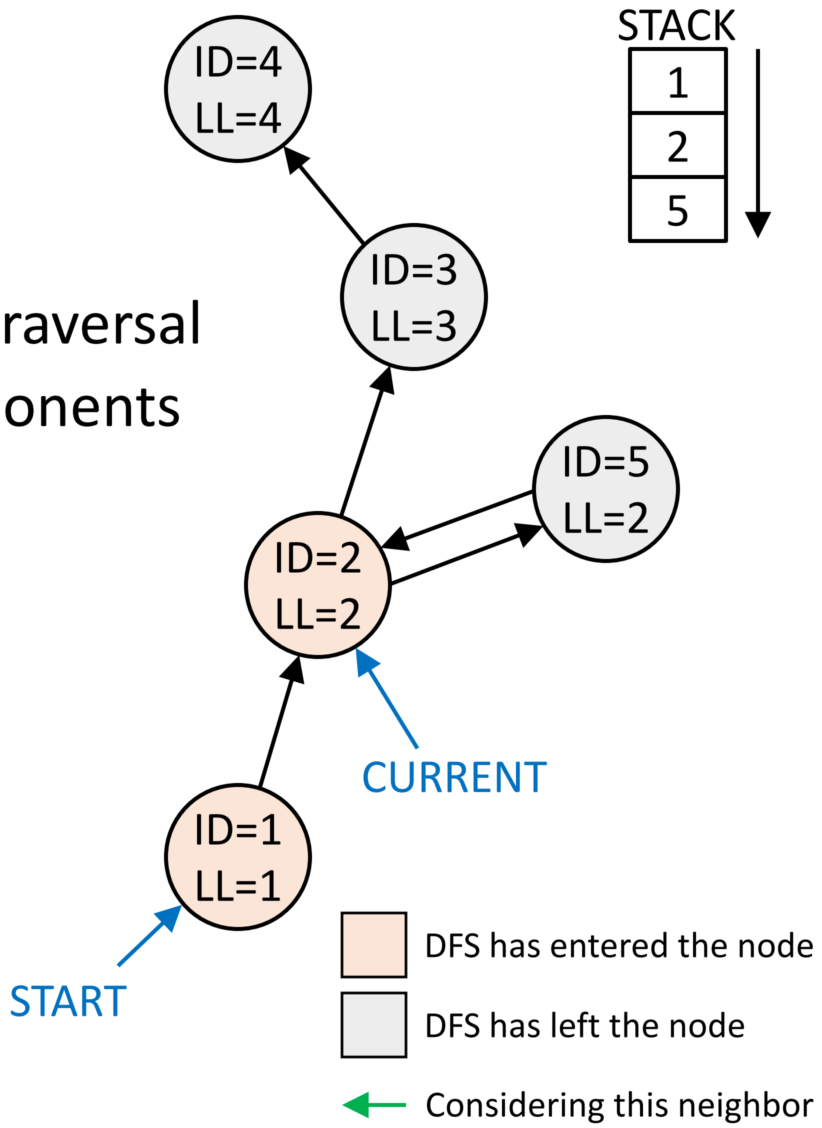
- Tarjan’s algorithm must discern between different kinds of links between nodes in the graph
- When traversing the graph via DFS, non-trivial SCCs will include at least one **back-link**, pointing to some node entered earlier in the DFS
- We may also find **cross-links** between subtrees within the graph
  - Either part of this DFS traversal, or part of some previous DFS traversal
  - We don’t care about cross-links
- The stack allows us to distinguish between back-links and cross-links



(diagram stolen from CMU lecture)

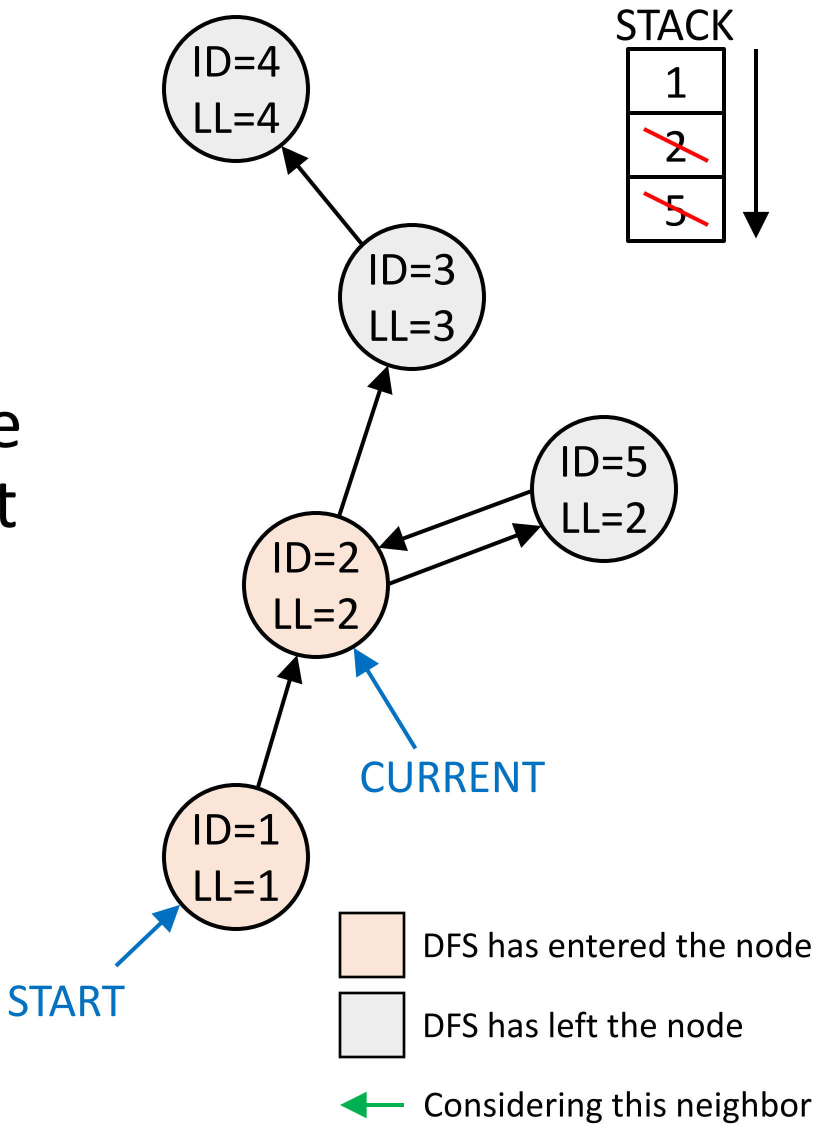
Tarjan's Algorithm: Updating the Stack

- **How do we update the stack?**
- Recall:
  - The stack records nodes we have entered in the DFS traversal
  - The stack is used to identify strongly connected components by finding their back-links
  - When we enter a node, it is pushed onto this stack
  - Nodes are only popped off when we identify SCCs
- Also:
  - Each SCC is identified by the node with the lowest ID in that SCC



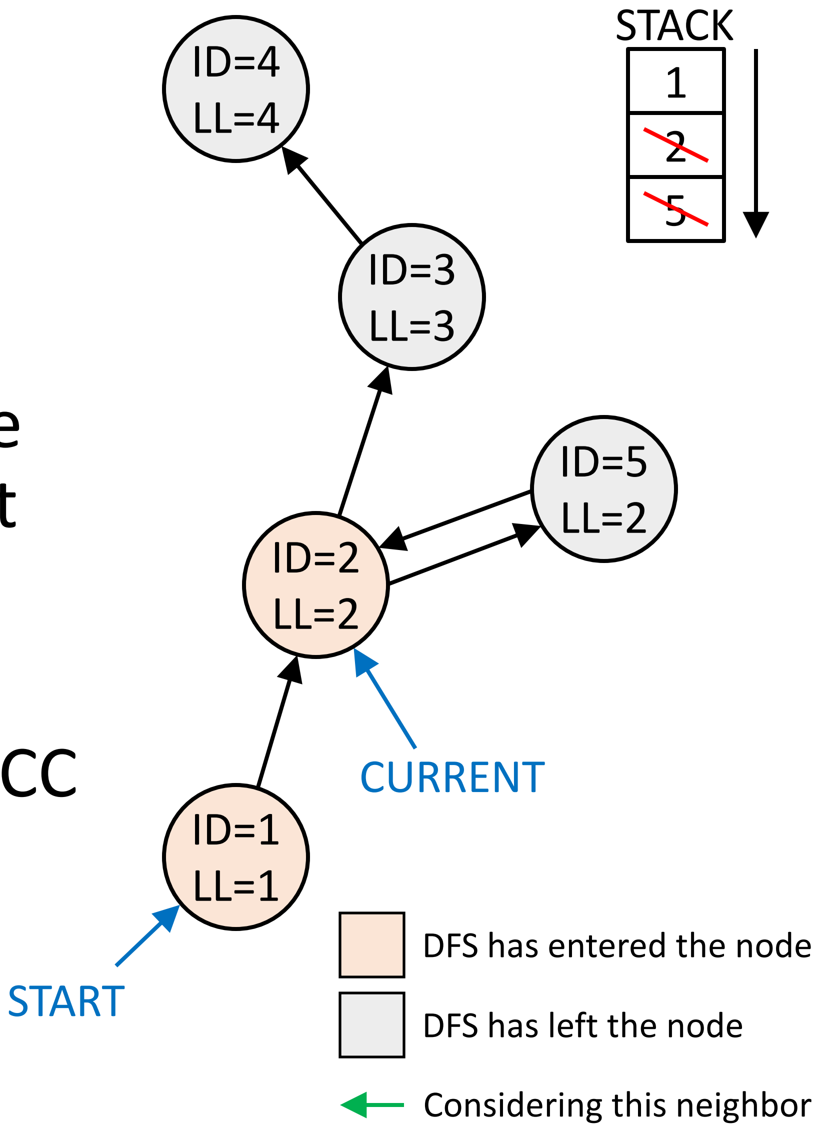
Tarjan’s Algorithm: Updating the Stack (2)

- **How do we update the stack?**
- When we are ready to leave a node, compare its ID and lowlink values
- If these values are the same, then this node is the starting point of a strongly connected component
- Pop all nodes in the SCC off the stack until we have also popped off the current node’s ID
  - *Must not forget the node that identifies the SCC*



Tarjan’s Algorithm: Updating the Stack (3)

- **How do we update the stack?**
- When we are ready to leave a node, compare its ID and lowlink values
- If these values are the same, then this node is the starting point of a strongly connected component
- As these nodes are popped off the stack, can be recorded into a data structure representing the SCC
  - Whether the SCC is trivial or non-trivial can also be stored in the structure representing the SCC



Tarjan’s Algorithm – Wikipedia Version

```
algorithm tarjan is
    input: graph G = (V, E)
    output: set of SCCs (sets of vertices)

    index := 0
    S := empty stack
    for each v in V do
        if v.index is undefined then
            strongconnect(v)

    function strongconnect(v)
        // Set index, initial lowlink for v
        v.index := index
        v.lowlink := index
        index := index + 1

        // Record v on the stack
        S.push(v)
        v.onStack := true

        ...
```

```
// Consider neighbors of v to compute v.lowlink
for each (v, w) in E do
    if w.index is undefined then
        // Successor w has not yet been visited
        strongconnect(w)
        v.lowlink := min(v.lowlink, w.lowlink)

    else if w.onStack then
        // Successor w has been visited, and is
        // also on stack S and is therefore in
        // the current SCC.
        v.lowlink := min(v.lowlink, w.index)

// If v is a root of an SCC, pop vertices off
// the stack to generate/record the SCC.
if v.lowlink = v.index then
    start a new strongly connected component
    repeat
        w := S.pop()
        w.onStack := false
        add w to the current SCC
    while w ≠ v
    store or output the current SCC
```

References

- Wikipedia article on Tarjan's algorithm (ofc)
- CMU lecture notes on strongly connected components in graphs
- A great visual explanation of Tarjan's algorithm (YouTube)
  - Note that this implementation differs slightly from the pseudocode on Wikipedia, in this lecture, in CMU's notes, etc!
  - The CMU lecture notes also include this alternate formulation in the "Implementation" section

A Common Variation of Tarjan's Algorithm

```
public class TarjanSCC {
  int n, count, comp;
  int[] num, low, answer;
  boolean[] onStack;
  Stack<Integer> stack;
  int[][] graph;

  public int[] strong(int[][] g) {
    graph = g;
    n = graph.length;
    num = new int[n];
    low = new int[n];
    answer = new int[n];
    onStack = new boolean[n];
    stack = new Stack<Integer>();
    count = 0;
    comp = 0;
    for (int x=0; x<n; x++) DFS(x);
    return answer;
  }

  ...
```

```
  void DFS(int v) {
    if (num[v] != 0) return;     // Already visited

    num[v] = low[v] = ++count;   // Assign ID and
    stack.push(v);               // and push
    onStack[v] = true;           // onto stack

    for (int w: graph[v]) DFS(w);

    for (int w: graph[v])       // Note different
      if (onStack[w])           // update!
        low[v] = min(low[v], low[w]);

    if (num[v] == low[v]) {     // Construct any
      while (true) {            // SCC that was
        int x = stack.pop();    // identified
        onStack[x] = false;
        answer[x] = comp;
        if (x == v) break;
      }
      comp++;
    }
  }
}
```

Implementation Notes

- Use helper classes to manage the state required for Tarjan’s algorithm
  - Don’t be like these implementations! 🤮
  - Your implementation can in fact be very clean and readable
- The recursive version of the algorithm is straightforward…
- Making it iterative can be a bit more challenging
- The variation may be a bit easier for this
  - Always calls DFS on all neighbors…
  - Can break the operation into two phases, “before visiting neighbors” and “after visiting neighbors”
  - May work well with iterative DFS approach shown in Lecture 3