Applications of BFS and DFS

Madhavan Mukund

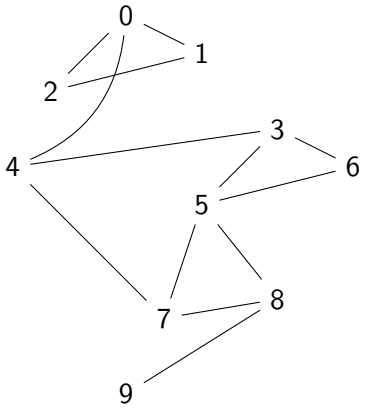
https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 4

BFS and DFS

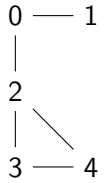
- BFS and DFS systematically compute reachability in graphs
- BFS works level by level
  - Discovers shortest paths in terms of number of edges
- DFS explores a vertex as soon as it is visited neighbours
  - Suspend a vertex while exploring its neighbours
  - DFS numbering describes the order in which vertices are explored
- Beyond reachability, what can we find out about a graph using BFS/DFS?



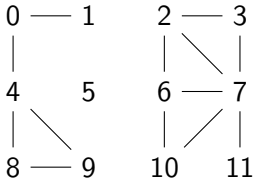
Connectivity

- An undirected graph is connected if every vertex is reachable from every other vertex

Connected Graph



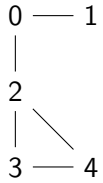
Disconnected Graph



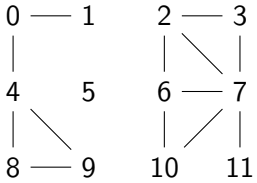
Connectivity

- An undirected graph is connected if every vertex is reachable from every other vertex
- In a disconnected graph, we can identify the connected components

Connected Graph



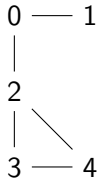
Disconnected Graph



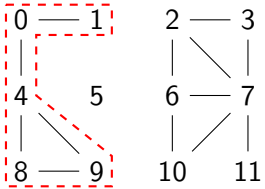
Connectivity

- An undirected graph is connected if every vertex is reachable from every other vertex
- In a disconnected graph, we can identify the connected components
  - Maximal subsets of vertices that are connected

Connected Graph



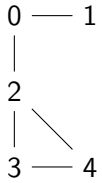
Disconnected Graph



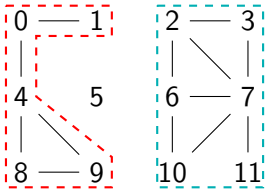
Connectivity

- An undirected graph is connected if every vertex is reachable from every other vertex
- In a disconnected graph, we can identify the connected components
  - Maximal subsets of vertices that are connected

Connected Graph



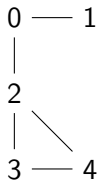
Disconnected Graph



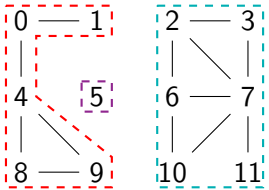
Connectivity

- An undirected graph is connected if every vertex is reachable from every other vertex
- In a disconnected graph, we can identify the connected components
  - Maximal subsets of vertices that are connected
  - Isolated vertices are trivial components

Connected Graph



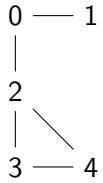
Disconnected Graph



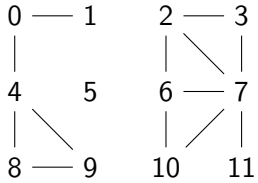
Identifying connected components

- Assign each vertex a component number

Connected Graph



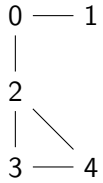
Disconnected Graph



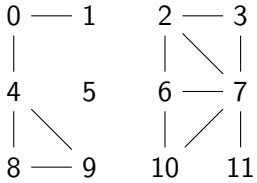
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0

Connected Graph



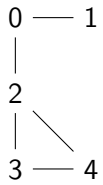
Disconnected Graph



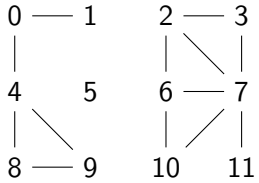
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
  - Initialize component number to 0

Connected Graph



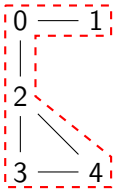
Disconnected Graph



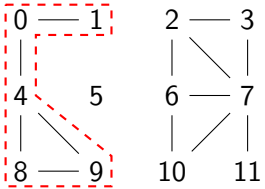
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
  - Initialize component number to 0
  - All visited nodes form a connected component

Connected Graph



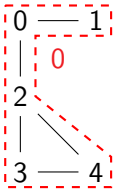
Disconnected Graph



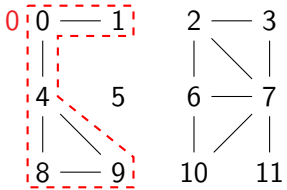
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
  - Initialize component number to 0
  - All visited nodes form a connected component
  - Assign each visited node component number 0

Connected Graph



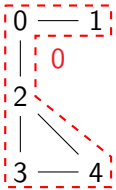
Disconnected Graph



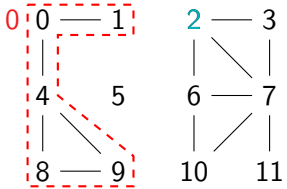
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
  - Initialize component number to 0
  - All visited nodes form a connected component
  - Assign each visited node component number 0
- Pick smallest unvisited node j

Connected Graph



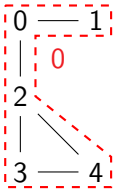
Disconnected Graph



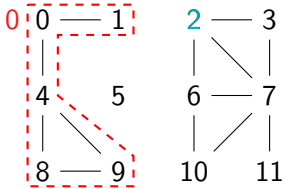
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
  - Initialize component number to 0
  - All visited nodes form a connected component
  - Assign each visited node component number 0
- Pick smallest unvisited node j
  - Increment component number to 1

Connected Graph



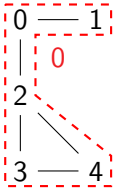
Disconnected Graph



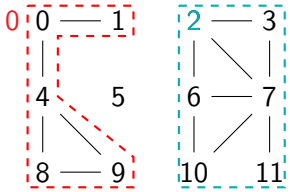
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
  - Initialize component number to 0
  - All visited nodes form a connected component
  - Assign each visited node component number 0
- Pick smallest unvisited node j
  - Increment component number to 1
  - Run BFS/DFS from node j

Connected Graph



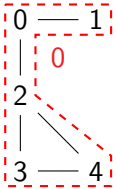
Disconnected Graph



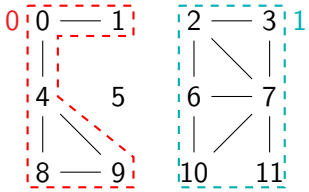
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
  - Initialize component number to 0
  - All visited nodes form a connected component
  - Assign each visited node component number 0
- Pick smallest unvisited node j
  - Increment component number to 1
  - Run BFS/DFS from node j
  - Assign each visited node component number 1

Connected Graph



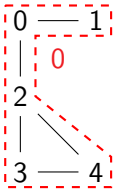
Disconnected Graph



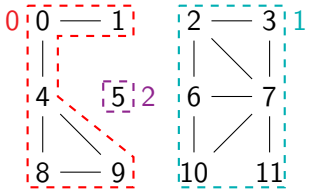
Identifying connected components

- Assign each vertex a component number
- Start BFS/DFS from vertex 0
  - Initialize component number to 0
  - All visited nodes form a connected component
  - Assign each visited node component number 0
- Pick smallest unvisited node j
  - Increment component number to 1
  - Run BFS/DFS from node j
  - Assign each visited node component number 1
- Repeat until all nodes are visited

Connected Graph



Disconnected Graph



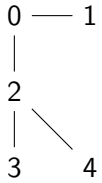
Identifying connected components

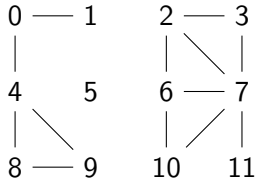
- Assign each vertex a component number
- Start BFS/DFS from vertex 0
  - Initialize component number to 0
  - All visited nodes form a connected component
  - Assign each visited node component number 0
- Pick smallest unvisited node j
  - Increment component number to 1
  - Run BFS/DFS from node j
  - Assign each visited node component number 1
- Repeat until all nodes are visited

```
def Components(AList):
  component = {}
  for i in AList.keys():
    component[i] = -1

  (compid,seen) = (0,0)

  while seen <= max(AList.keys()):
    startv = min([i for i in AList.keys()
                  if component[i] == -1])
    visited = BFSList(AList,startv)
    for i in visited.keys():
      if visited[i]:
        seen = seen + 1
        component[i] = compid
    compid = compid + 1

  return(component)
```

Detecting cycles

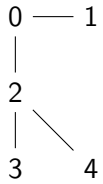
- A cycle is a path (technically, a walk) that starts and ends at the same vertex

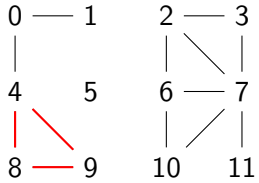




Detecting cycles

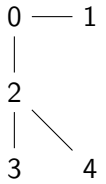
- A cycle is a path (technically, a walk) that starts and ends at the same vertex
  - $4 - 8 - 9 - 4$ is a cycle

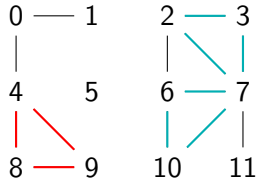




Detecting cycles

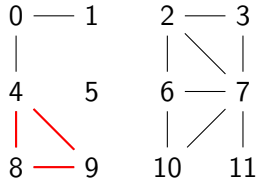
- A cycle is a path (technically, a walk) that starts and ends at the same vertex
  - $4 - 8 - 9 - 4$ is a cycle
  - Cycle may repeat a vertex: $2 - 3 - 7 - 10 - 6 - 7 - 2$





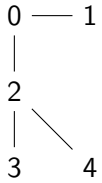
Detecting cycles

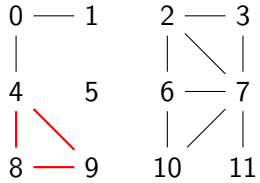
- A cycle is a path (technically, a walk) that starts and ends at the same vertex
  - $4 - 8 - 9 - 4$ is a cycle
  - Cycle may repeat a vertex: $2 - 3 - 7 - 10 - 6 - 7 - 2$
  - Cycle should not repeat edges: $i - j - i$ is not a cycle, e.g., $2 - 4 - 2$



Detecting cycles

- A cycle is a path (technically, a walk) that starts and ends at the same vertex
  - $4 - 8 - 9 - 4$ is a cycle
  - Cycle may repeat a vertex: $2 - 3 - 7 - 10 - 6 - 7 - 2$
  - Cycle should not repeat edges: $i - j - i$ is not a cycle, e.g., $2 - 4 - 2$
  - Simple cycle — only repeated vertices are start and end

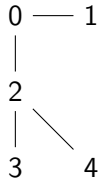




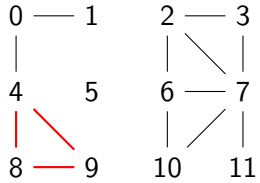
Detecting cycles

- A cycle is a path (technically, a walk) that starts and ends at the same vertex
  - $4 - 8 - 9 - 4$ is a cycle
  - Cycle may repeat a vertex: $2 - 3 - 7 - 10 - 6 - 7 - 2$
  - Cycle should not repeat edges: $i - j - i$ is not a cycle, e.g., $2 - 4 - 2$
  - Simple cycle — only repeated vertices are start and end
- A graph is acyclic if it has no cycles

Acyclic Graph



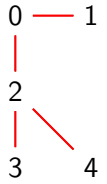
Graph with cycles



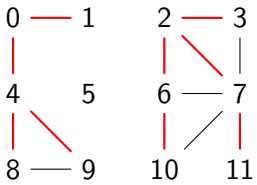
BFS tree

- Edges explored by BFS form a tree
  - Technically, one tree per component
  - Collection of trees is a forest

Acyclic Graph



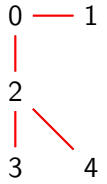
Graph with cycles



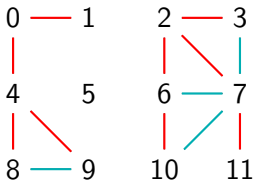
BFS tree

- Edges explored by BFS form a tree
  - Technically, one tree per component
  - Collection of trees is a forest
- Any non-tree edge creates a cycle
  - Detect cycles by searching for non-tree edges

Acyclic Graph

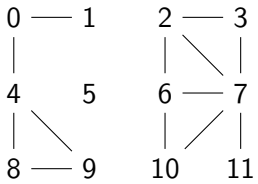


Graph with cycles



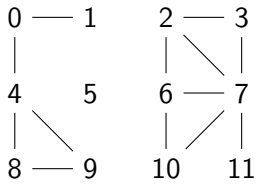
DFS tree

- Maintain a DFS counter, initially 0



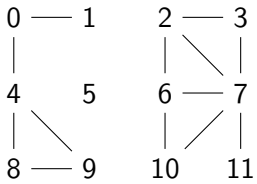
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node



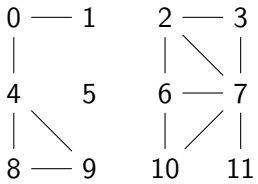
DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)



DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

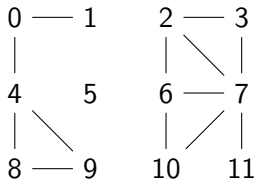


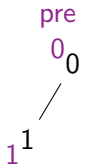
pre

0 0

DFS tree

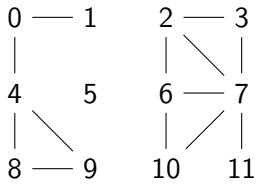
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

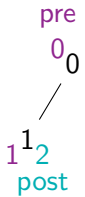




DFS tree

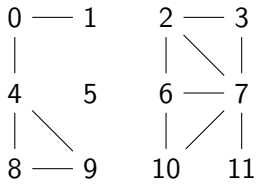
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

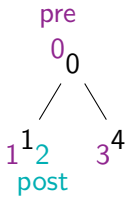




DFS tree

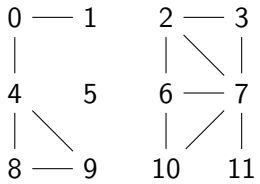
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

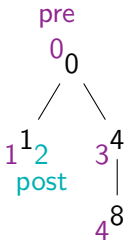




DFS tree

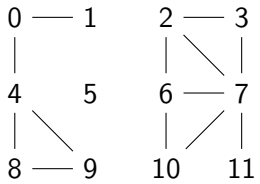
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

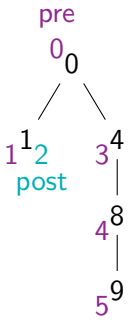




DFS tree

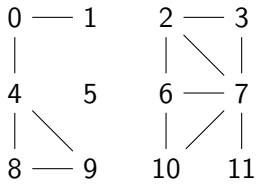
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

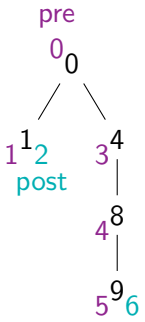




DFS tree

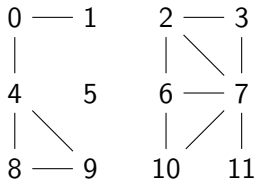
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

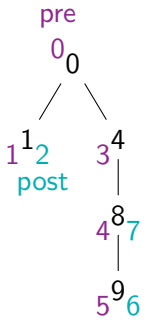




DFS tree

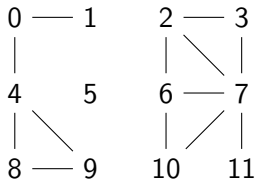
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

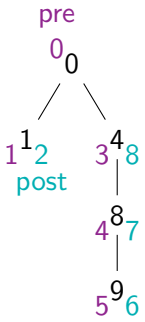




DFS tree

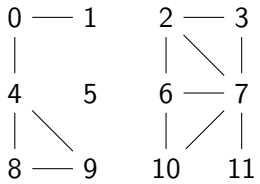
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

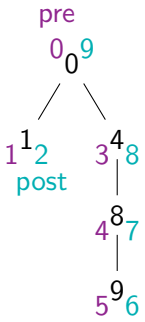




DFS tree

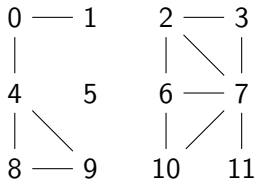
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

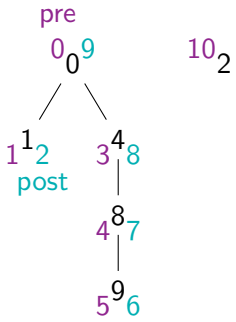




DFS tree

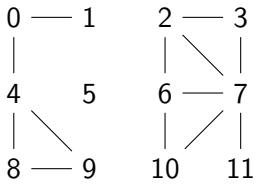
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

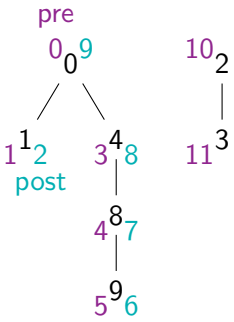




DFS tree

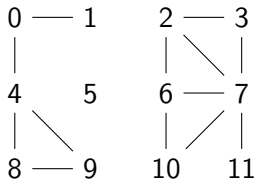
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

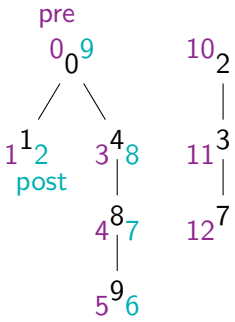




DFS tree

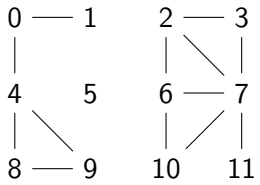
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

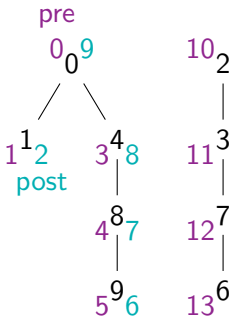




DFS tree

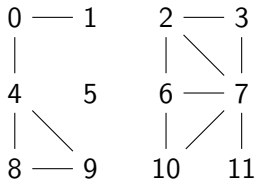
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

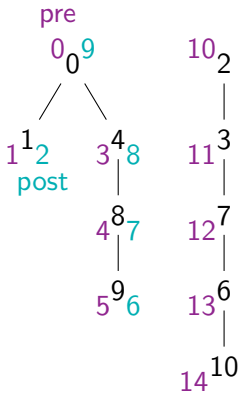




DFS tree

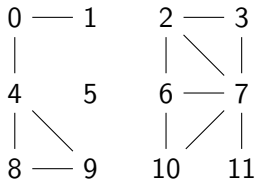
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

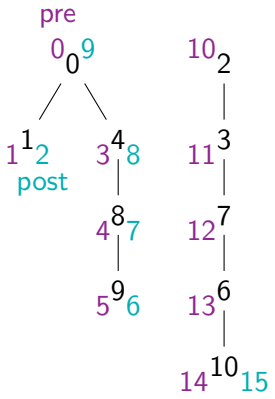




DFS tree

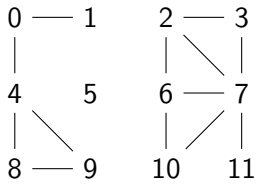
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

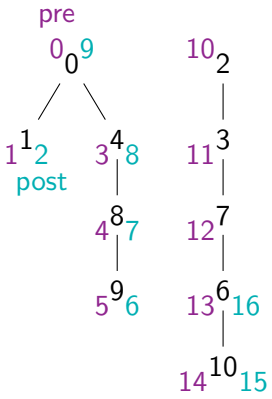




DFS tree

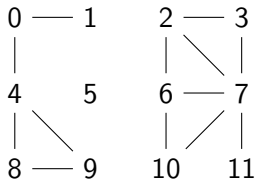
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

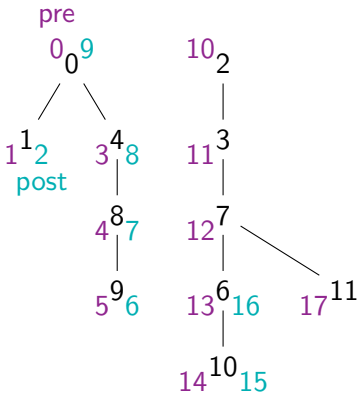




DFS tree

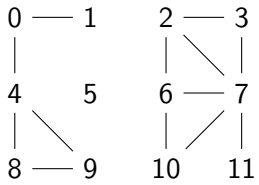
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

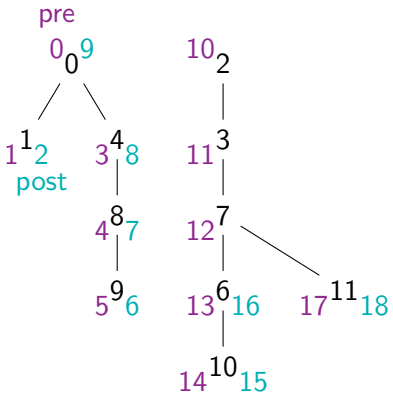




DFS tree

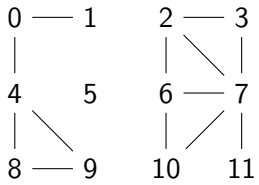
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

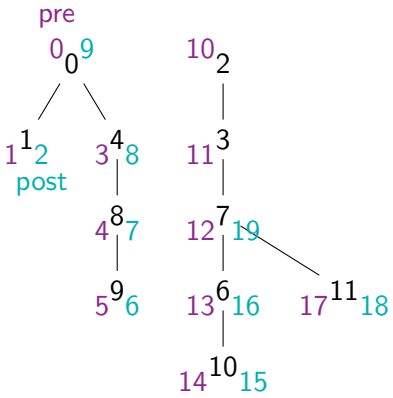




DFS tree

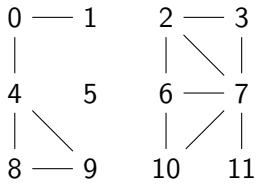
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

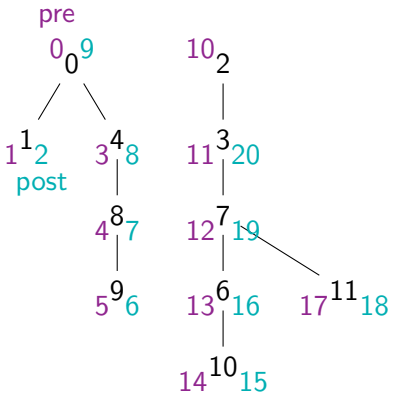




DFS tree

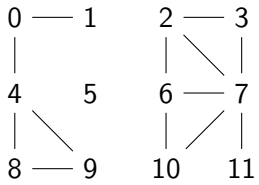
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

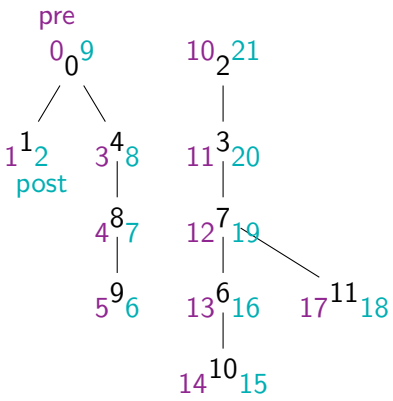




DFS tree

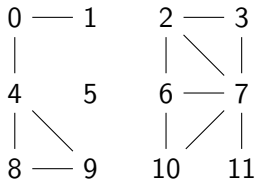
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

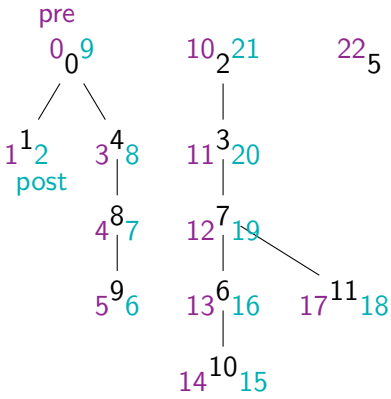




DFS tree

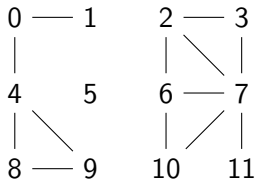
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

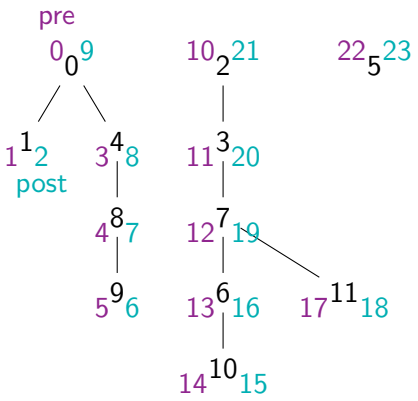




DFS tree

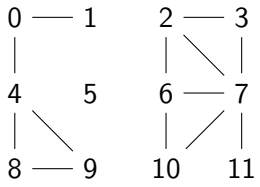
- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)



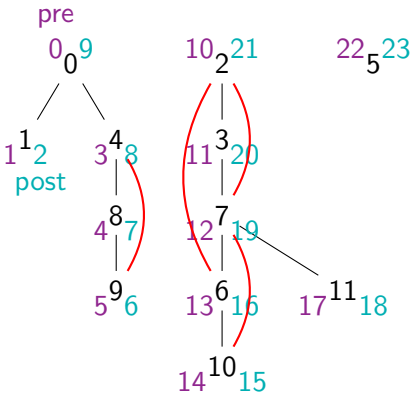


DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

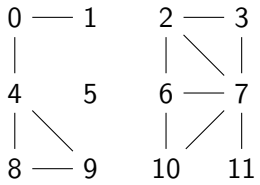


- As before, non-tree edges generate cycles



DFS tree

- Maintain a DFS counter, initially 0
- Increment counter each time we start and finish exploring a node
- Each vertex is assigned an entry number (pre) and exit number (post)

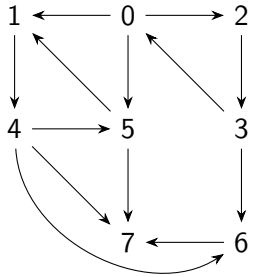


- As before, non-tree edges generate cycles
- To compute `pre` and `post` pass counter via recursive DFS calls

```
(visited,pre,post) = ({},{},{})

def DFSInitPrePost(AList):
  # Initialization
  for i in AList.keys():
    visited[i] = False
    pre[i],post[i]) = (-1,-1)
  return

def DFSPrePost(AList,v,count):
  visited[v] = True
  pre[v] = count
  count = count+1
  for k in AList[v]:
    if (not visited[k]):
      count = DFSPrePost(AList,k,count)
  post[v] = count
  count = count+1
  return(count)
```

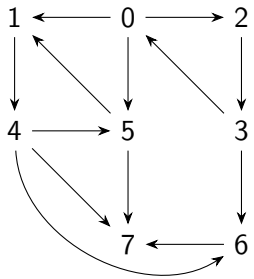
Directed cycles

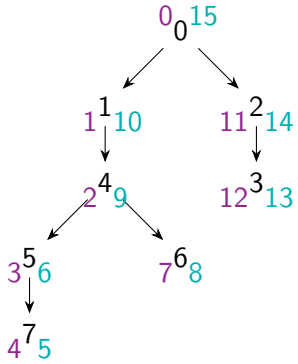
- In a directed graph, a cycle must follow same direction
  - $0 \rightarrow 2 \rightarrow 3 \rightarrow 0$ is a cycle
  - $0 \rightarrow 5 \rightarrow 1 \leftarrow 0$ is not



Directed cycles

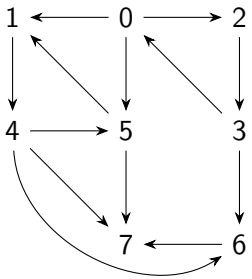
- In a directed graph, a cycle must follow same direction
  - $0 \to 2 \to 3 \to 0$ is a cycle
  - $0 \to 5 \to 1 \leftarrow 0$ is not

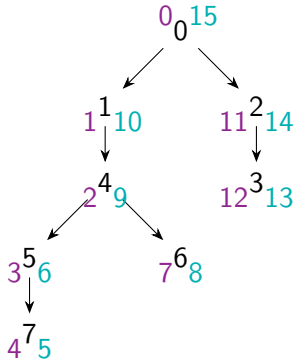




Directed cycles

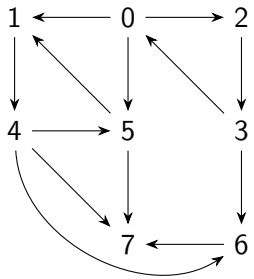
- In a directed graph, a cycle must follow same direction
  - $0 \to 2 \to 3 \to 0$ is a cycle
  - $0 \to 5 \to 1 \leftarrow 0$ is not
- Tree edges

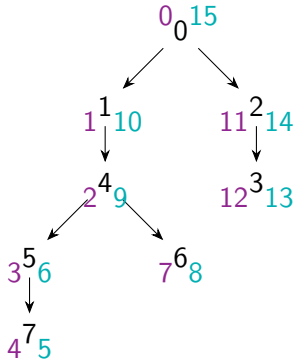




Directed cycles

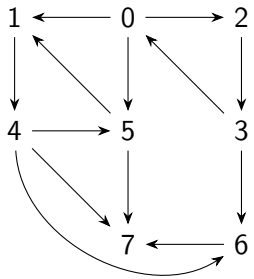
- In a directed graph, a cycle must follow same direction
  - $0 \to 2 \to 3 \to 0$ is a cycle
  - $0 \to 5 \to 1 \leftarrow 0$ is not
- Tree edges
- Different types of non-tree edges

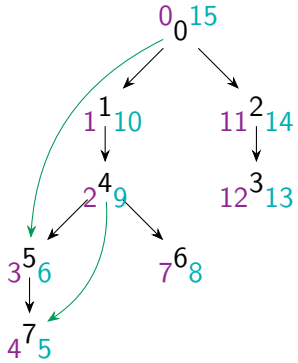




Directed cycles

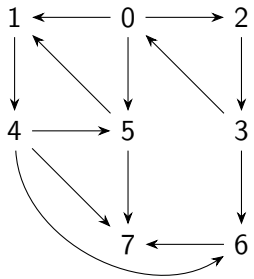
- In a directed graph, a cycle must follow same direction
  - $0 \to 2 \to 3 \to 0$ is a cycle
  - $0 \to 5 \to 1 \leftarrow 0$ is not
- Tree edges
- Different types of non-tree edges
  - Forward edges

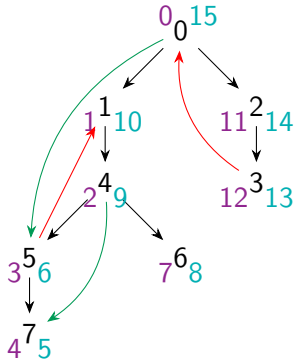




Directed cycles

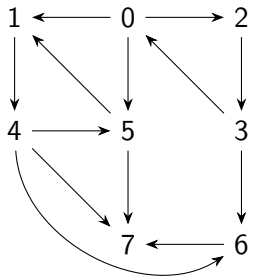
- In a directed graph, a cycle must follow same direction
  - $0 \to 2 \to 3 \to 0$ is a cycle
  - $0 \to 5 \to 1 \leftarrow 0$ is not
- Tree edges
- Different types of non-tree edges
  - Forward edges
  - Back edges

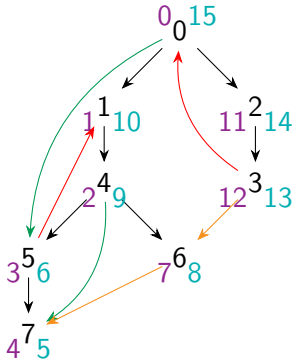




Directed cycles

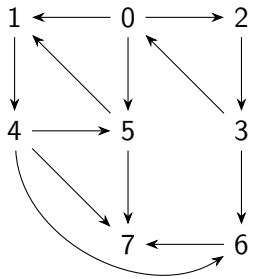
- In a directed graph, a cycle must follow same direction
  - $0 \to 2 \to 3 \to 0$ is a cycle
  - $0 \to 5 \to 1 \leftarrow 0$ is not
- Tree edges
- Different types of non-tree edges
  - Forward edges
  - Back edges
  - Cross edges



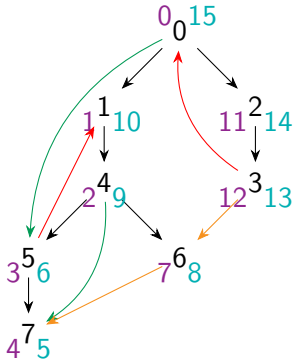


Directed cycles

- In a directed graph, a cycle must follow same direction
  - $0 \to 2 \to 3 \to 0$ is a cycle
  - $0 \to 5 \to 1 \leftarrow 0$ is not
- Tree edges
- Different types of non-tree edges
  - Forward edges
  - Back edges
  - Cross edges

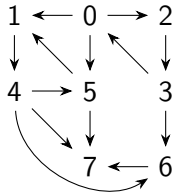


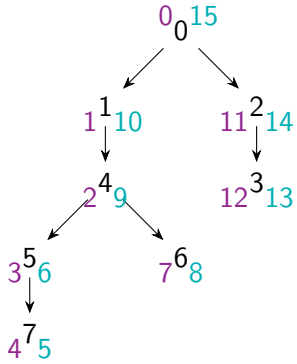
- Only back edges correspond to cycles



Classifying non-tree edges in directed graphs

- Use pre/post numbers

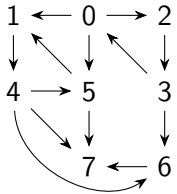


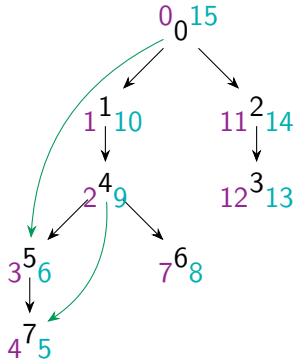


Classifying non-tree edges in directed graphs

- Use pre/post numbers
- Tree edge/forward edge (u, v)

  Interval $[\text{pre}(u), \text{post}(u)]$ contains $[\text{pre}(v), \text{post}(v)]$



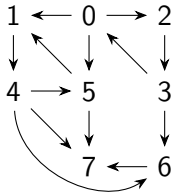


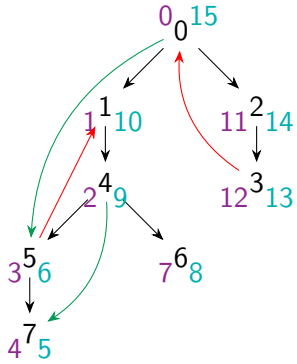
Classifying non-tree edges in directed graphs

- Use pre/post numbers
- Tree edge/forward edge (u, v)

  Interval $[\text{pre}(u), \text{post}(u)]$ contains $[\text{pre}(v), \text{post}(v)]$
- Back edge (u, v)

  Interval $[\text{pre}(v), \text{post}(v)]$ contains $[\text{pre}(u), \text{post}(u)]$





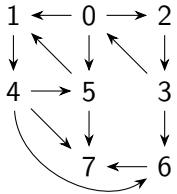
Classifying non-tree edges in directed graphs

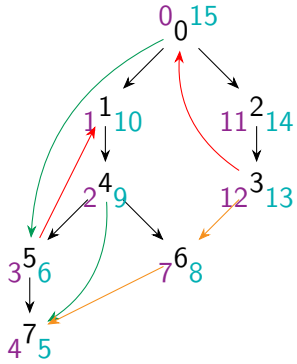
- Use pre/post numbers
- Tree edge/forward edge (u, v)

  Interval $[\text{pre}(u), \text{post}(u)]$ contains $[\text{pre}(v), \text{post}(v)]$
- Back edge (u, v)

  Interval $[\text{pre}(v), \text{post}(v)]$ contains $[\text{pre}(u), \text{post}(u)]$
- Cross edge (u, v)

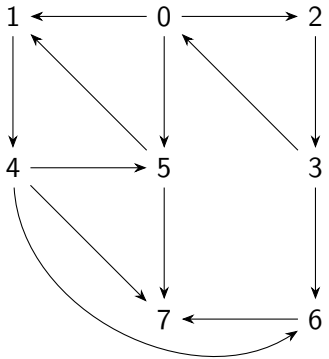
  Intervals $[\text{pre}(u), \text{post}(u)]$ and $[\text{pre}(v), \text{post}(v)]$ are disjoint





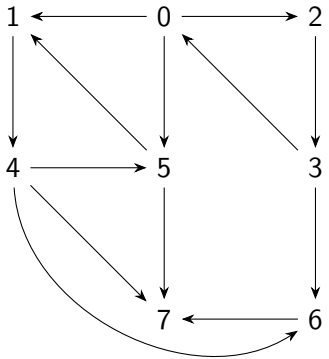
Connectivity in directed graphs

- Take directions into account



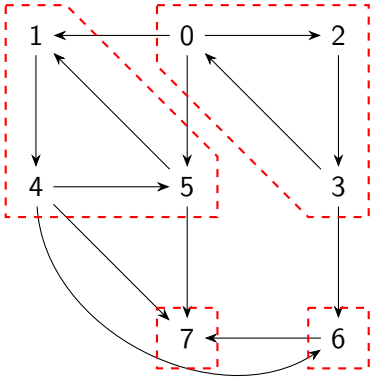
Connectivity in directed graphs

- Take directions into account
- Vertices i and j are strongly connected if there is a path from i to j and a path from j to i



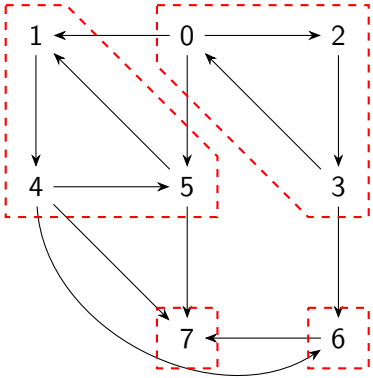
Connectivity in directed graphs

- Take directions into account
- Vertices i and j are strongly connected if there is a path from i to j and a path from j to i
- Directed graphs can be decomposed into strongly connected components (SCCs)
  - Within an SCC, each pair of vertices is strongly connected



Connectivity in directed graphs

- Take directions into account
- Vertices i and j are strongly connected if there is a path from i to j and a path from j to i
- Directed graphs can be decomposed into strongly connected components (SCCs)
  - Within an SCC, each pair of vertices is strongly connected
- DFS numbering can be used to compute SCCs



Summary

- BFS and DFS can be used to identify connected components in an undirected graph
  - BFS and DFS identify an underlying tree, non-tree edges generate cycles

Summary

- BFS and DFS can be used to identify connected components in an undirected graph
  - BFS and DFS identify an underlying tree, non-tree edges generate cycles
- In a directed graph, non-tree edges can be forward / back / cross
  - Only back edges generate cycles
  - Classify non-tree edges using DFS numbering

Summary

- BFS and DFS can be used to identify connected components in an undirected graph
  - BFS and DFS identify an underlying tree, non-tree edges generate cycles
- In a directed graph, non-tree edges can be forward / back / cross
  - Only back edges generate cycles
  - Classify non-tree edges using DFS numbering
- Directed graphs decompose into strongly connected components
  - DFS numbering can be used to compute SCC decomposition

Summary

- BFS and DFS can be used to identify connected components in an undirected graph
  - BFS and DFS identify an underlying tree, non-tree edges generate cycles
- In a directed graph, non-tree edges can be forward / back / cross
  - Only back edges generate cycles
  - Classify non-tree edges using DFS numbering
- Directed graphs decompose into strongly connected components
  - DFS numbering can be used to compute SCC decomposition
- DFS numbering can also be used to identify other features such as articulation points (cut vertices) and bridges (cut edges)

Summary

- BFS and DFS can be used to identify connected components in an undirected graph
  - BFS and DFS identify an underlying tree, non-tree edges generate cycles
- In a directed graph, non-tree edges can be forward / back / cross
  - Only back edges generate cycles
  - Classify non-tree edges using DFS numbering
- Directed graphs decompose into strongly connected components
  - DFS numbering can be used to compute SCC decomposition
- DFS numbering can also be used to identify other features such as articulation points (cut vertices) and bridges (cut edges)
- Directed acyclic graphs are useful for representing dependencies
  - Given course prerequisites, find a valid sequence to complete a programme