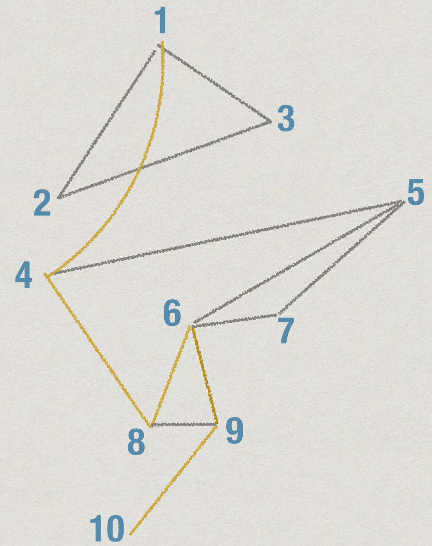
Graphs, formally

G = (V,E)

- Set of vertices V
- Set of edges E
  - E is a subset of pairs (v,v'): $E \subseteq V \times V$
  - Undirected graph: (v,v') and (v',v) are the same edge
  - Directed graph:
    - (v,v') is an edge from v to v'
    - Does not guarantee that (v',v) is also an edge

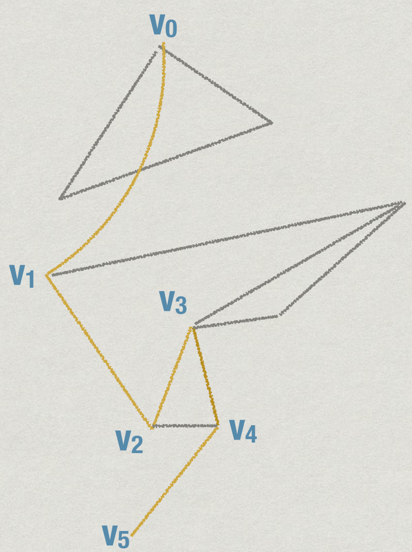
Adjacency matrix



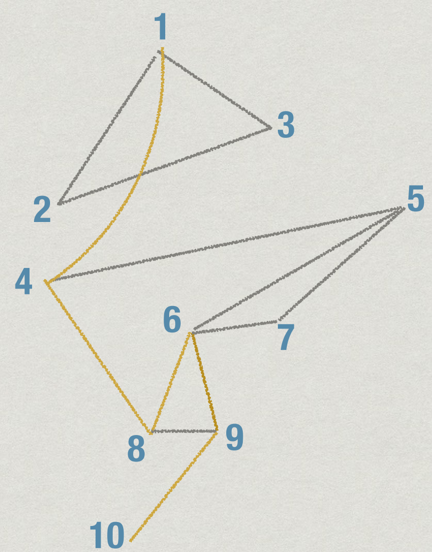
	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	0	0	0	0	0	0
2	1	0	1	0	0	0	0	0	0	0
3	1	1	0	0	0	0	0	0	0	0
4	1	0	0	0	1	0	0	1	0	0
5	0	0	0	1	0	1	1	0	0	0
6	0	0	0	0	1	0	1	1	1	0
7	0	0	0	0	1	1	0	0	0	0
8	0	0	0	1	0	1	0	0	1	0
9	0	0	0	0	0	1	0	1	0	1
10	0	0	0	0	0	0	0	0	1	0

Finding a route

- Find a sequence of vertices v_0, v_1, …, v_k such that
  - v_0 is **source**
  - Each (v_i,v_{i+1}) is an edge in E
  - v_k is **target**



Adjacency list



- For each vertex, maintain a list of its neighbours

1	2,3,4
2	1,3
3	1,2
4	1,5,8
5	4,6,7
6	5,7,8,9
7	5,6
8	4,6,9
9	6,8,10
10	9

Finding a path

- Mark vertices that have been visited
- Keep track of vertices whose neighbours have already been explored
  - Avoid going round indefinitely in circles
- Two fundamental strategies: breadth first and depth first

Breadth first search

- Recall that V = {1,2,…,n}
- Array visited[i] records whether i has been visited
- When a vertex is visited for the first time, add it to a **queue**
  - Explore vertices in the order they reach the queue

Breadth first search

- Explore the graph level by level
  - First visit vertices one step away
  - Then two steps away
  - …
- Remember which vertices have been **visited**
- Also keep track of vertices visited, but whose neighbours are yet to be **explored**

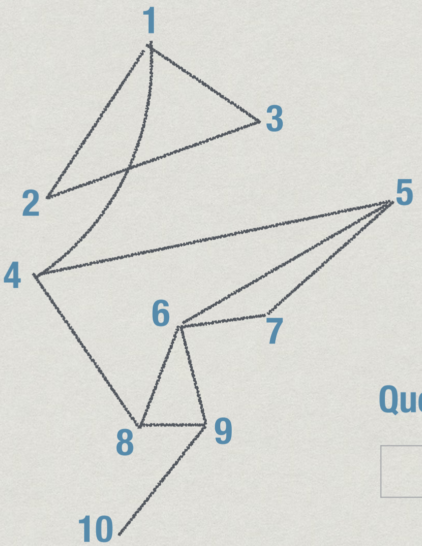
Breadth first search

- Exploring a vertex i:

```
for each edge (i,j)
  if visited[j] == 0
    visited[j] = 1
    append j to queue
```

- Initially, queue contains only source vertex
- At each stage, explore vertex at the head of the queue
- Stop when the queue becomes empty

Breadth first search

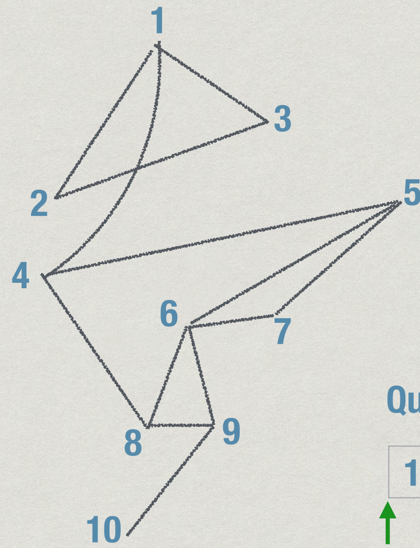


Visited

1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Queue

Breadth first search



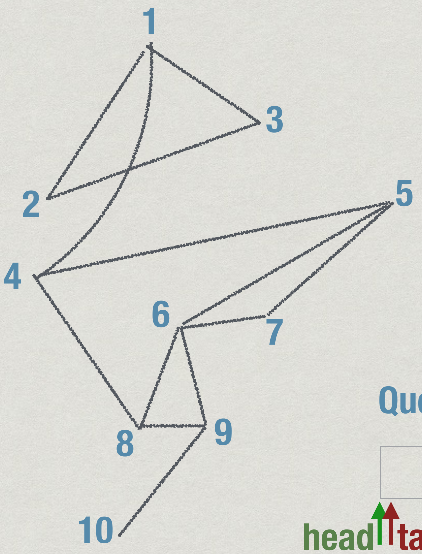
Visited

1	1
2	
3	
4	
5	
6	
7	
8	
9	
10	

Queue

1									

Breadth first search



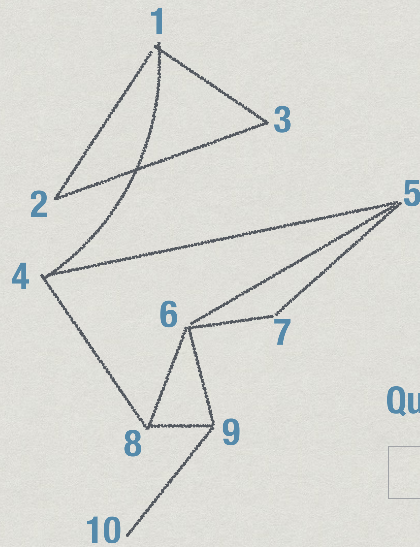
Visited

1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Queue

head tail

Breadth first search

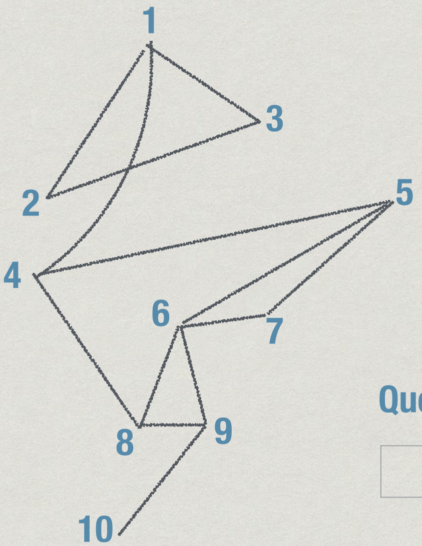


Visited

1	1
2	
3	
4	
5	
6	
7	
8	
9	
10	

Queue

Breadth first search



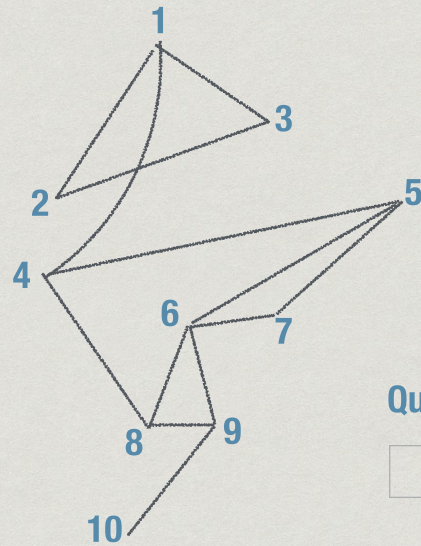
Visited

1	1
2	1
3	
4	
5	
6	
7	
8	
9	
10	

Queue

	2								

Breadth first search



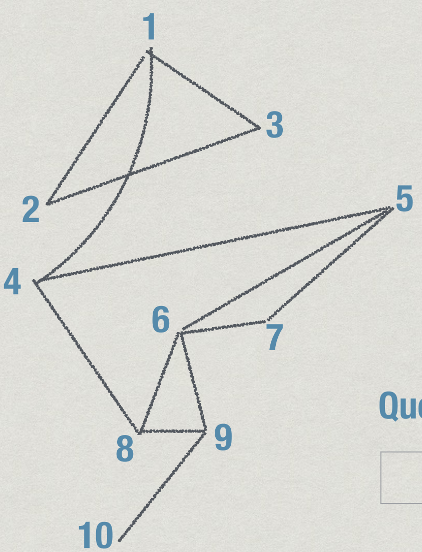
Visited

1	1
2	1
3	1
4	1
5	
6	
7	
8	
9	
10	

Queue

	2	3	4						

Breadth first search



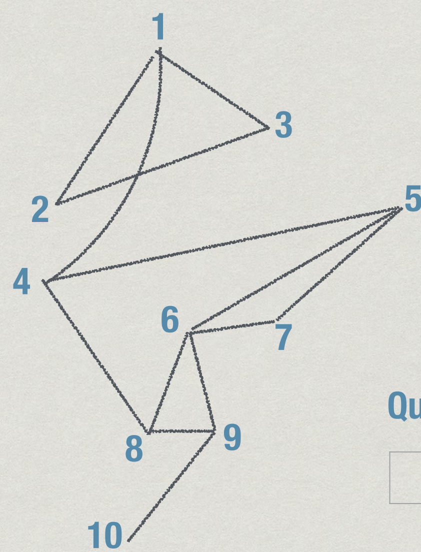
Visited

1	1
2	1
3	1
4	
5	
6	
7	
8	
9	
10	

Queue

	2	3							

Breadth first search



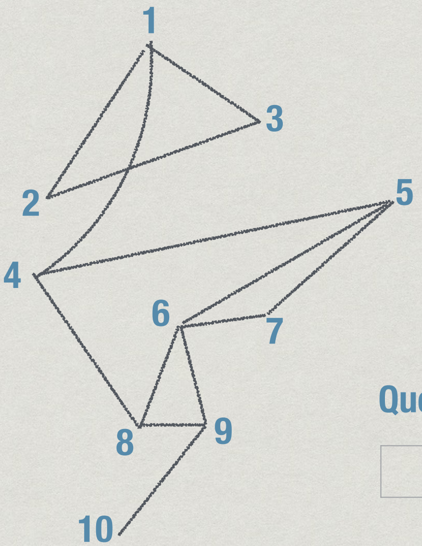
Visited

1	1
2	1
3	1
4	1
5	
6	
7	
8	
9	
10	

Queue

		3	4						

Breadth first search



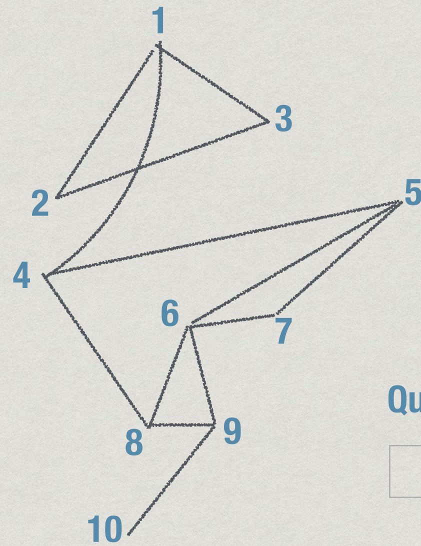
Visited

1	1
2	1
3	1
4	1
5	
6	
7	
8	
9	
10	

Queue

			4						

Breadth first search



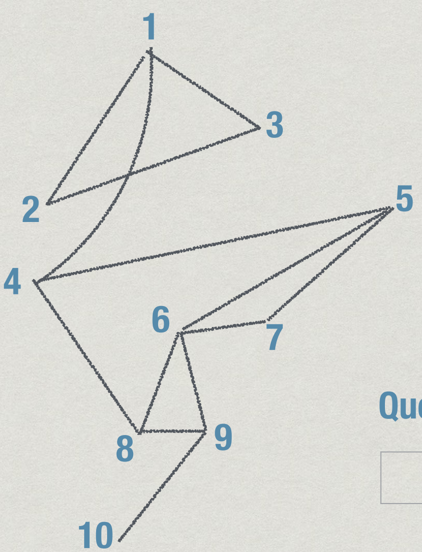
Visited

1	1
2	1
3	1
4	1
5	1
6	
7	
8	
9	
10	

Queue

				5					

Breadth first search

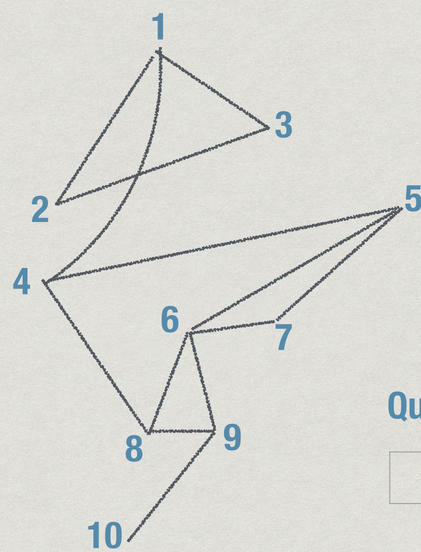


Visited

1	1
2	1
3	1
4	1
5	
6	
7	
8	
9	
10	

Queue

Breadth first search



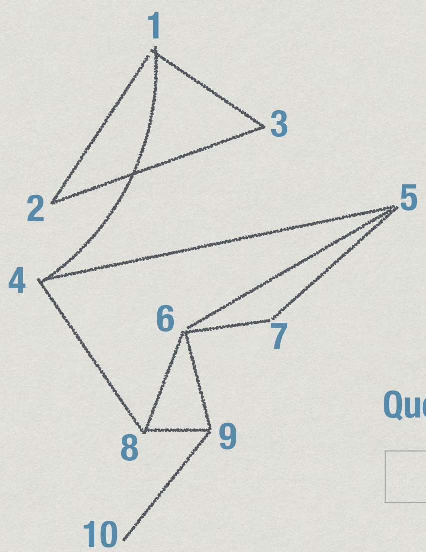
Visited

1	1
2	1
3	1
4	1
5	1
6	
7	
8	1
9	
10	

Queue

				5	8				

Breadth first search



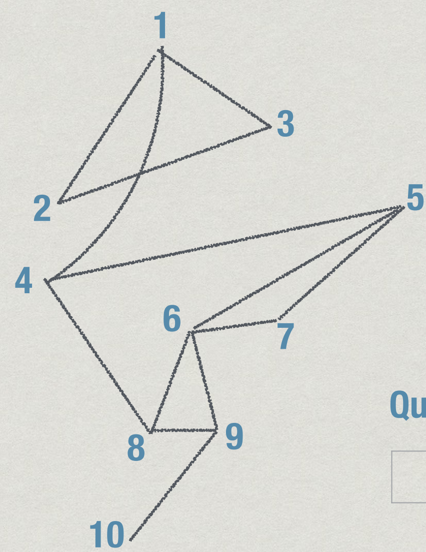
Visited

1	1
2	1
3	1
4	1
5	1
6	
7	
8	1
9	
10	

Queue

					8				

Breadth first search



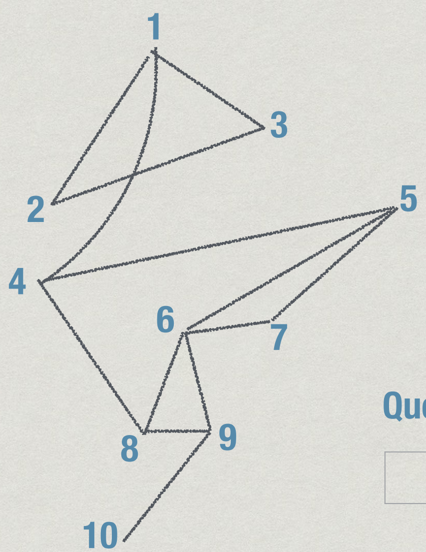
Visited

1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	
10	

Queue

					8	6	7		

Breadth first search



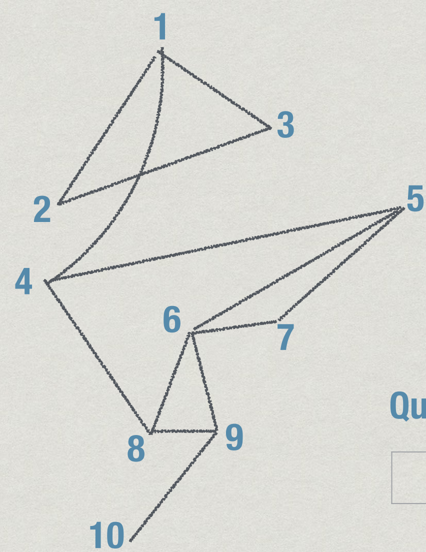
Visited

1	1
2	1
3	1
4	1
5	1
6	1
7	
8	1
9	
10	

Queue

					8	6			

Breadth first search



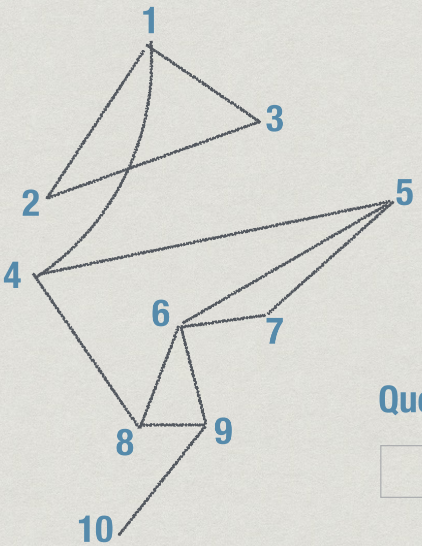
Visited

1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	
10	

Queue

						6	7		

Breadth first search



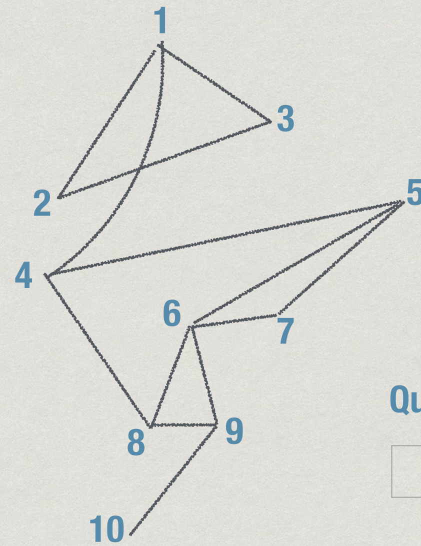
Visited

1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	

Queue

						6	7	9	

Breadth first search



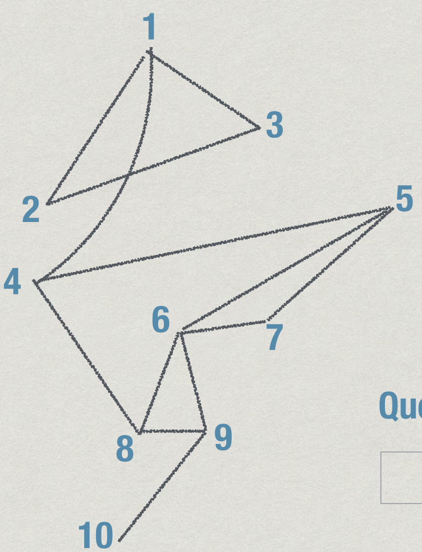
Visited

1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	

Queue

								9	

Breadth first search



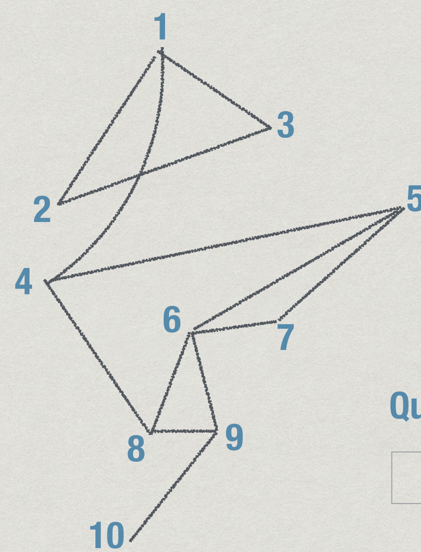
Visited

1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	

Queue

							7	9	

Breadth first search

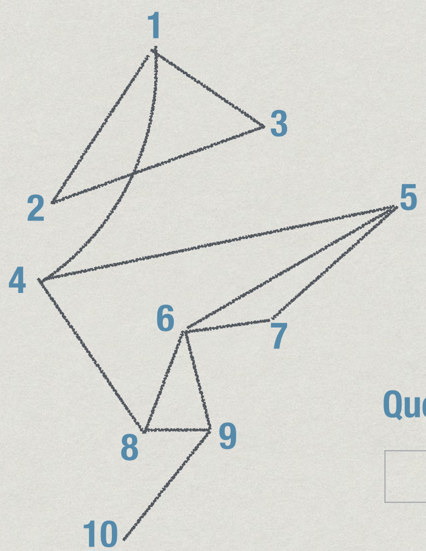


Visited

1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	

Queue

Breadth first search



Visited

1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1

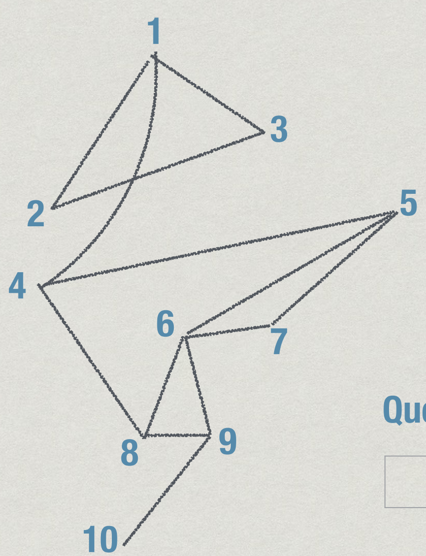
Queue

									10

Breadth first search

```
function BFS(i) // BFS starting from vertex i

  //Initialization
  for j = 1..n {visited[j] = 0}; Q = []

  //Start the exploration at i
  visited[i] = 1; append(Q,i)

  //Explore each vertex in Q
  while Q is not empty
     j = extract_head(Q)
     for each (j,k) in E
        if visited[k] == 0
           visited[k] = 1; append(Q,k)
```

Breadth first search



Visited

1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1

Queue

Complexity of BFS

- Each vertex enters Q exactly once
- If graph is connected, loop to process Q iterated n times
  - For each j extracted from Q, need to examine all neighbours of j
  - In adjacency matrix, scan row j: n entries
- Hence, overall $O(n^2)$

Complexity of BFS

- Let m be the number of edges in E. What if $m << n^2$?
- Adjacency list: scanning neighbours of j takes time proportional to number of neighbours (**degree** of j)
- Across the loop, each edge (i,j) is scanned twice, once when exploring i and again when exploring j
  - Overall, exploring neighbours takes time O(m)
- Marking n vertices visited still takes O(n)
- Overall, O(n+m)

Enhancements to BFS

- If BFS(i) sets visited[j] = 1, we know that i and j are connected
- How do we identify a path from i to j
- When we mark visited[k] = 1, remember the neighbour from which we marked it
  - If exploring edge (j,k) visits k, set parent[k] = j

Complexity of BFS

- For graphs, O(m+n) is considered the best possible
  - Need to see each edge and vertex at least once
- O(m+n) is considered to be **linear** in the size of the graph

Breadth first search

```
function BFS(i) // BFS starting from vertex i

  //Initialization
  for j = 1..n {visited[j] = 0; parent[j] = -1}
  Q = []

  //Start the exploration at i
  visited[i] = 1; append(Q,i)

  //Explore each vertex in Q
  while Q is not empty
    j = extract_head(Q)
    for each (j,k) in E
      if visited[k] == 0
        visited[k] = 1; parent[k] = j; append(Q,k);
```

Reconstructing the path

- BFS(i) sets visited[j] = 1
- visited[j] = 1, so parent[j] = j' for some j'
- visited[j'] = 1, so parent[j'] = j" for some j"
- ...
- Eventually, trace back path to k with parent[k] = i

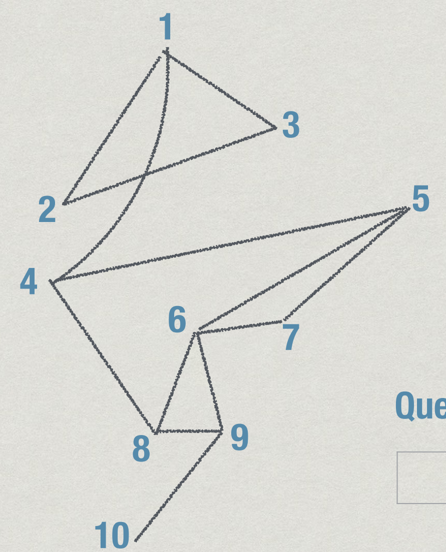
Breadth first search

```
function BFS(i) // BFS starting from vertex i

   //Initialization
   for j = 1..n {level[j] = -1; parent[j] = -1}
   Q = []

   //Start the exploration at i, level[i] set to 0
   level[i] = 0; append(Q,i)

   //Explore each vertex in Q, increment level for each new vertex
   while Q is not empty
      j = extract_head(Q)
      for each (j,k) in E
         if level[k] == -1
            level[k] = 1+level[j]; parent[k] = j;
            append(Q,k);
```

Recording distances

- BFS can record how long the path is to each vertex
- Instead of binary array visited[ ], keep integer array level[ ]
- level[j] = -1 initially
- level[j] = p means j is reached in p steps from i

Breadth first search

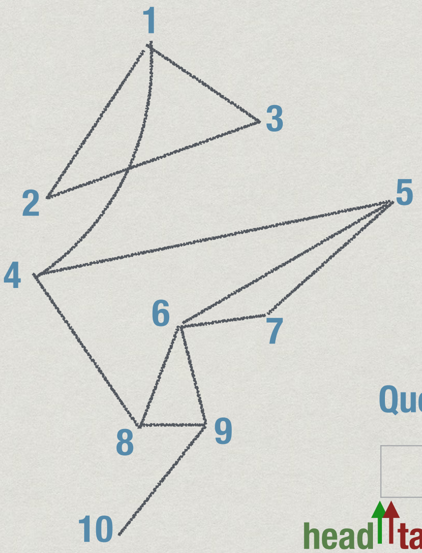


L : Level
P : Parent

	L	P
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		

Queue

Breadth first search



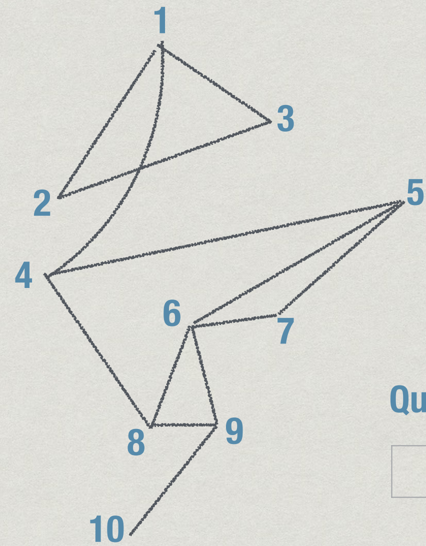
L : Level
P : Parent

	L	P
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		

Queue

head tail

Breadth first search

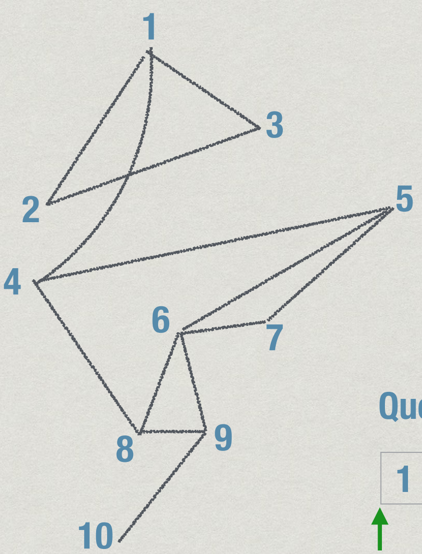


L : Level
P : Parent

	L	P
1	0	-
2		
3		
4		
5		
6		
7		
8		
9		
10		

Queue

Breadth first search



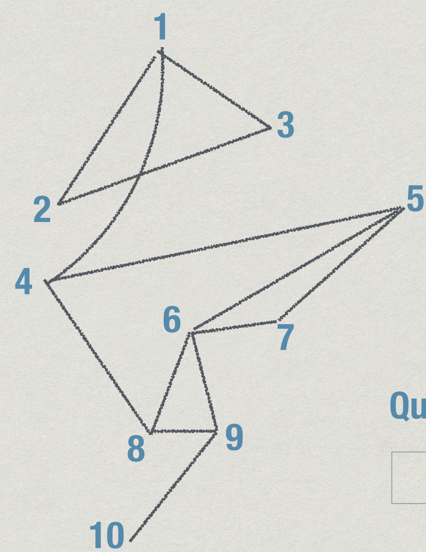
L : Level
P : Parent

	L	P
1	0	-
2		
3		
4		
5		
6		
7		
8		
9		
10		

Queue

1									

Breadth first search



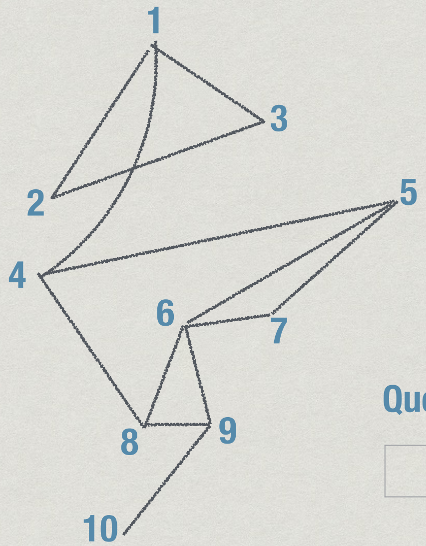
L : Level
P : Parent

	L	P
1	0	-
2	1	1
3		
4		
5		
6		
7		
8		
9		
10		

Queue

	2								

Breadth first search



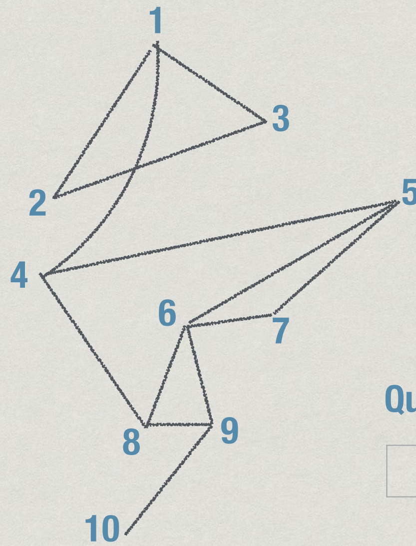
L : Level
P : Parent

	L	P
1	0	-
2	1	1
3	1	1
4		
5		
6		
7		
8		
9		
10		

Queue

	2	3							

Breadth first search



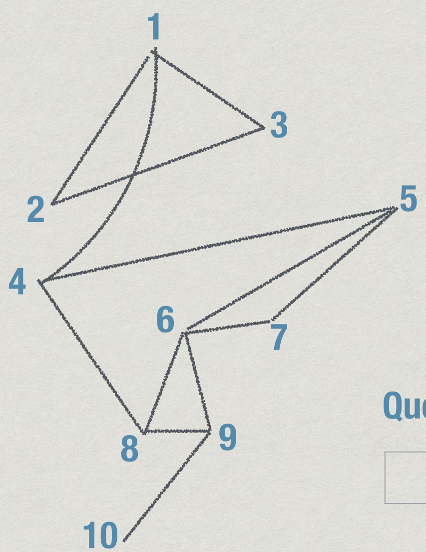
L : Level
P : Parent

	L	P
1	0	-
2	1	1
3	1	1
4	1	1
5		
6		
7		
8		
9		
10		

Queue

		3	4						

Breadth first search



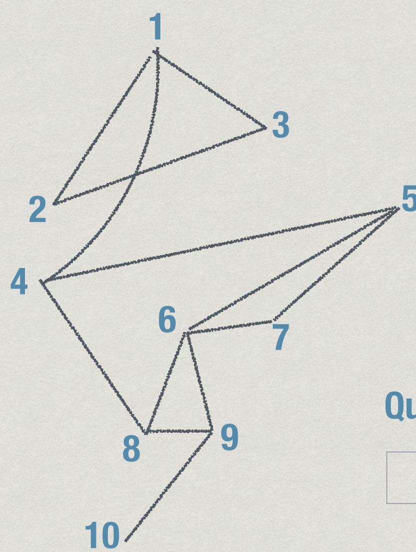
L : Level
P : Parent

	L	P
1	0	-
2	1	1
3	1	1
4	1	1
5		
6		
7		
8		
9		
10		

Queue

	2	3	4						

Breadth first search



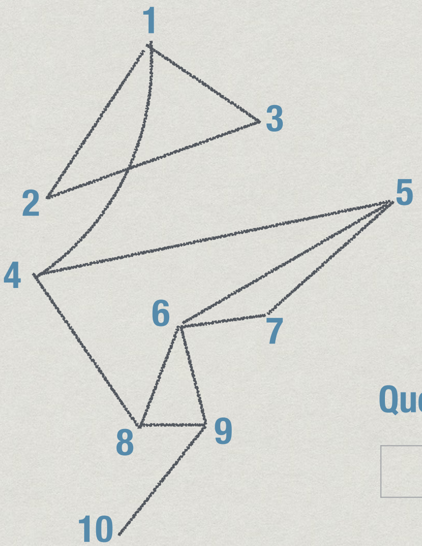
L : Level
P : Parent

	L	P
1	0	-
2	1	1
3	1	1
4	1	1
5		
6		
7		
8		
9		
10		

Queue

			4						

Breadth first search

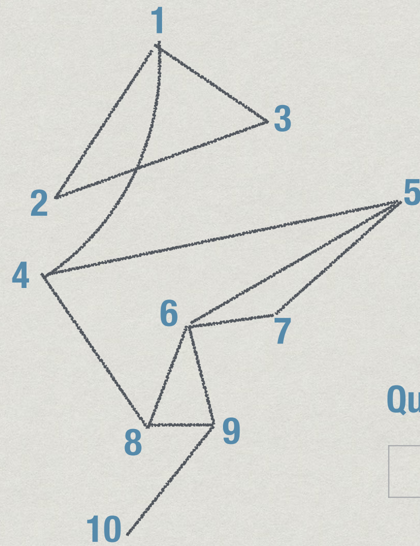


L : Level
P : Parent

	L	P
1	0	-
2	1	1
3	1	1
4	1	1
5		
6		
7		
8		
9		
10		

Queue

Breadth first search



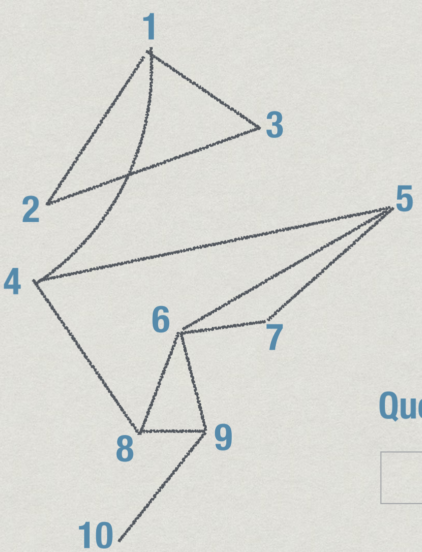
L : Level
P : Parent

	L	P
1	0	-
2	1	1
3	1	1
4	1	1
5	2	4
6		
7		
8	2	4
9		
10		

Queue

				5	8				

Breadth first search



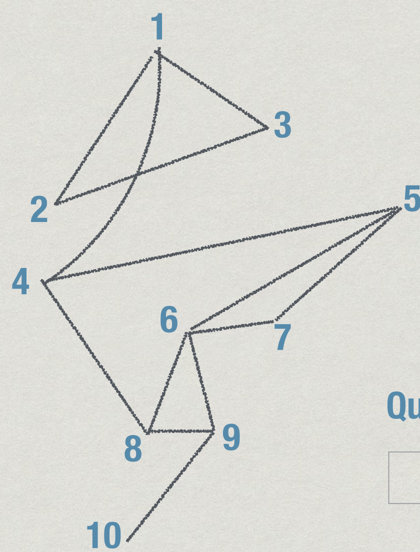
L : Level
P : Parent

	L	P
1	0	-
2	1	1
3	1	1
4	1	1
5	2	4
6		
7		
8		
9		
10		

Queue

				5					

Breadth first search



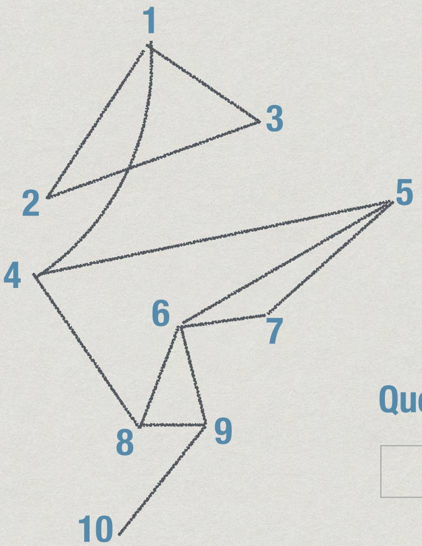
L : Level
P : Parent

	L	P
1	0	-
2	1	1
3	1	1
4	1	1
5	2	4
6		
7		
8	2	4
9		
10		

Queue

					8				

Breadth first search



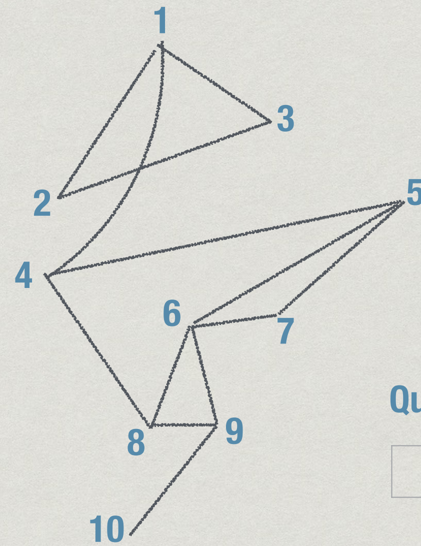
L : Level
P : Parent

	L	P
1	0	-
2	1	1
3	1	1
4	1	1
5	2	4
6	3	5
7		
8	2	4
9		
10		

Queue

					8	6			

Breadth first search



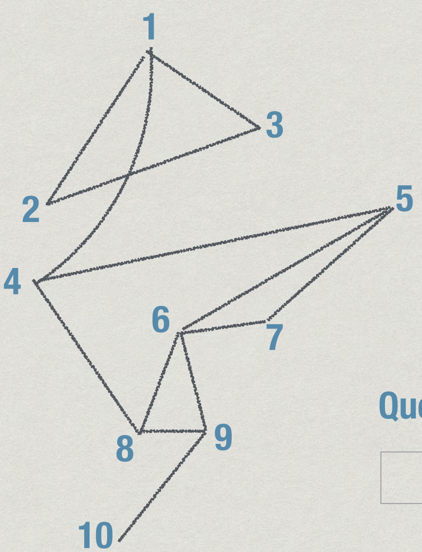
L : Level
P : Parent

	L	P
1	0	-
2	1	1
3	1	1
4	1	1
5	2	4
6	3	5
7	3	5
8	2	4
9		
10		

Queue

						6	7		

Breadth first search



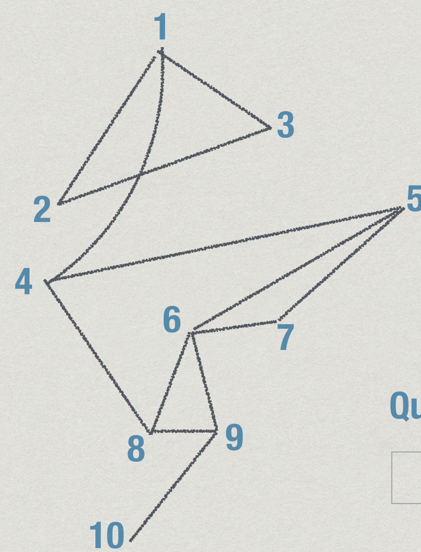
L : Level
P : Parent

	L	P
1	0	-
2	1	1
3	1	1
4	1	1
5	2	4
6	3	5
7	3	5
8	2	4
9		
10		

Queue

					8	6	7		

Breadth first search



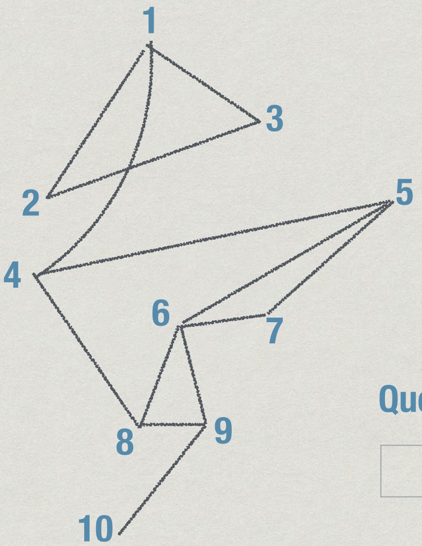
L : Level
P : Parent

	L	P
1	0	-
2	1	1
3	1	1
4	1	1
5	2	4
6	3	5
7	3	5
8	2	4
9	3	8
10		

Queue

						6	7	9	

Breadth first search



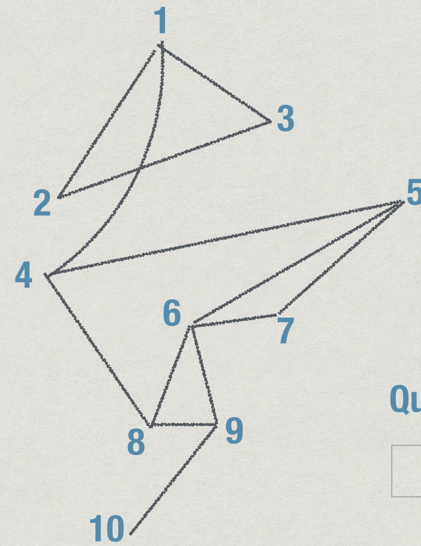
L : Level
P : Parent

	L	P
1	0	-
2	1	1
3	1	1
4	1	1
5	2	4
6	3	5
7	3	5
8	2	4
9	3	8
10		

Queue

							7	9	

Breadth first search

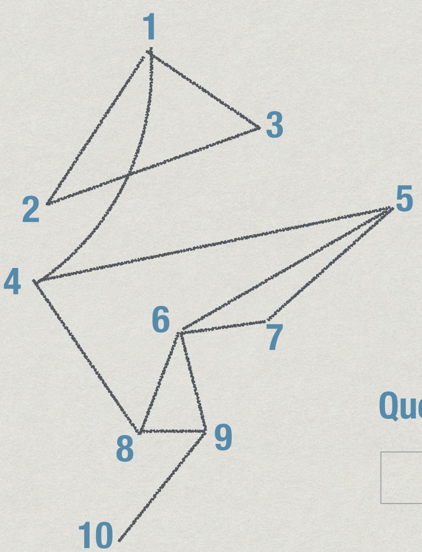


L : Level
P : Parent

	L	P
1	0	-
2	1	1
3	1	1
4	1	1
5	2	4
6	3	5
7	3	5
8	2	4
9	3	8
10		

Queue

Breadth first search



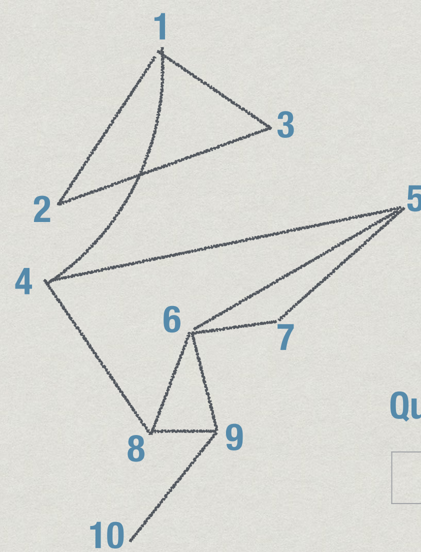
L : Level
P : Parent

	L	P
1	0	-
2	1	1
3	1	1
4	1	1
5	2	4
6	3	5
7	3	5
8	2	4
9	3	8
10		

Queue

								9	

Breadth first search



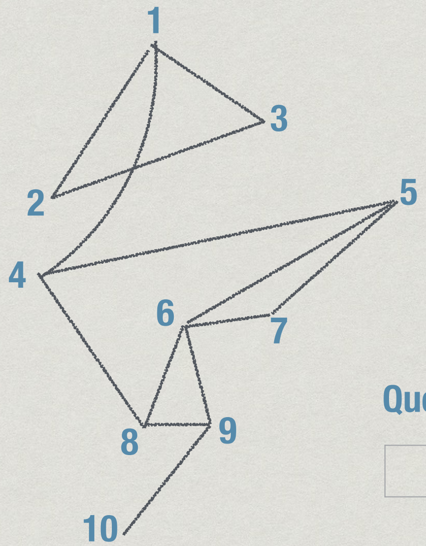
L : Level
P : Parent

	L	P
1	0	-
2	1	1
3	1	1
4	1	1
5	2	4
6	3	5
7	3	5
8	2	4
9	3	8
10	4	9

Queue

									10

Breadth first search



L : Level
P : Parent

	L	P
1	0	-
2	1	1
3	1	1
4	1	1
5	2	4
6	3	5
7	3	5
8	2	4
9	3	8
10	4	9

Queue

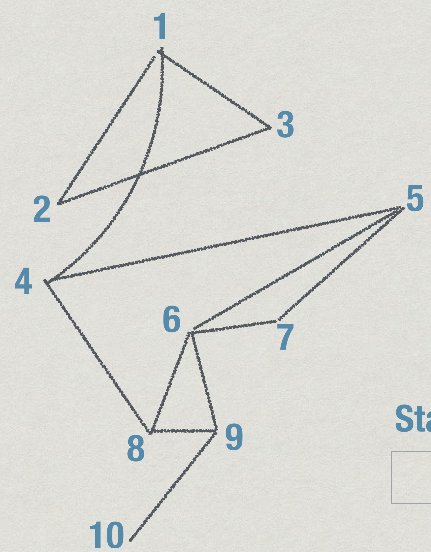
Depth first search

- Start from i, visit a neighbour j
- Suspend the exploration of i and explore j instead
- Continue till you reach a vertex with no unexplored neighbours
- Backtrack to nearest suspended vertex that still has an unexplored neighbour
- Suspended vertices are stored in a **stack**
  - Last in, first out: most recently suspended is checked first

Recording distances

- BFS with level[ ] gives us the shortest path to each node in terms of number of edges
- In general, edges are labelled by a cost (money, time, distance …)
  - Min cost path not same as fewest edges
- Will look at shortest paths in **weighted** graphs later
  - BFS computes shortest paths if all costs are 1

Depth first search



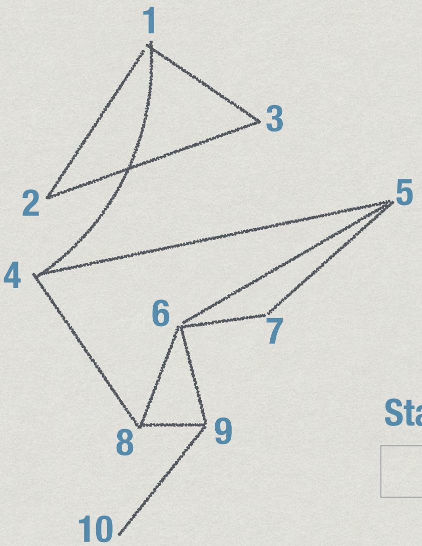
Visited

1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Stack of suspended vertices

Depth first search

Start at 4



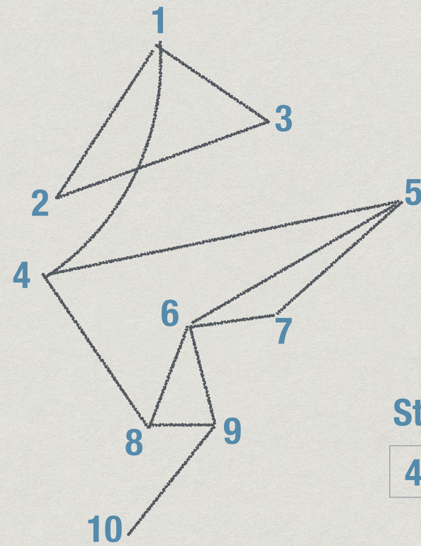
Visited

1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Stack of suspended vertices

Depth first search

Start at 4



Visited

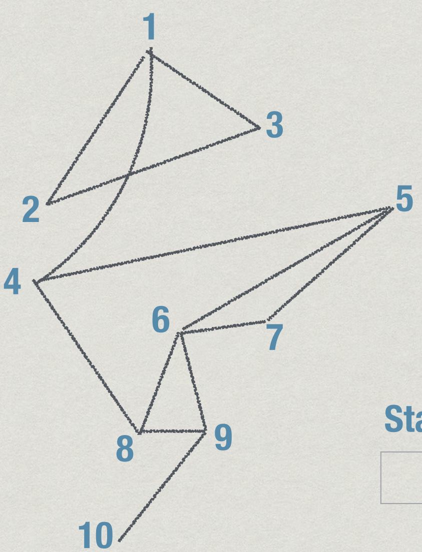
1	1
2	
3	
4	1
5	
6	
7	
8	
9	
10	

Stack of suspended vertices

4									

Depth first search

Start at 4



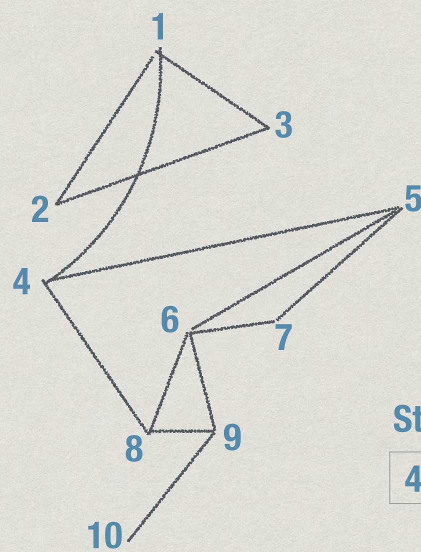
Visited

1	
2	
3	
4	1
5	
6	
7	
8	
9	
10	

Stack of suspended vertices

Depth first search

Start at 4



Visited

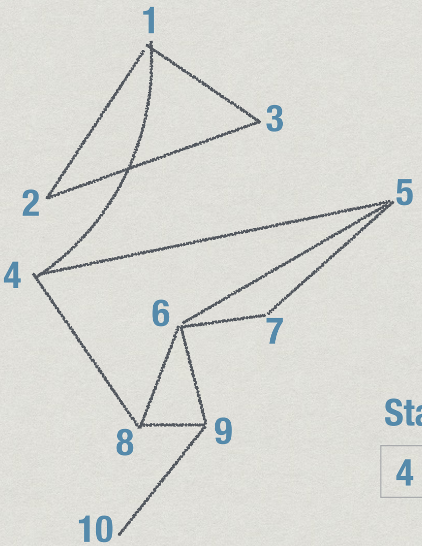
1	1
2	1
3	
4	1
5	
6	
7	
8	
9	
10	

Stack of suspended vertices

4	1								

Depth first search

Start at 4



Visited

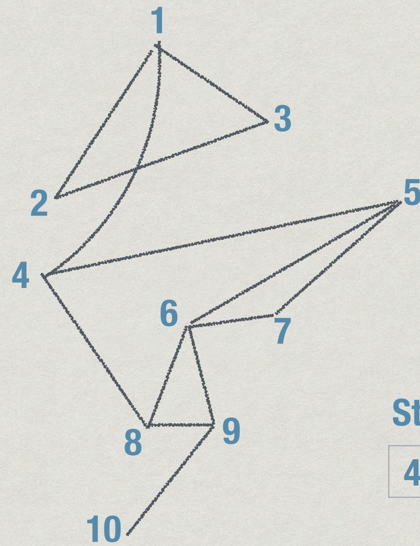
1	1
2	1
3	1
4	1
5	
6	
7	
8	
9	
10	

Stack of suspended vertices

4	1	2							

Depth first search

Start at 4



Visited

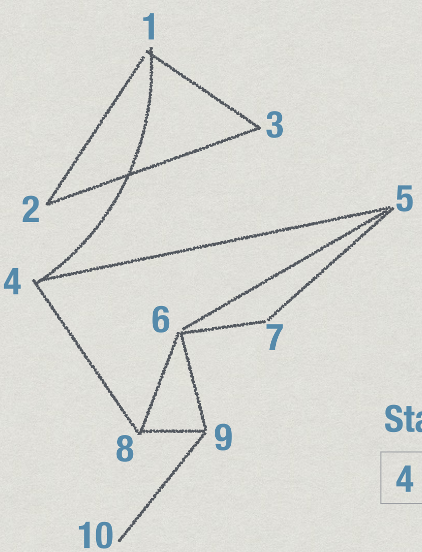
1	1
2	1
3	1
4	1
5	
6	
7	
8	
9	
10	

Stack of suspended vertices

4									

Depth first search

Start at 4



Visited

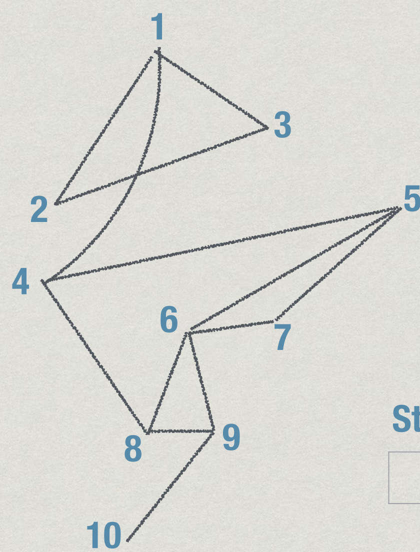
1	1
2	1
3	1
4	1
5	
6	
7	
8	
9	
10	

Stack of suspended vertices

4	1								

Depth first search

Start at 4



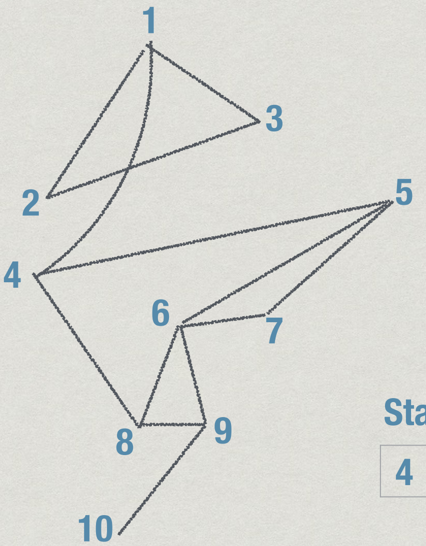
Visited

1	1
2	1
3	1
4	1
5	
6	
7	
8	
9	
10	

Stack of suspended vertices

Depth first search

Start at 4



Visited

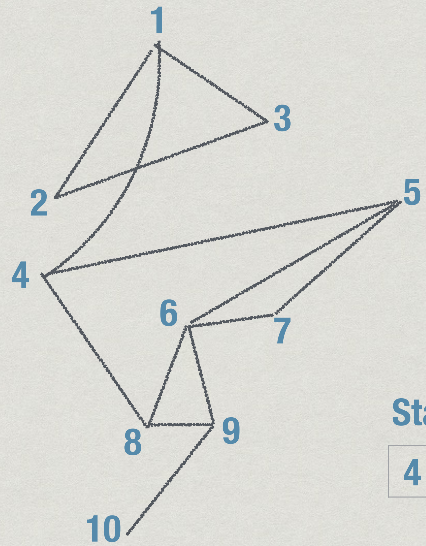
1	1
2	1
3	1
4	1
5	1
6	
7	
8	
9	
10	

Stack of suspended vertices

4									

Depth first search

Start at 4



Visited

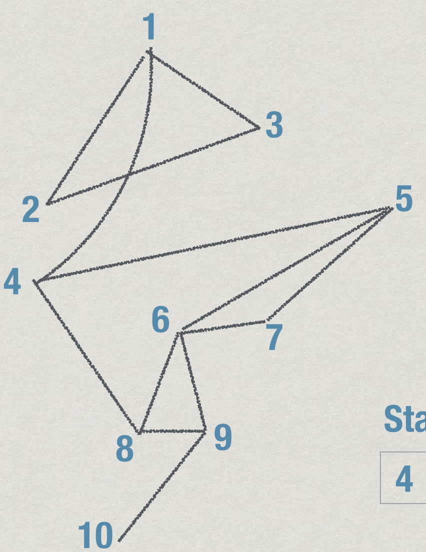
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	
9	
10	

Stack of suspended vertices

4	5	6							

Depth first search

Start at 4



Visited

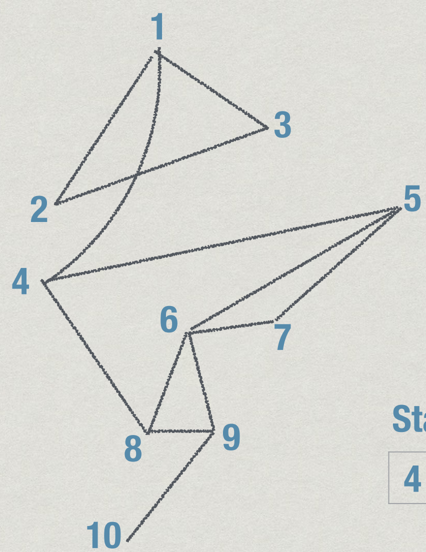
1	1
2	1
3	1
4	1
5	1
6	1
7	
8	
9	
10	

Stack of suspended vertices

4	5								

Depth first search

Start at 4



Visited

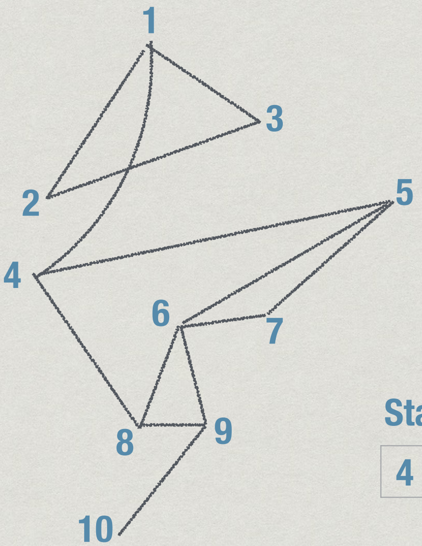
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	
9	
10	

Stack of suspended vertices

4	5								

Depth first search

Start at 4



Visited

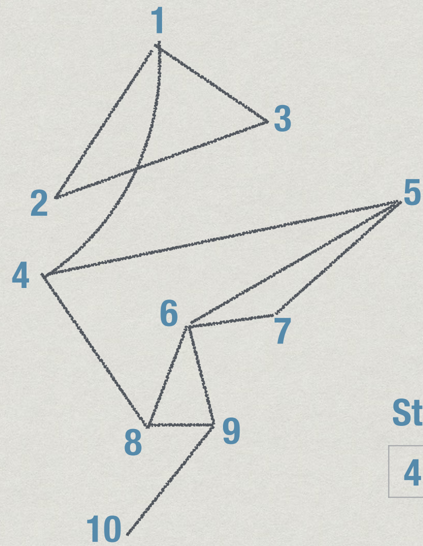
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	
10	

Stack of suspended vertices

4	5	6							

Depth first search

Start at 4



Visited

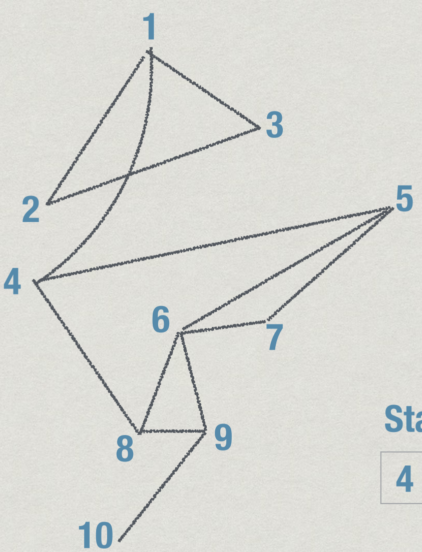
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1

Stack of suspended vertices

4	5	6	8	9					

Depth first search

Start at 4



Visited

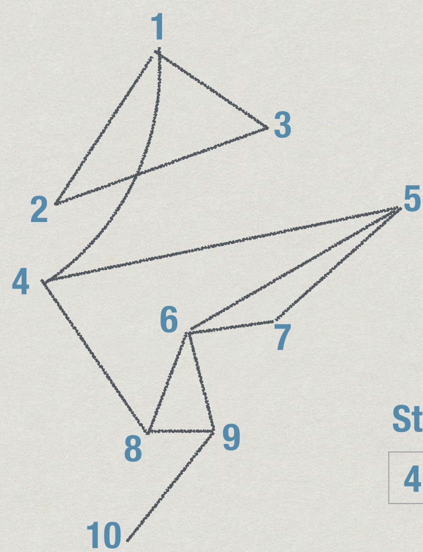
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	

Stack of suspended vertices

4	5	6	8						

Depth first search

Start at 4



Visited

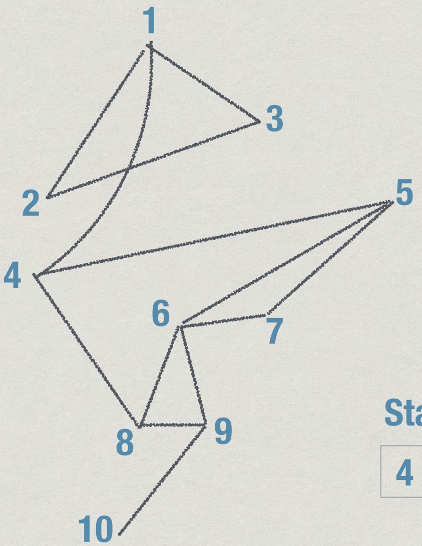
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1

Stack of suspended vertices

4	5	6	8						

Depth first search

Start at 4



Visited

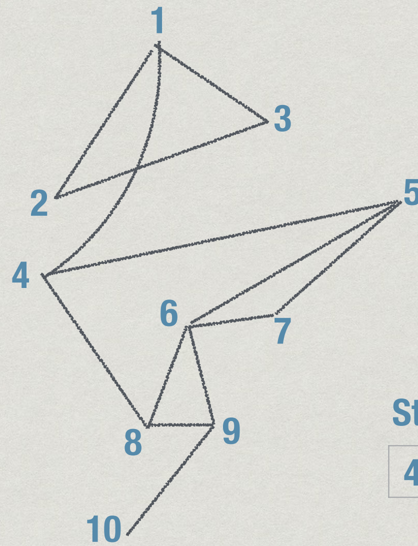
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1

Stack of suspended vertices

4	5	6							

Depth first search

Start at 4



Visited

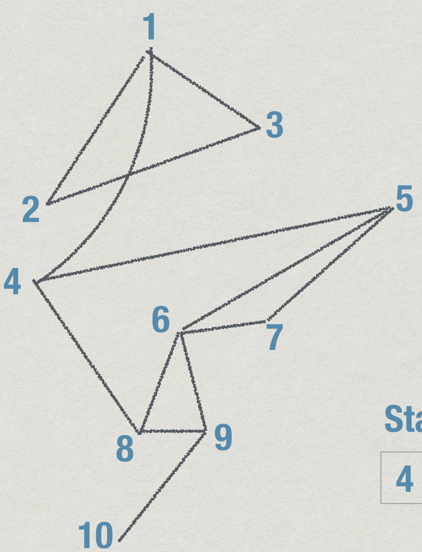
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1

Stack of suspended vertices

4									

Depth first search

Start at 4



Visited

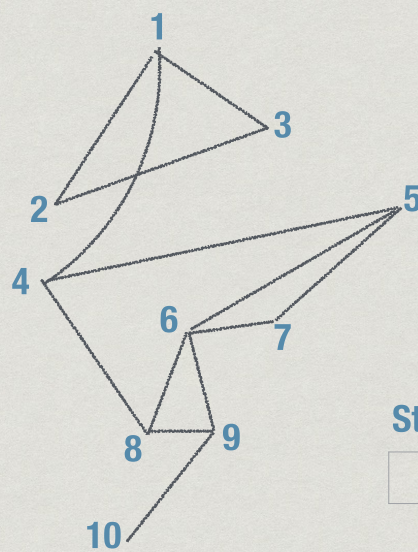
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1

Stack of suspended vertices

4	5								

Depth first search

Start at 4



Visited

1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1

Stack of suspended vertices

Depth first search

Depth first search

- DFS is most natural to implement recursively
  - For each unvisited neighbour j of i, call DFS(j)
- No need to explicitly maintain a stack
  - Stack is maintained implicitly by recursive calls

Depth first search

- DFS is most natural to implement recursively
  - For each unvisited neighbour j of i, call DFS(j)

Depth first search

```
//Initialization
   for j = 1..n {visited[j] = 0; parent[j] = -1}

function DFS(i) // DFS starting from vertex i

   //Mark i as visited
   visited[i] = 1

   //Explore each neighbour of i recursively
   for each (i,j) in E
      if visited[j] == 0
         parent[j] = i
         DFS(j)
```

Complexity of DFS

Complexity of DFS

- Each vertex marked and explored exactly once
- DFS(j) need to examine all neighbours of j

Complexity of DFS

- Each vertex marked and explored exactly once

Complexity of DFS

- Each vertex marked and explored exactly once
- DFS(j) need to examine all neighbours of j
- In adjacency matrix, scan row j: n entries
  - Overall $O(n^2)$

Complexity of DFS

- Each vertex marked and explored exactly once
- DFS(j) need to examine all neighbours of j
- In adjacency matrix, scan row j: n entries
  - Overall $O(n^2)$
- With adjacency list, scanning takes O(m) time across all vertices
  - Total time is O(m+n), like BFS

Properties of DFS

- Paths discovered by DFS are not shortest paths, unlike BFS

Properties of DFS

Properties of DFS

- Paths discovered by DFS are not shortest paths, unlike BFS
- Why use DFS at all?

Properties of DFS

- Paths discovered by DFS are not shortest paths, unlike BFS
- Why use DFS at all?
- Many useful features can be extracted from recording the order in which DFS visited vertices

Properties of DFS

- Paths discovered by DFS are not shortest paths, unlike BFS
- Why use DFS at all?
- Many useful features can be extracted from recording the order in which DFS visited vertices
  - **DFS numbering**
  - Maintain a counter

Properties of DFS

- Paths discovered by DFS are not shortest paths, unlike BFS
- Why use DFS at all?
- Many useful features can be extracted from recording the order in which DFS visited vertices
  - **DFS numbering**

Properties of DFS

- Paths discovered by DFS are not shortest paths, unlike BFS
- Why use DFS at all?
- Many useful features can be extracted from recording the order in which DFS visited vertices
  - **DFS numbering**
  - Maintain a counter
  - Increment and record counter value when entering and leaving a vertex.

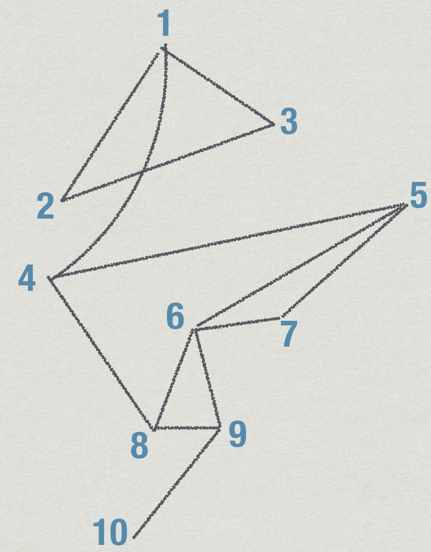
Depth first search

```
//Initialization
   for j = 1..n {visited[j] = 0; parent[j] = -1}
   count = 0

function DFS(i) // DFS starting from vertex i

   //Mark i as visited
   visited[i] = 1; pre[i] = count; count++

   //Explore each neighbours of i recursively
   for each (i,j) in E
      if visited[j] == 0
         parent[j] = i
         DFS(j)
         post[i] = count; count++
```

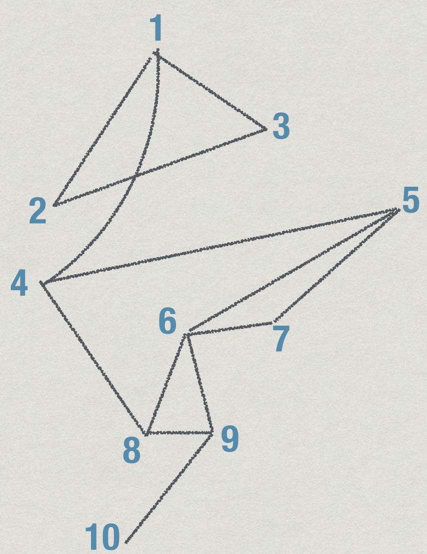
DFS numbering

pre[i] and post[i] can be used to find

- if the graph has a **cycle** — i.e., a loop
- **cut vertex** — removal disconnects the graph
- …



DFS numbering



Summary

- BFS and DFS are two systematic ways to explore a graph
  - Both take time linear in the size of the graph with adjacency lists
- Recover paths by keeping parent information
- BFS can compute shortest paths, in terms of number of edges
- DFS numbering can reveal many interesting features

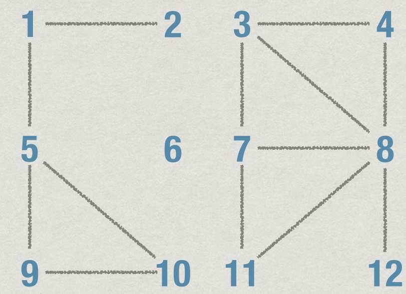
Graphs, formally

G = (V,E)

- Set of vertices V
- Set of edges E
  - E is a subset of pairs (v,v’): $E \subseteq V \times V$
  - Undirected graph: (v,v’) and (v’,v) are the same edge
  - Directed graph:
    - (v,v’) is an edge from v to v’
    - Does not guarantee that (v’,v) is also an edge

Connectivity

Connected graph



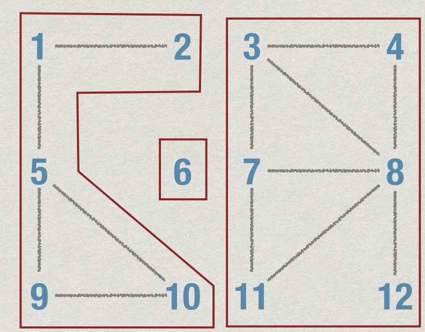
Disconnected graph

Exploring graph structure

- Breadth first search
  - Level by level exploration
- Depth first search
  - Explore each vertex as soon as it is visited
  - DFS numbering
- What can we find out about a graph using BFS/DFS?

Connectivity

Connected graph



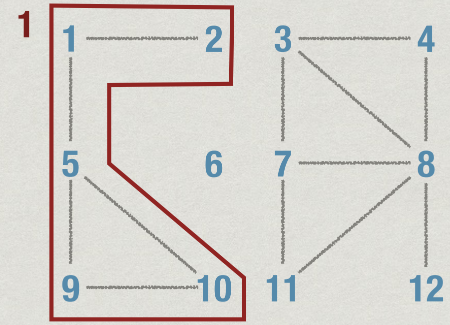
Disconnected graph

Connected components

Identifying connected components

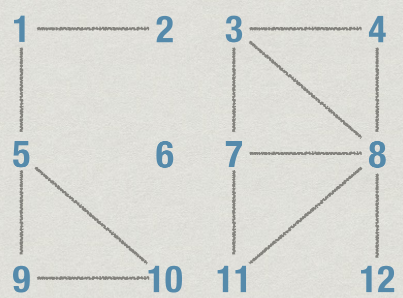
- Vertices {1,2,…,N}
- Start BFS or DFS from 1
  - All nodes marked Visited form a connected component
  - Pick first unvisited node, say j, and run BFS or DFS from j
  - Repeat till all nodes are visited
- Update BFS/DFS to label each visited node with component number

Connected components



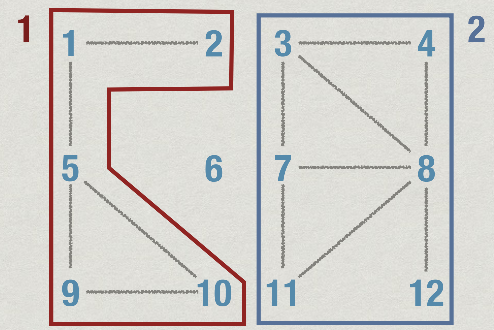
- Add a counter `comp` to number components
- Increment counter each time a fresh BFS/DFS starts
- Label each visited node `j` with `component[j] = comp`

Connected components



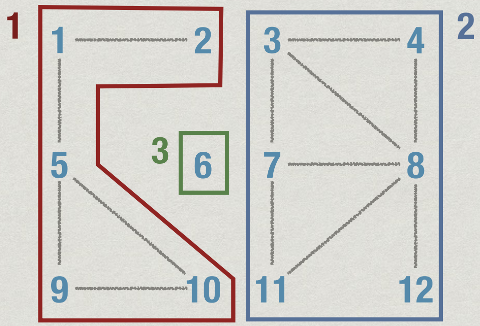
- Add a counter `comp` to number components
- Increment counter each time a fresh BFS/DFS starts
- Label each visited node `j` with `component[j] = comp`

Connected components



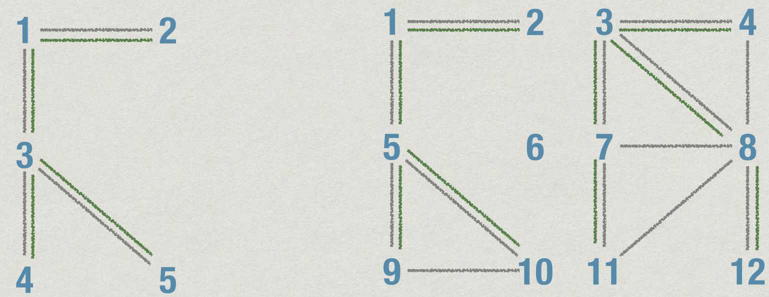
- Add a counter `comp` to number components
- Increment counter each time a fresh BFS/DFS starts
- Label each visited node `j` with `component[j] = comp`

Connected components



- Add a counter `comp` to number components
- Increment counter each time a fresh BFS/DFS starts
- Label each visited node `j` with `component[j] = comp`

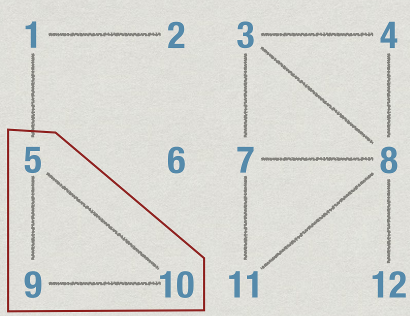
BFS tree



- Edges explored by BFS form a tree
  - Acyclic graph = connected, with n-1 edges

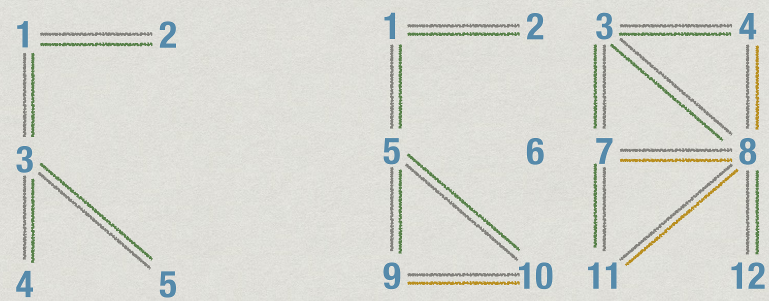
Cycles

Acyclic graph



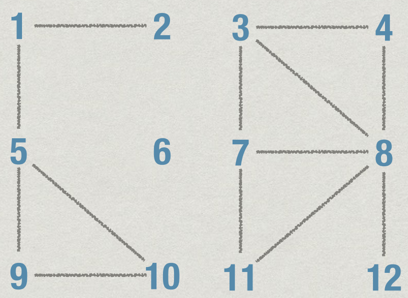
Graph with cycles

BFS tree

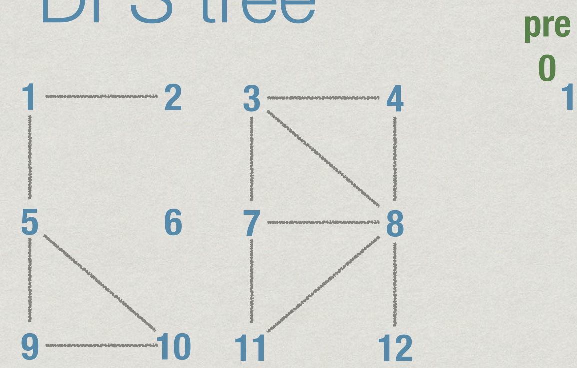


- Edges explored by BFS form a tree
  - Acyclic graph = connected, with n-1 edges
- Any non-tree edge generates a cycle

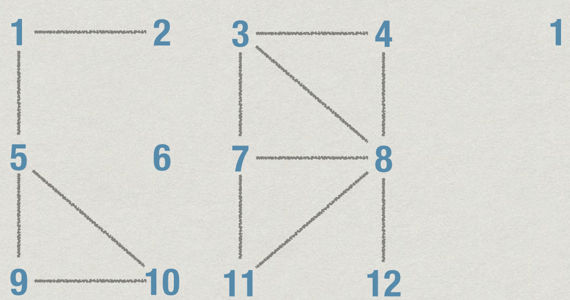
DFS tree



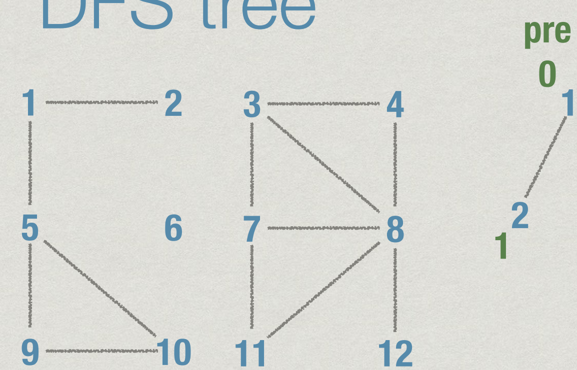
DFS tree



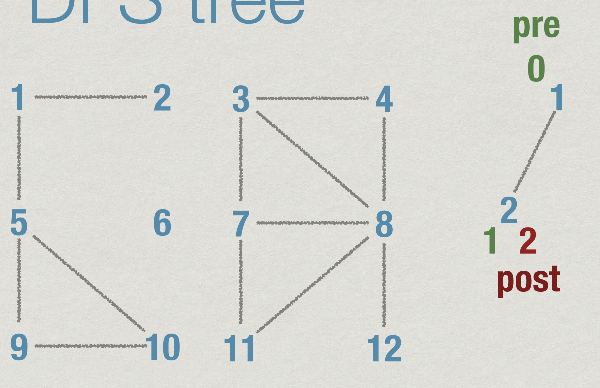
DFS tree



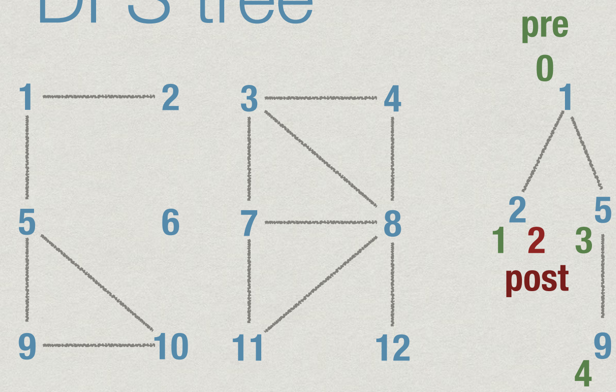
DFS tree



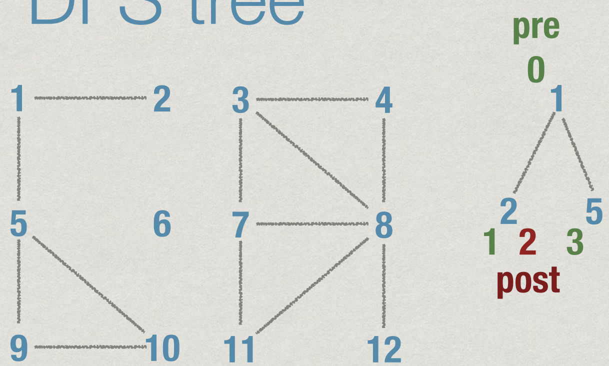
DFS tree



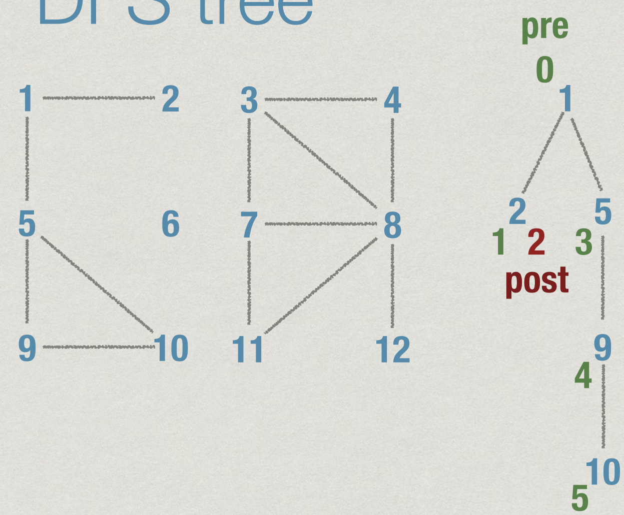
DFS tree



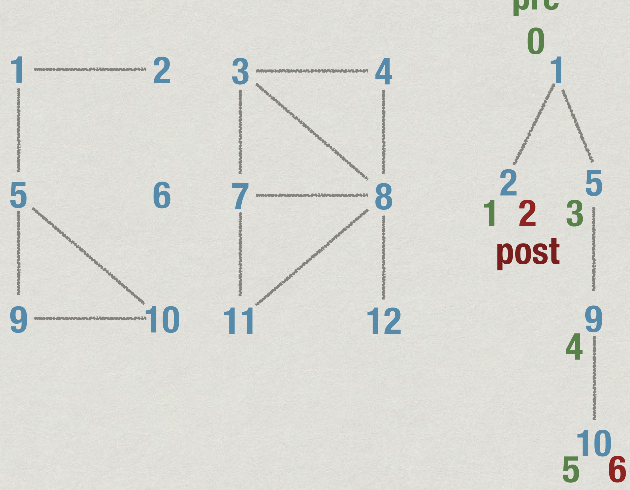
DFS tree



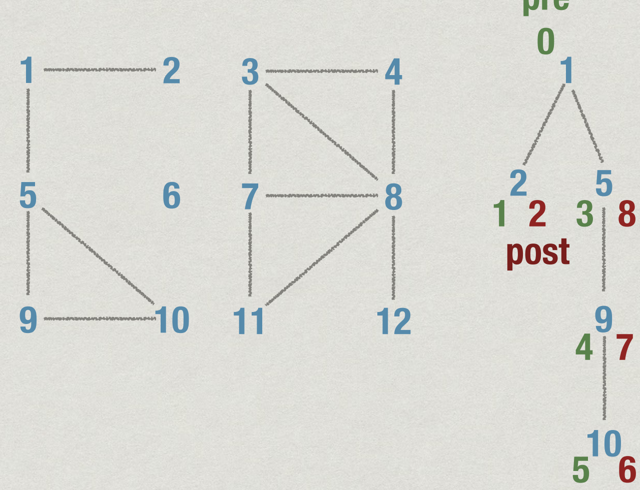
DFS tree



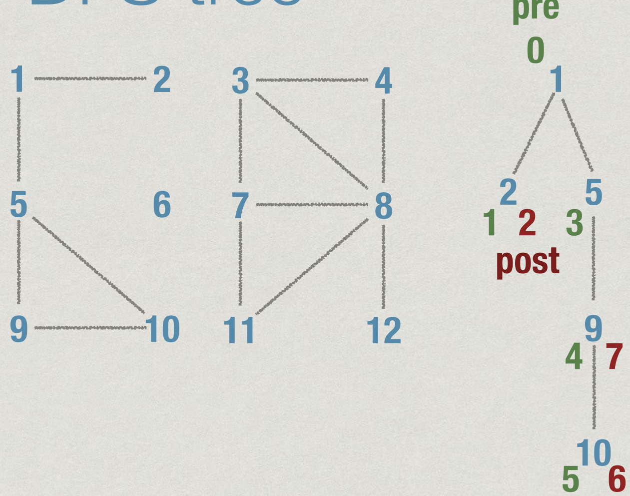
DFS tree



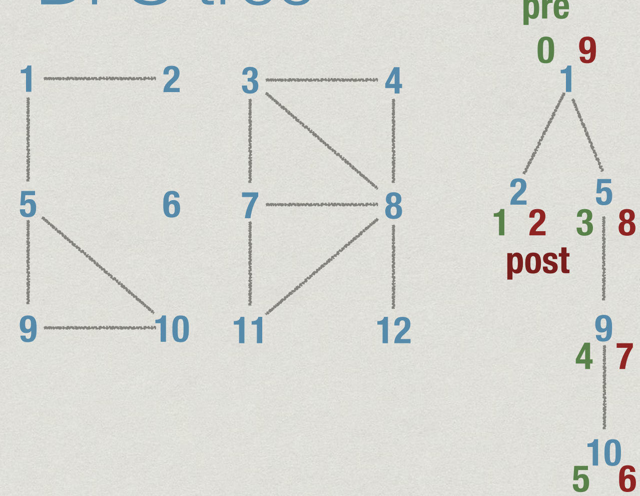
DFS tree



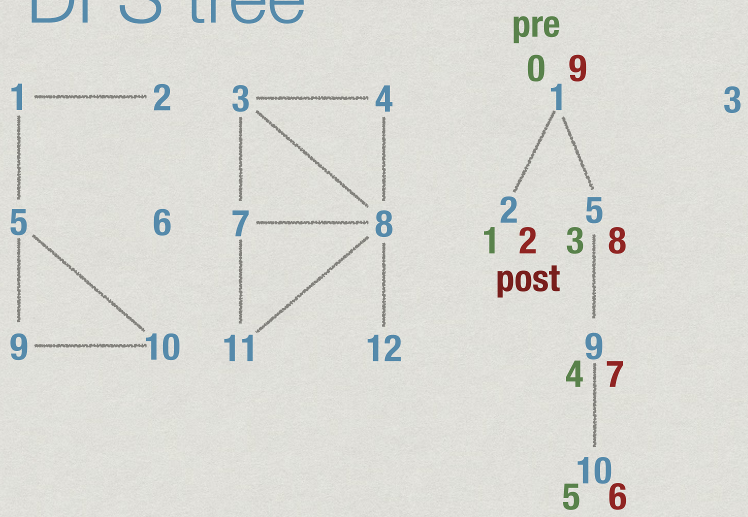
DFS tree



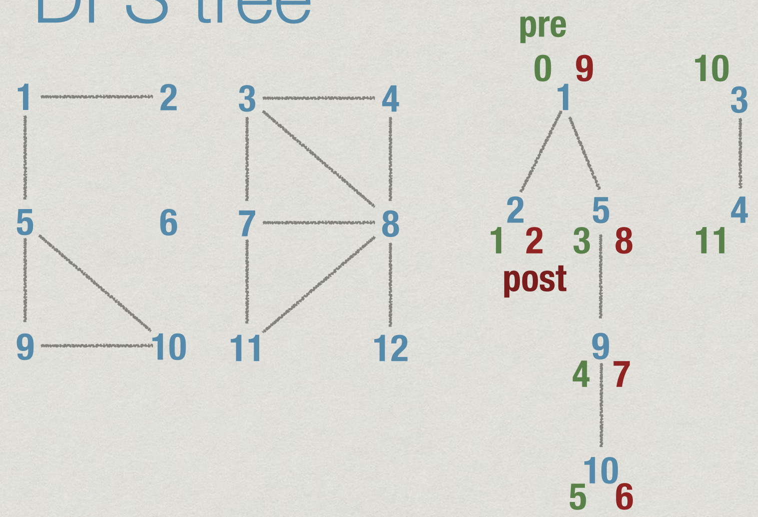
DFS tree



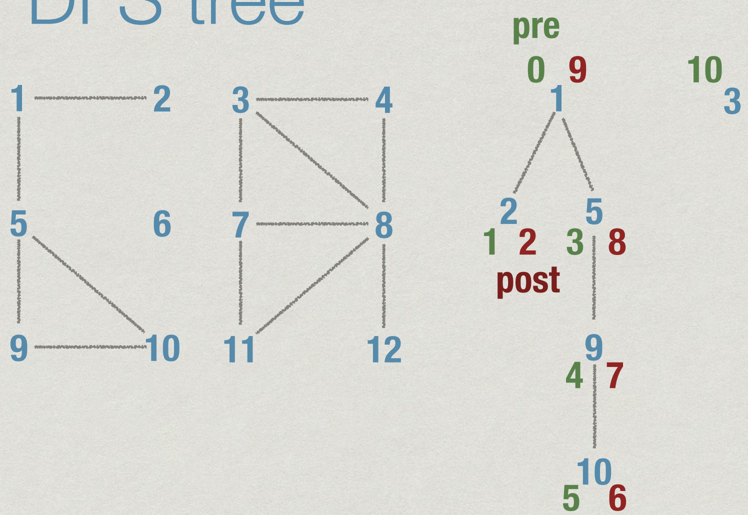
DFS tree



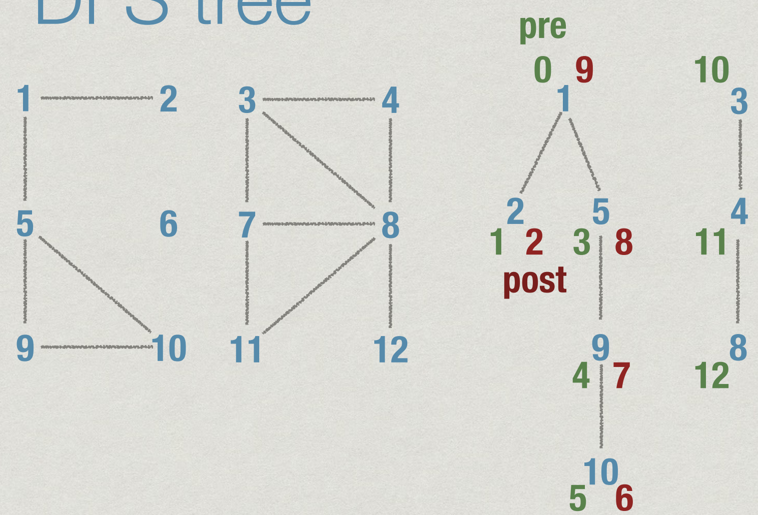
DFS tree



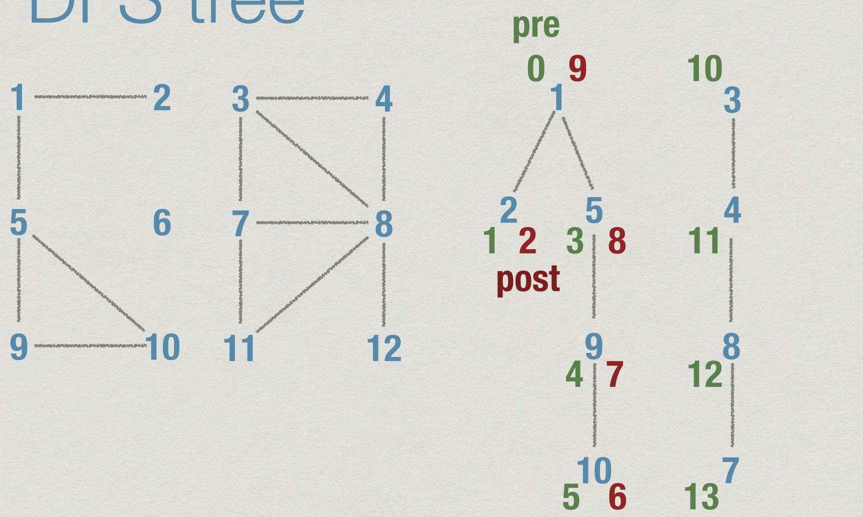
DFS tree



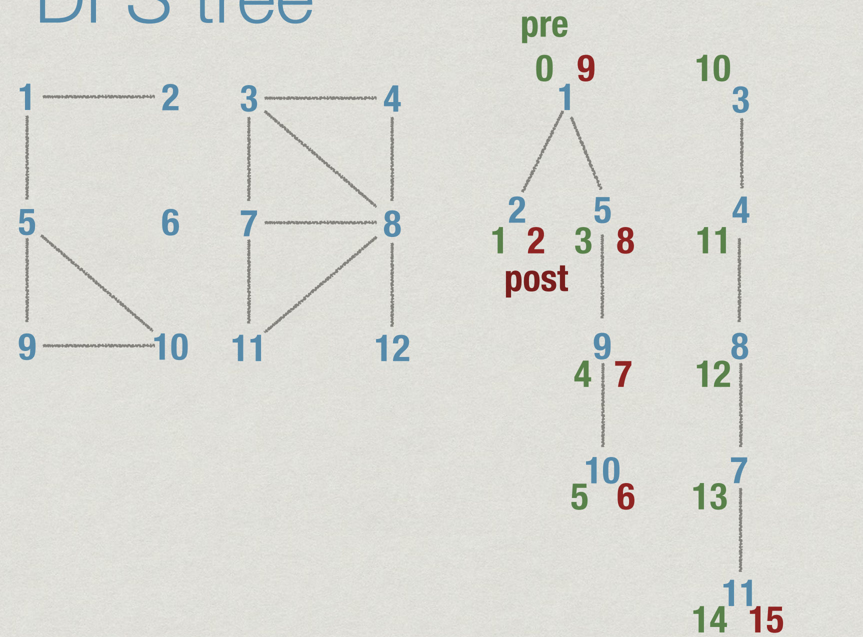
DFS tree



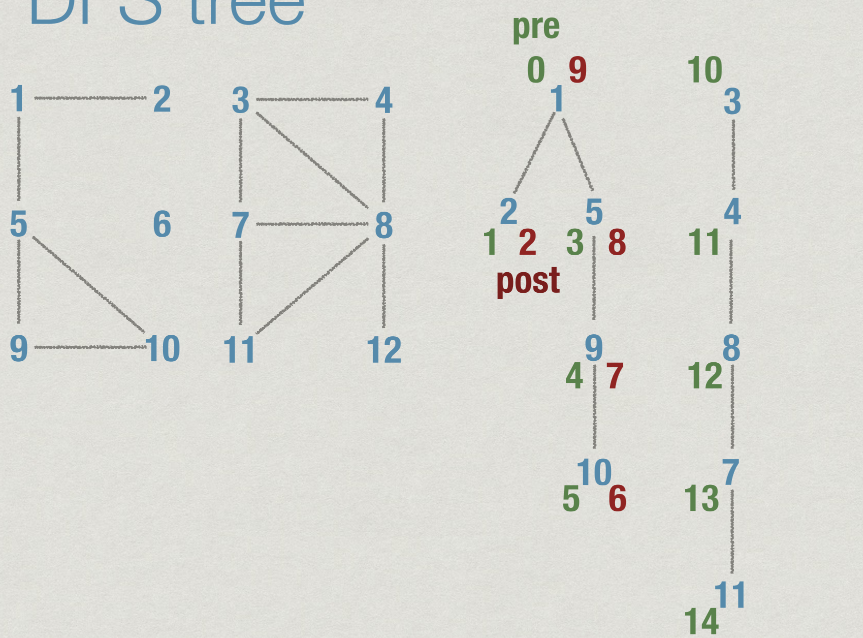
DFS tree



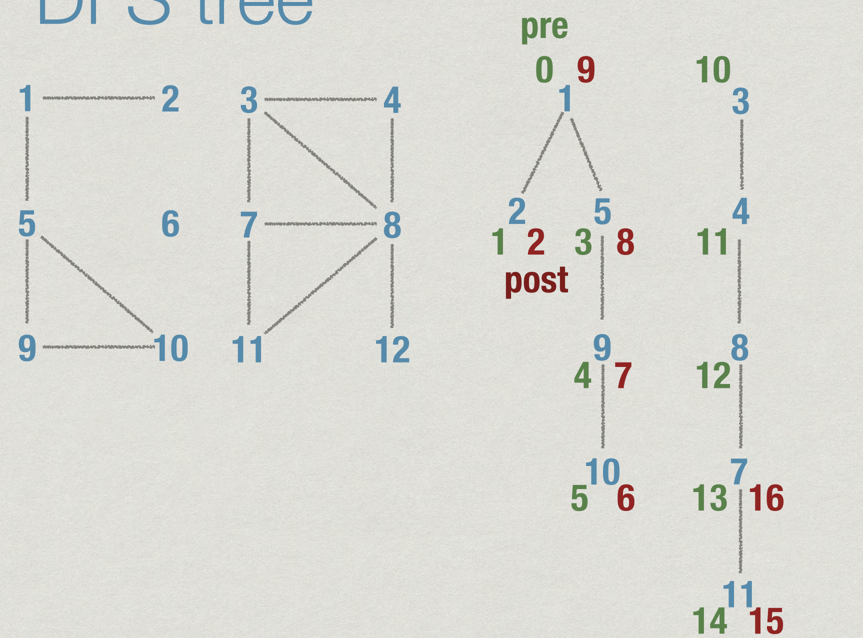
DFS tree



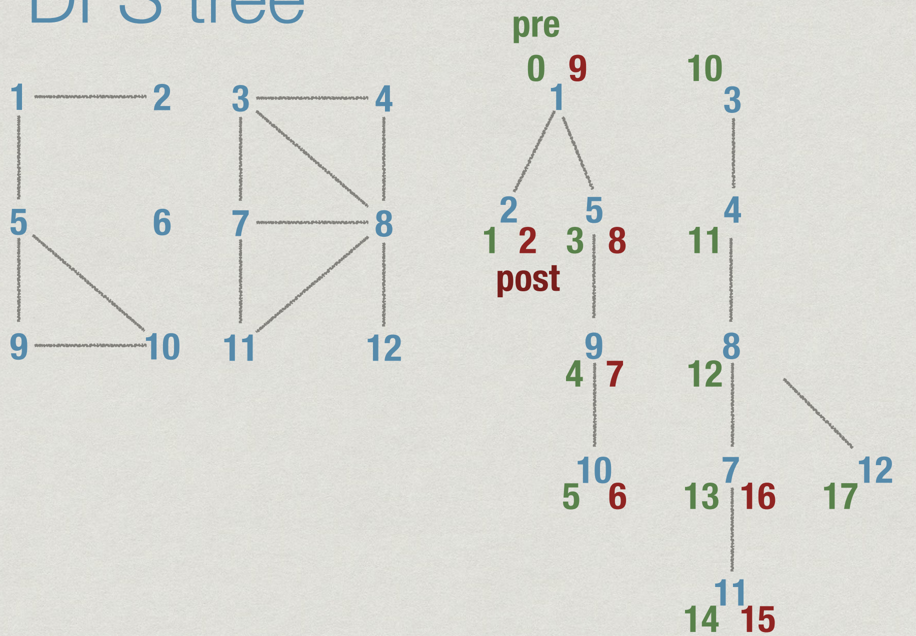
DFS tree



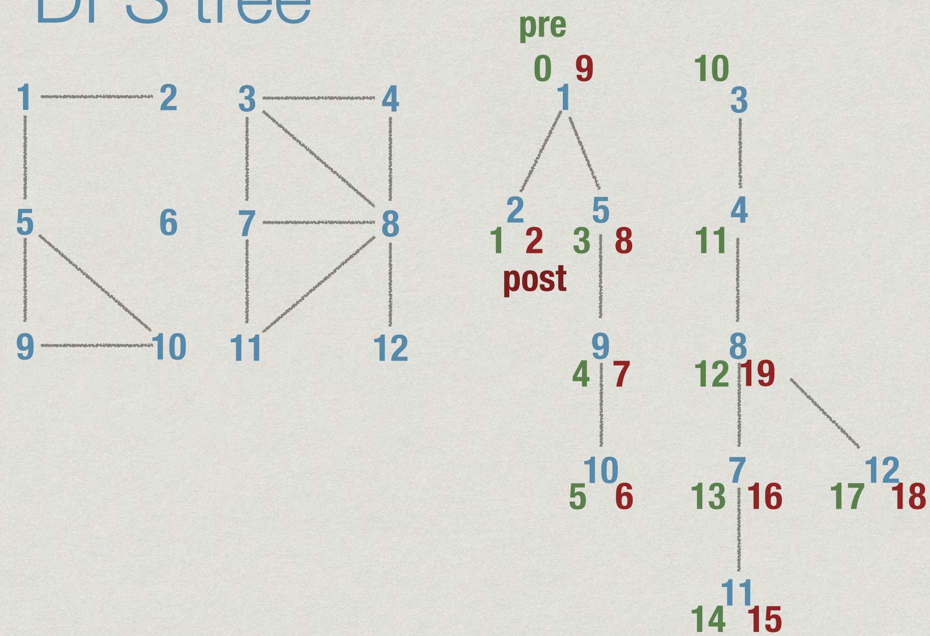
DFS tree



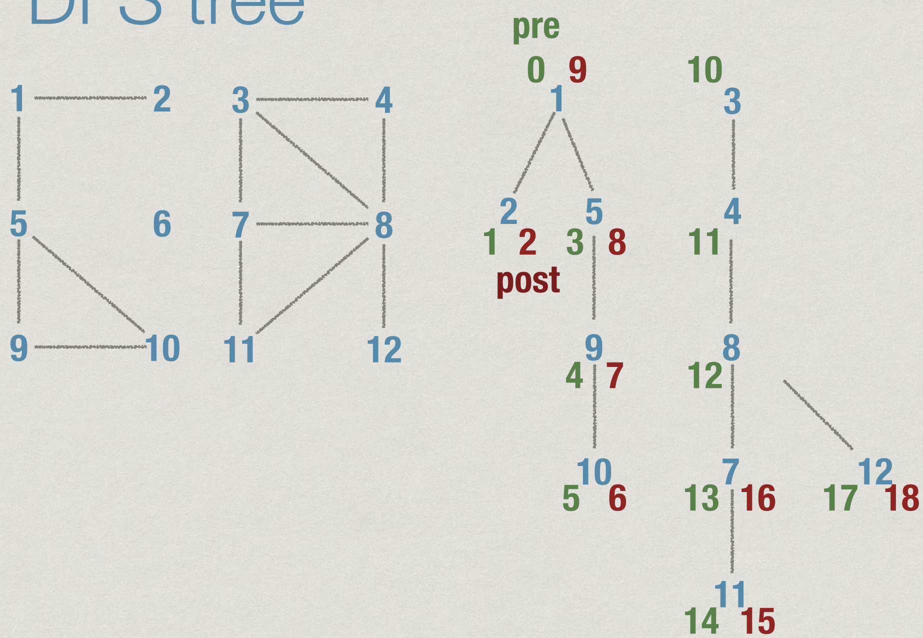
DFS tree



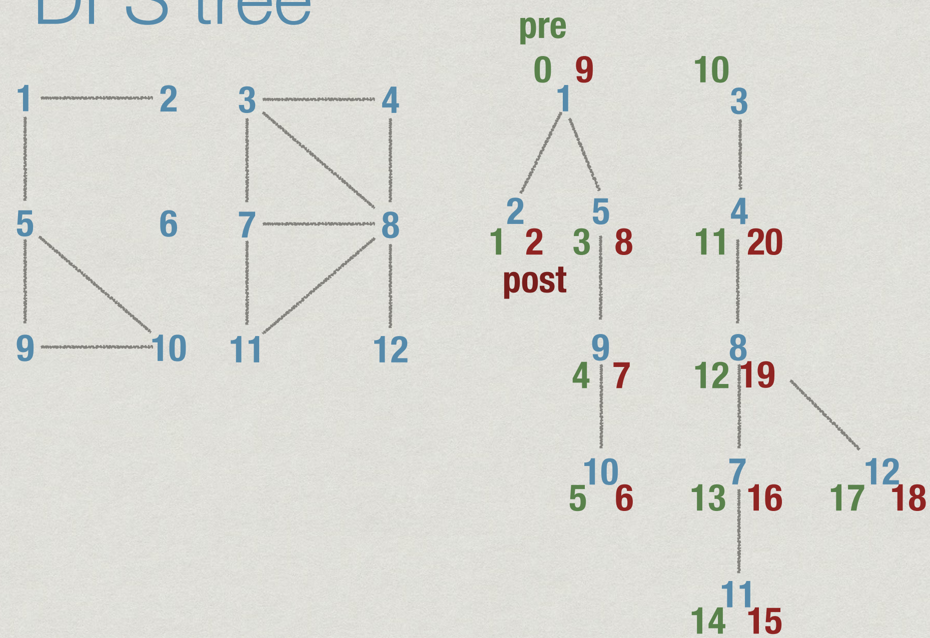
DFS tree



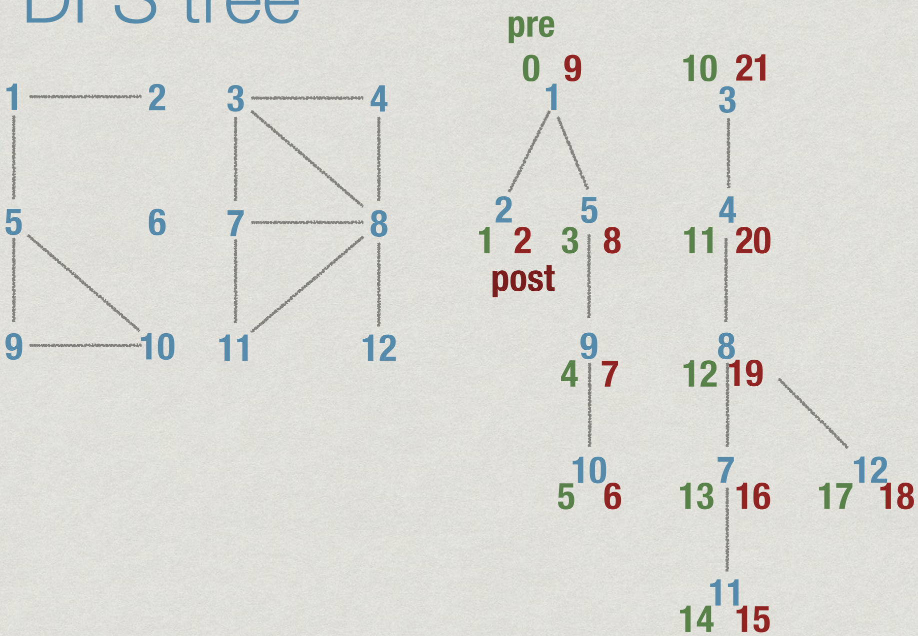
DFS tree



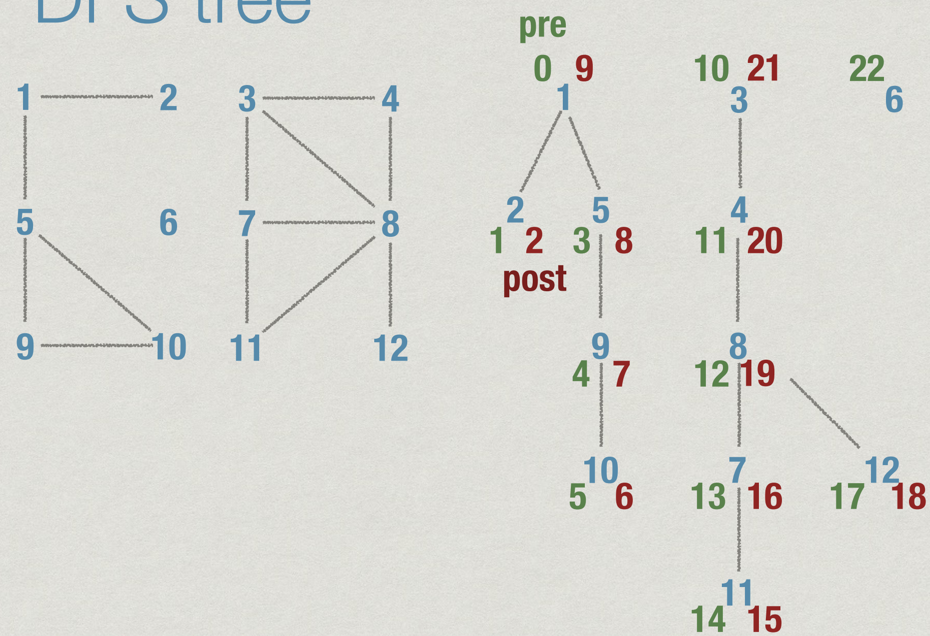
DFS tree



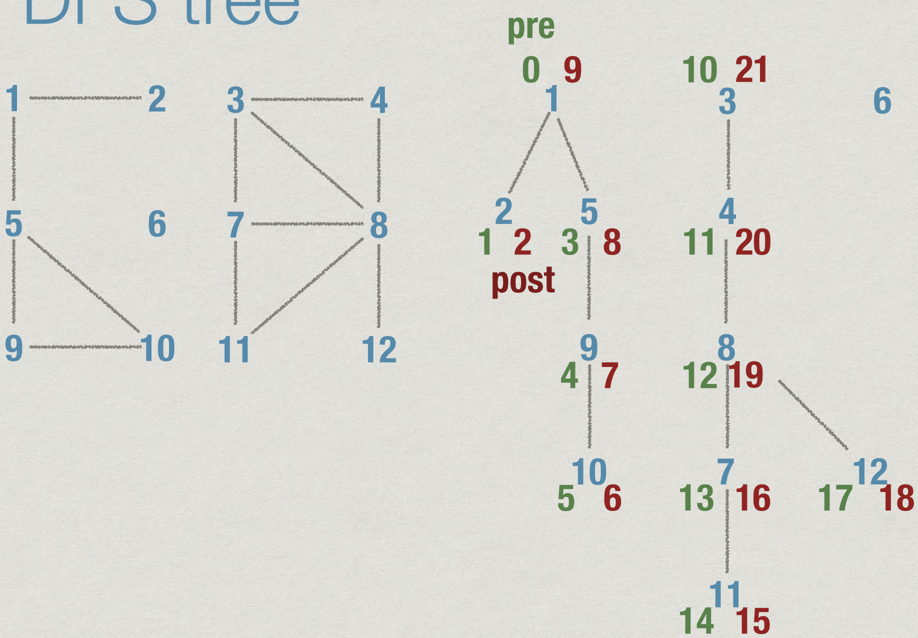
DFS tree



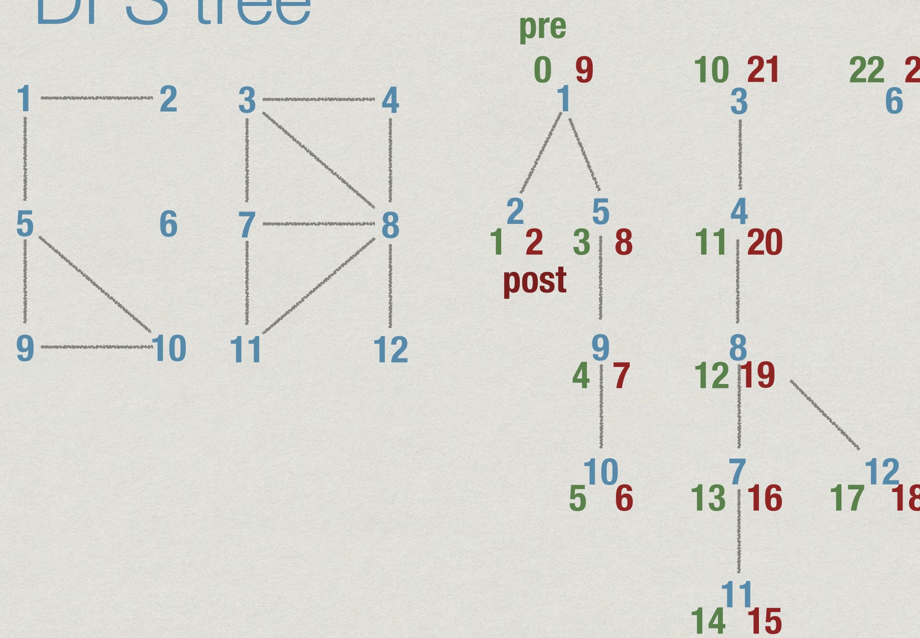
DFS tree



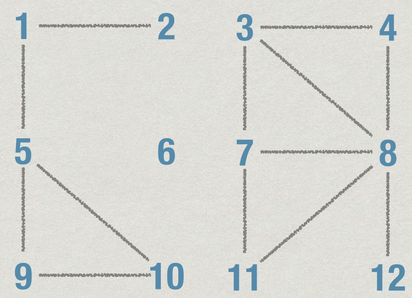
DFS tree

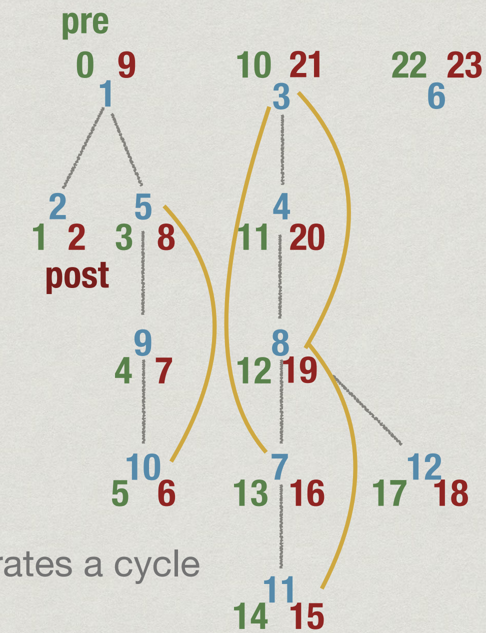


DFS tree



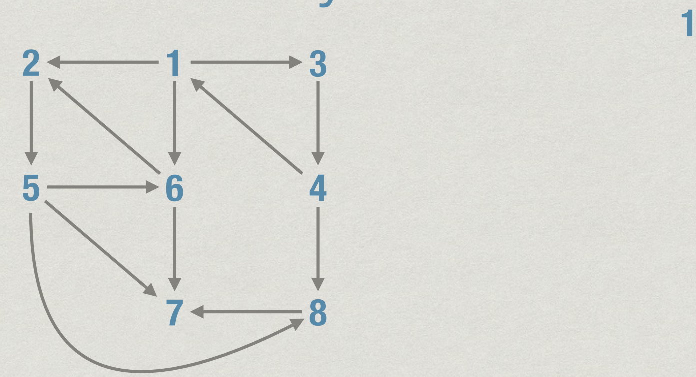
DFS tree



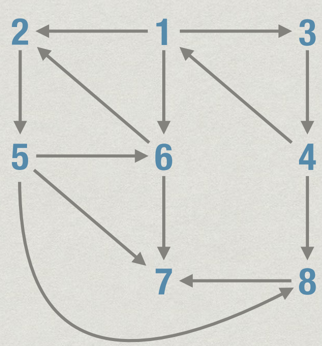


- Any non-tree edge generates a cycle

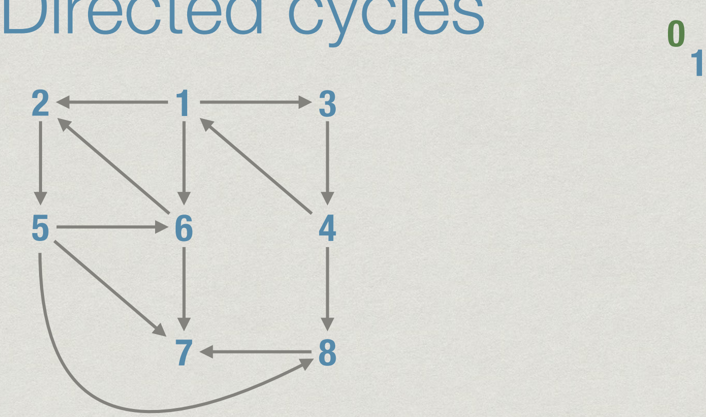
Directed cycles



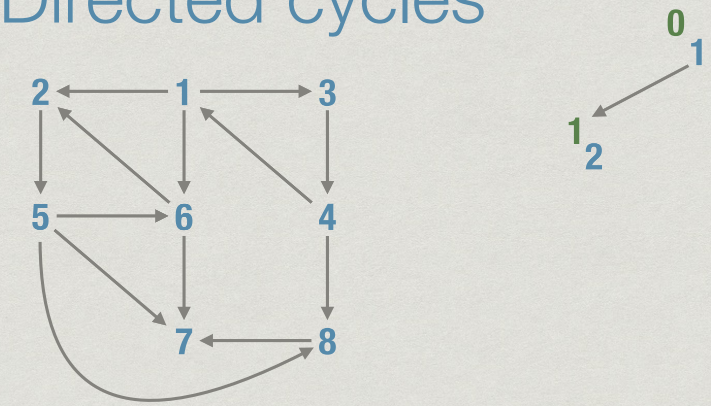
Directed cycles



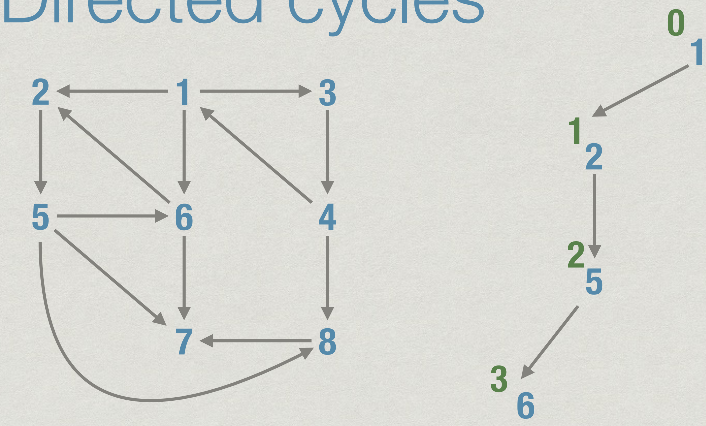
Directed cycles



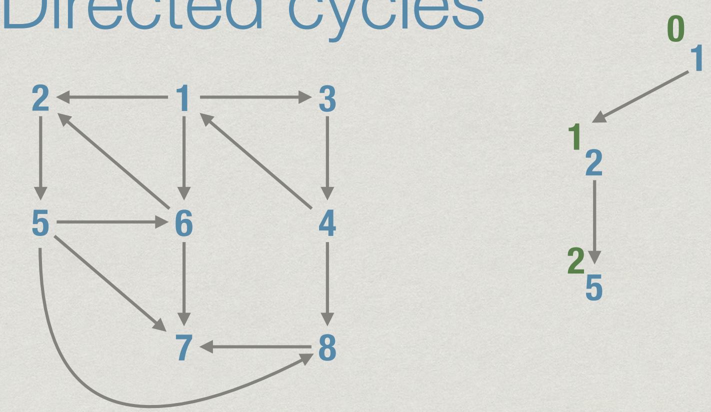
Directed cycles



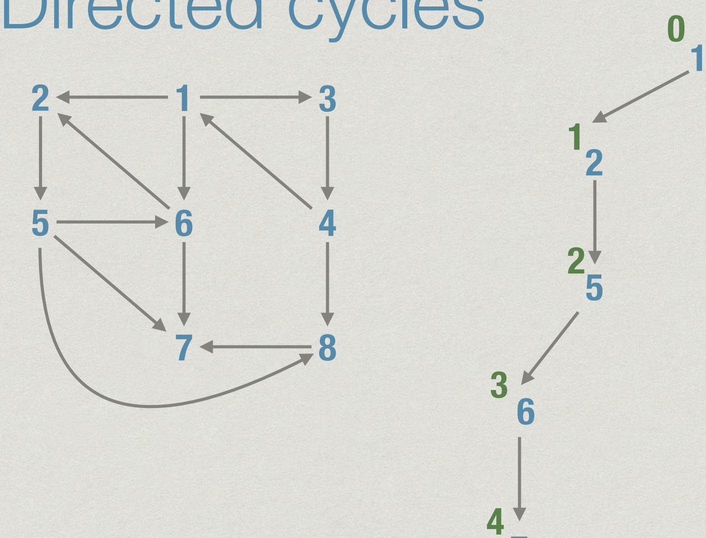
Directed cycles



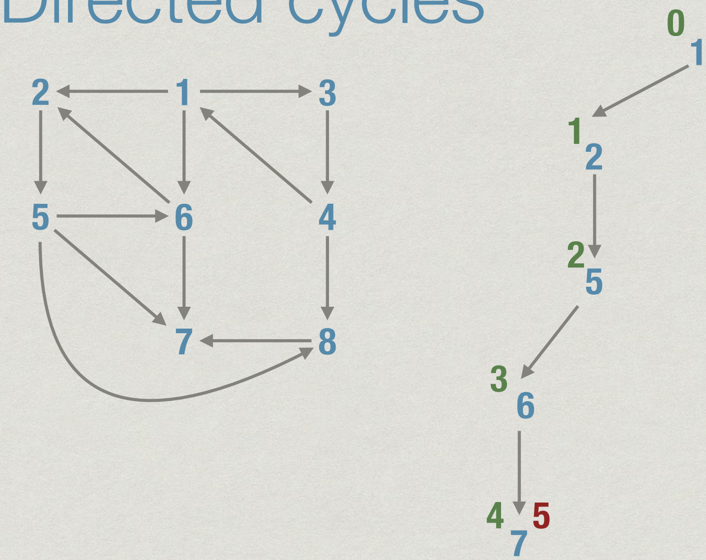
Directed cycles



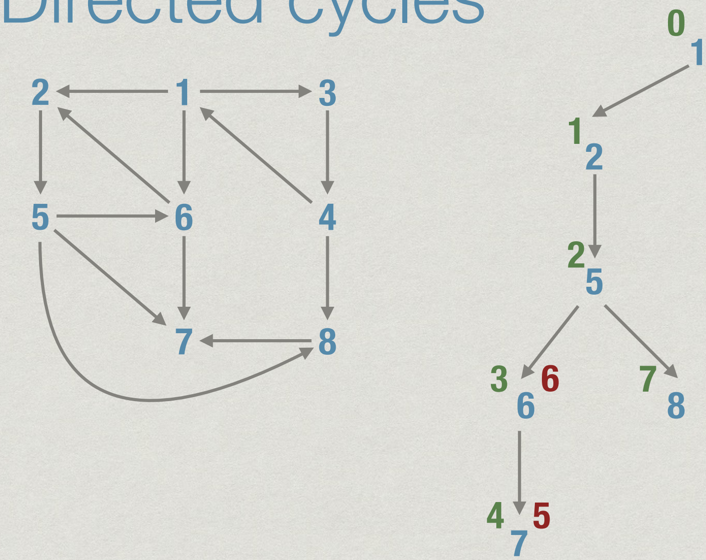
Directed cycles



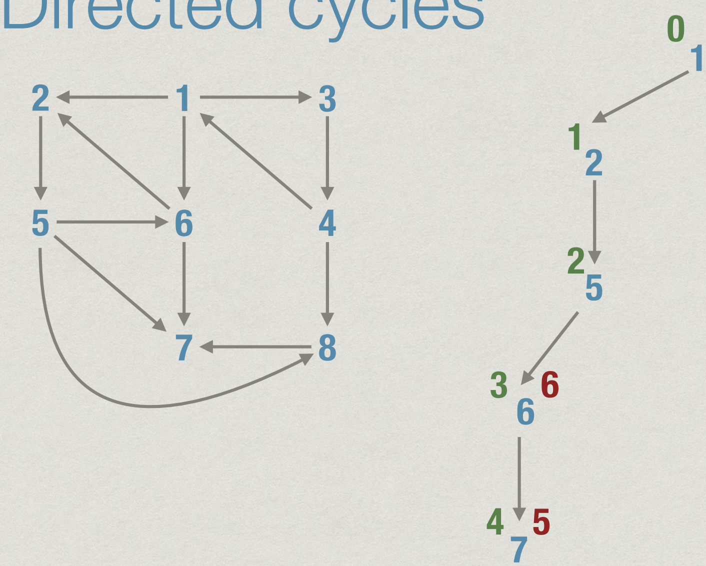
Directed cycles



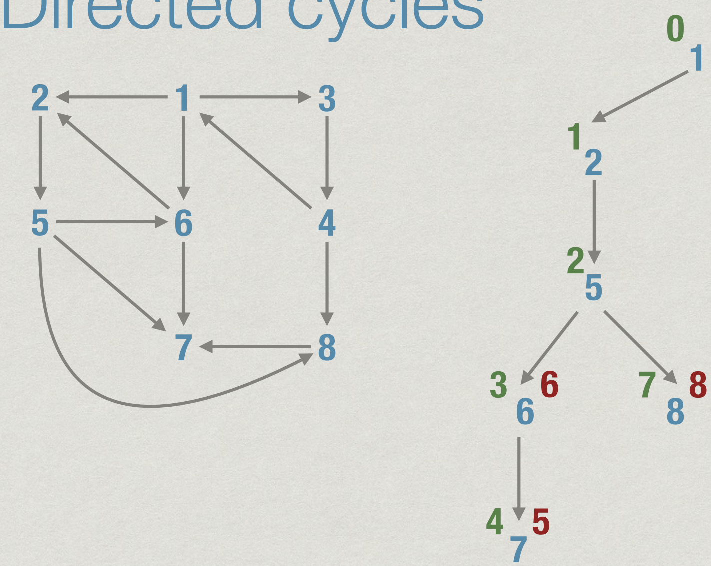
Directed cycles



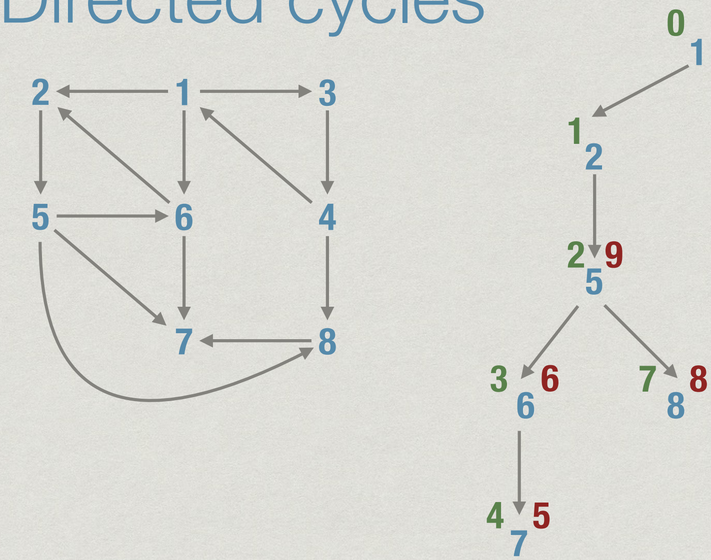
Directed cycles



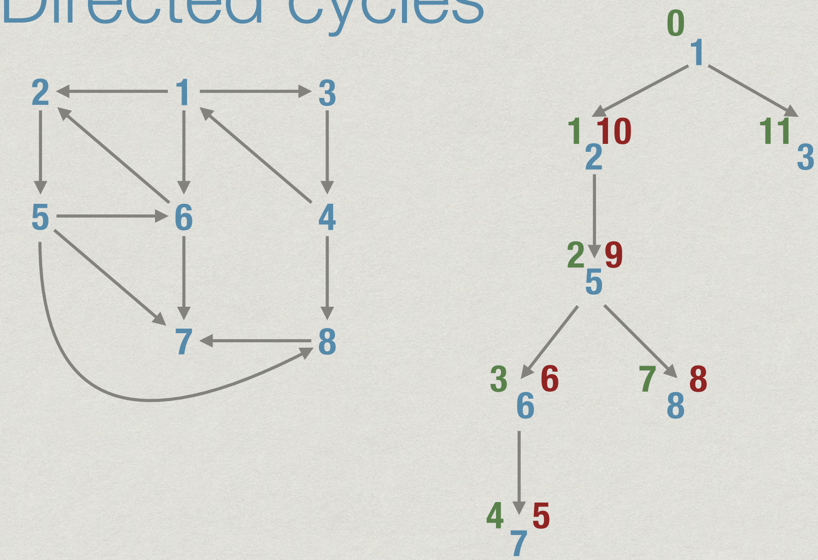
Directed cycles



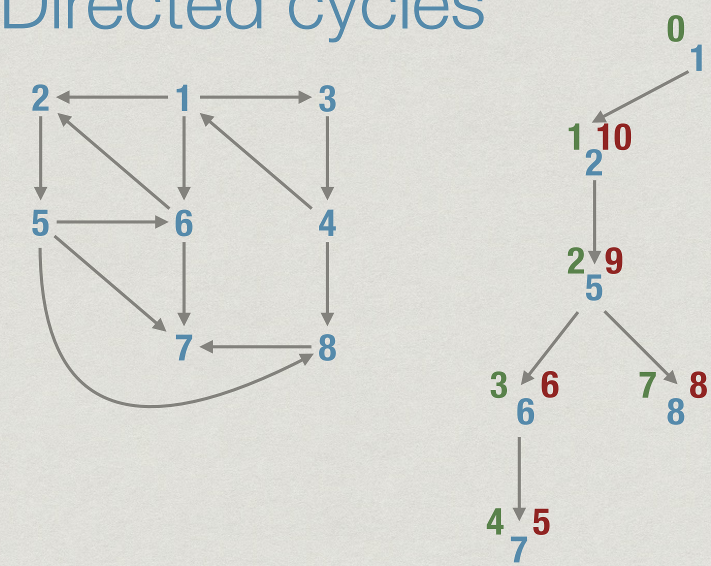
Directed cycles



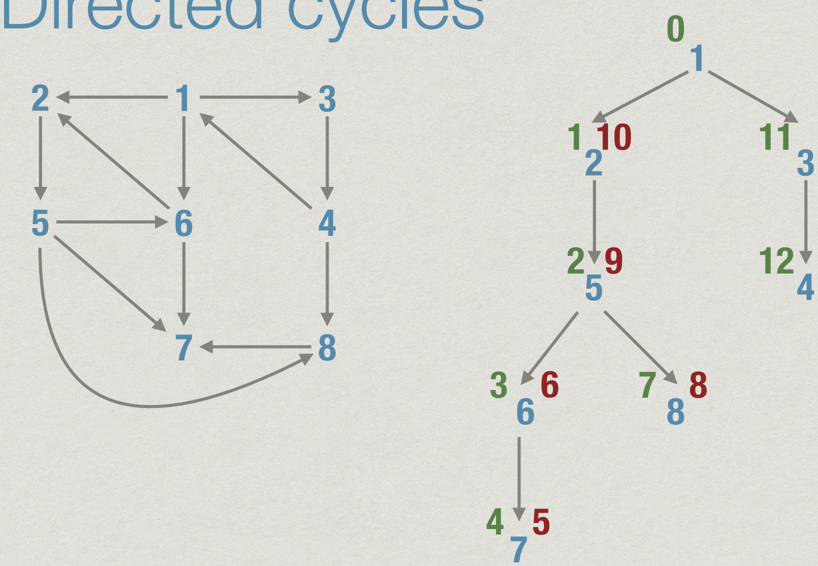
Directed cycles



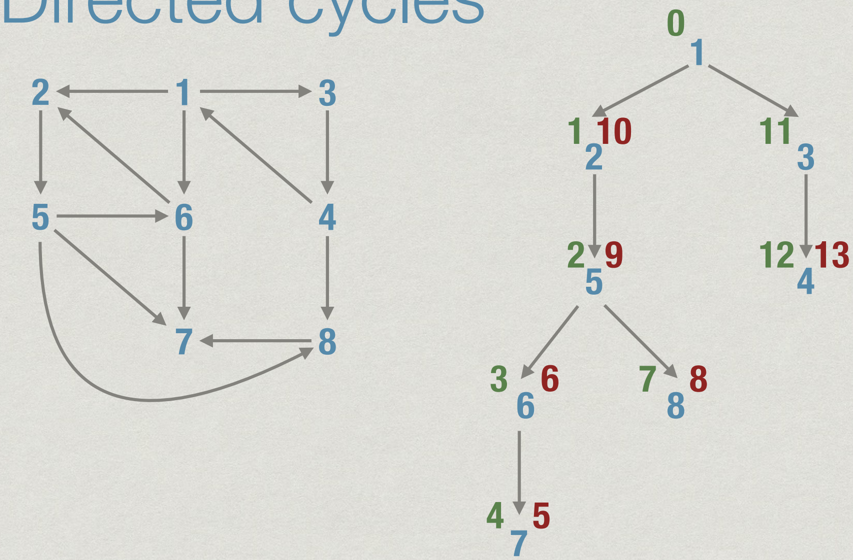
Directed cycles



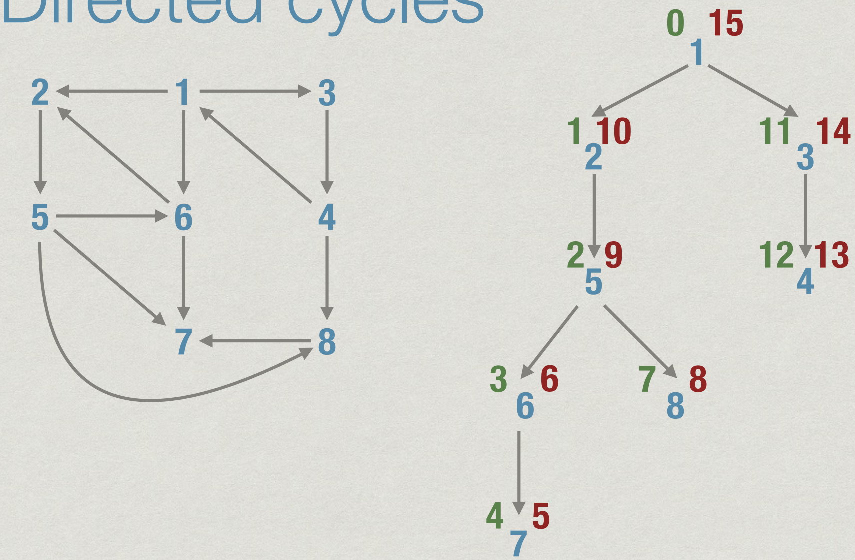
Directed cycles



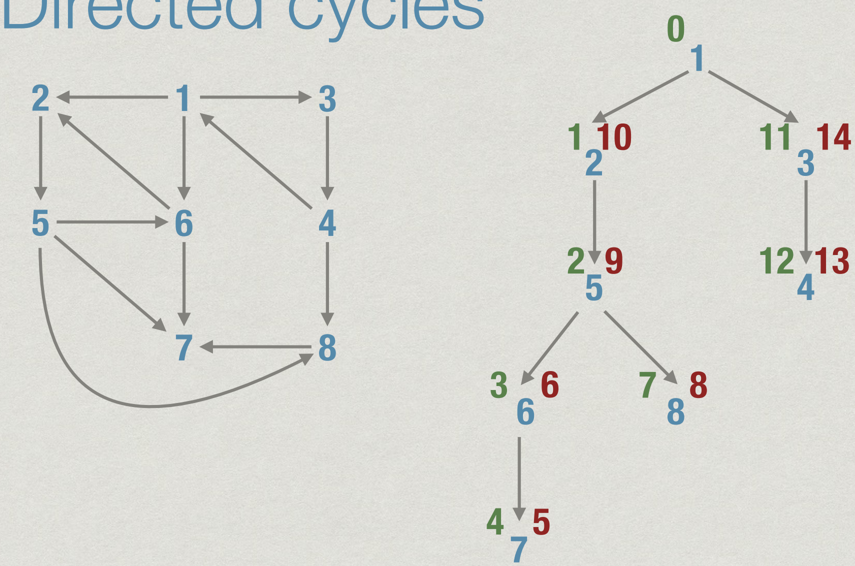
Directed cycles



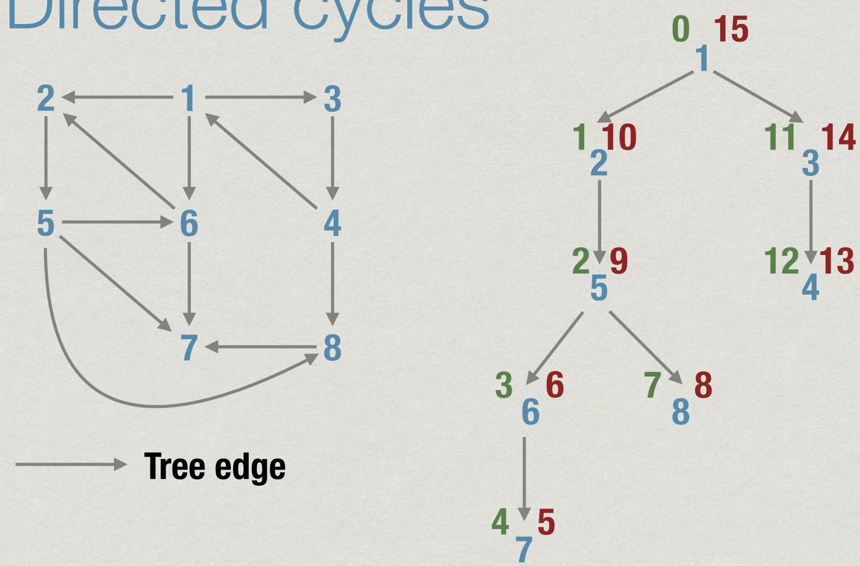
Directed cycles



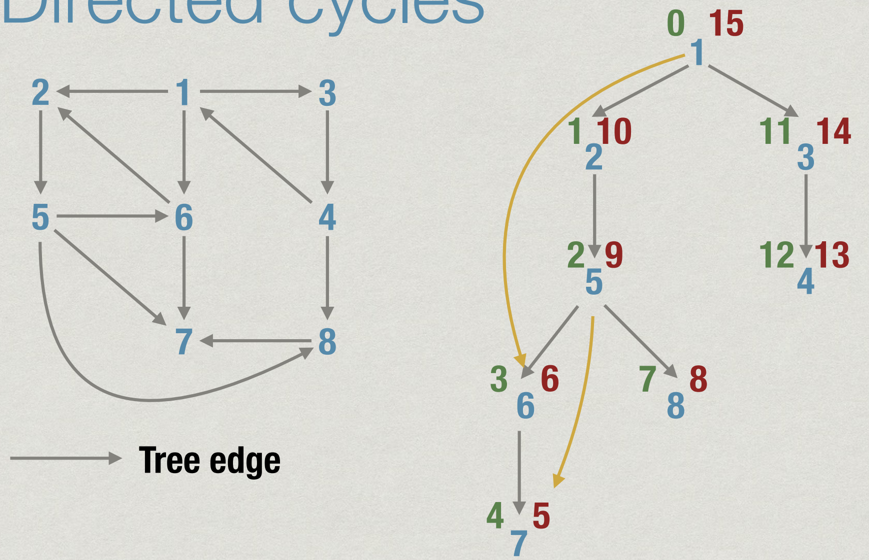
Directed cycles



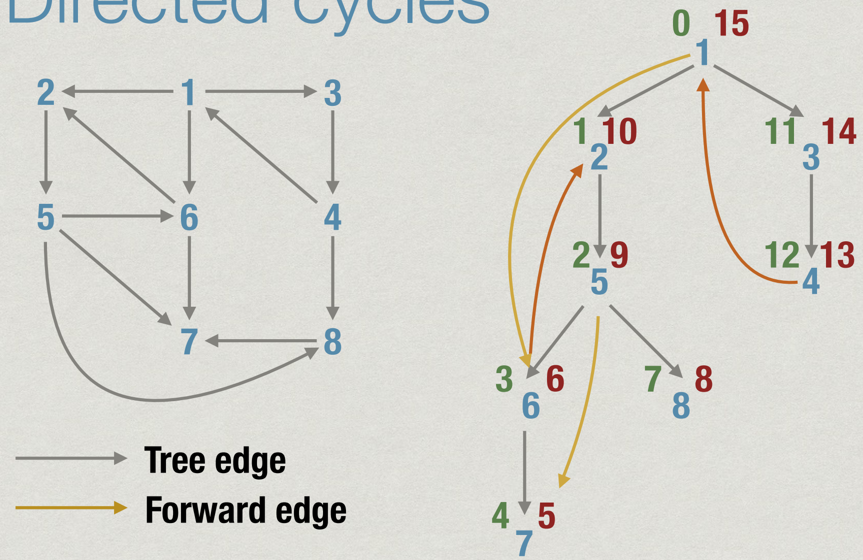
Directed cycles



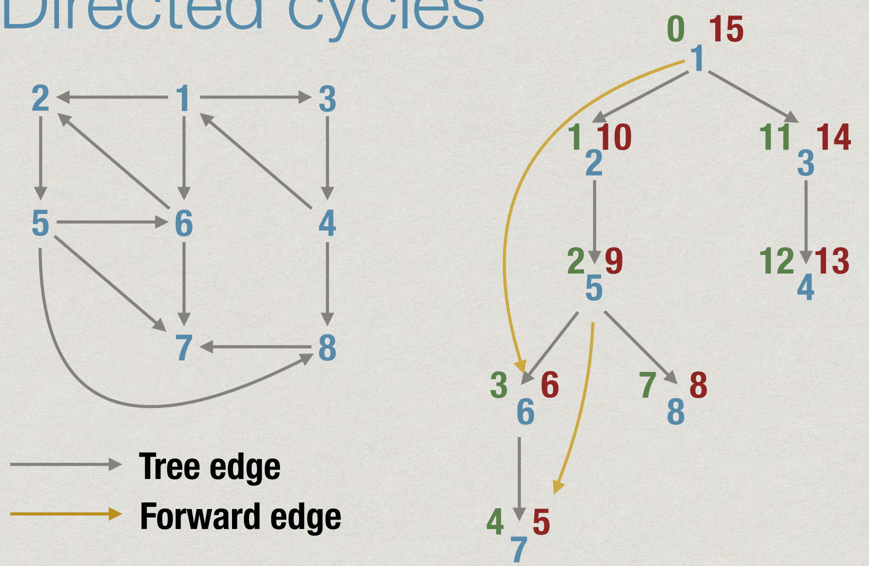
Directed cycles



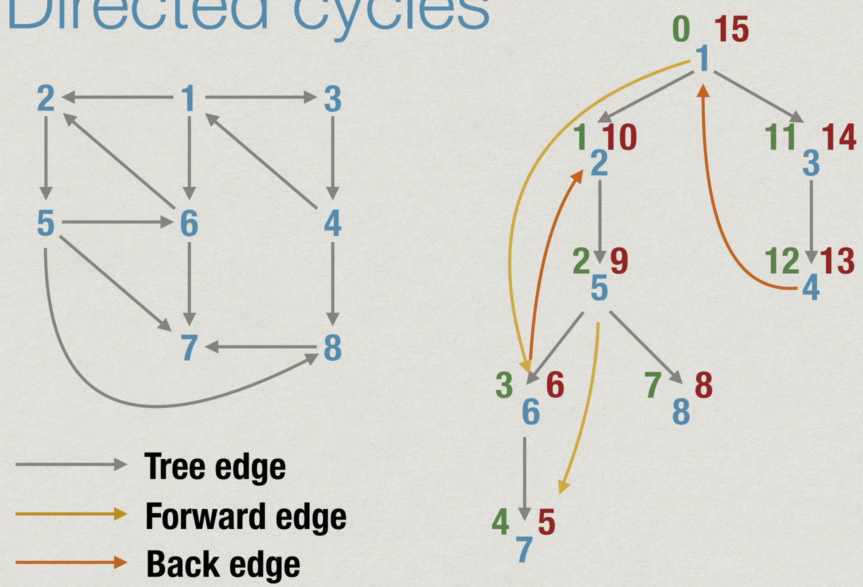
Directed cycles



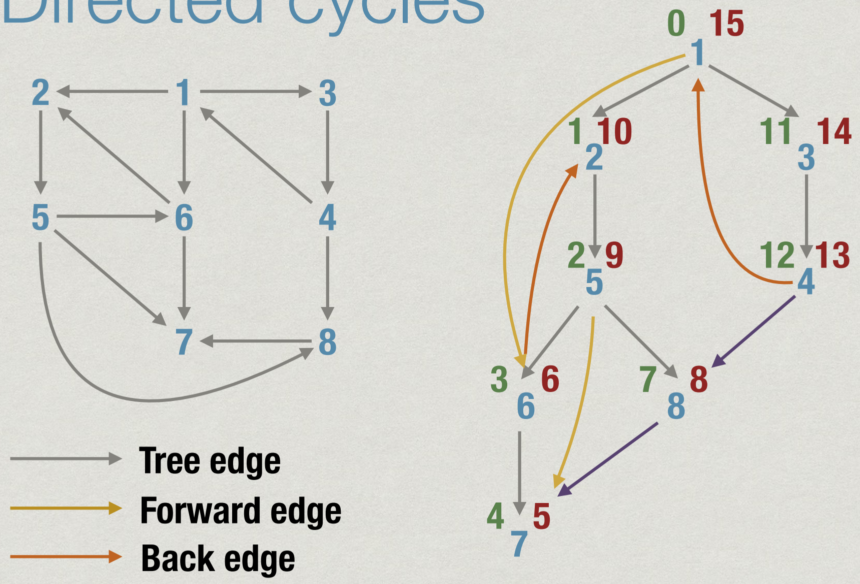
Directed cycles



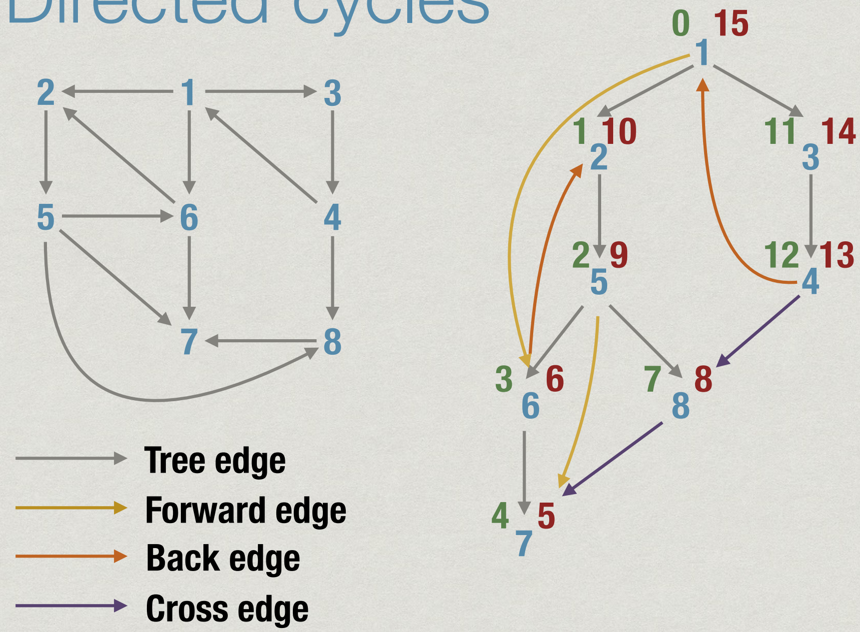
Directed cycles



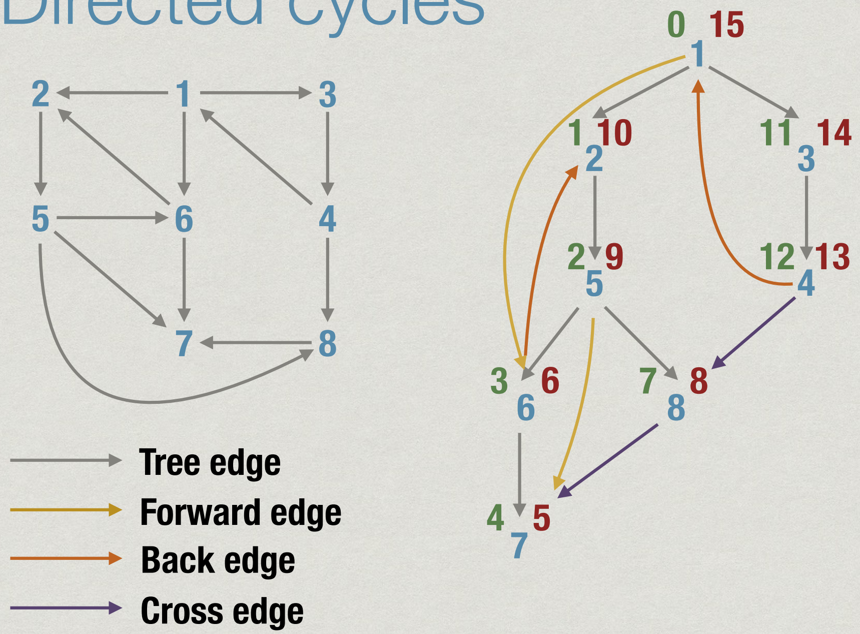
Directed cycles



Directed cycles



Directed cycles



Directed cycles

- A directed graph has a cycle if and only if DFS reveals a back edge
- Can classify edges using pre and post numbers
  - Tree/Forward edge (u,v) :
    Interval [pre(u),post(u)] contains [(pre(v),post(v)]
  - Backward edge (u,v):
    Interval [pre(v),post(v)] contains [(pre(u),post(u)]
  - Cross edge (u,v):
    Intervals [(pre(u),post(u)] and [(pre(v),post(v)] disjoint

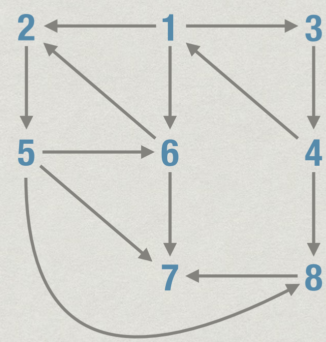
Directed acyclic graphs

- Directed graphs without cycles are useful for modelling dependencies
  - Courses with prerequisites
  - Edge (Algebra,Calculus) indicates that Algebra is a prerequisite for Calculus
- Will look at Directed Acyclic Graphs (DAGs) soon

Connectivity in directed graphs

- Need to take directions into account
- Nodes i and j are strongly connected if there is a path from i to j and a path from j to i
- Directed graph can be decomposed into strongly connected components (SCCs)
  - All pairs of nodes in an SCC are strongly connected

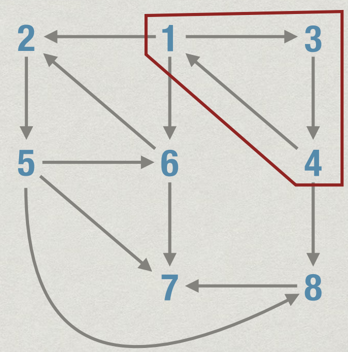
Computing SCCs



- DFS numbering (pre and post) can be used to compute SCCs

[Dasgupta, Papadimitriou,Vazirani]

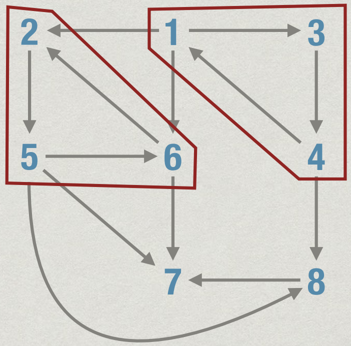
Computing SCCs



- DFS numbering (pre and post) can be used to compute SCCs

[Dasgupta, Papadimitriou,Vazirani]

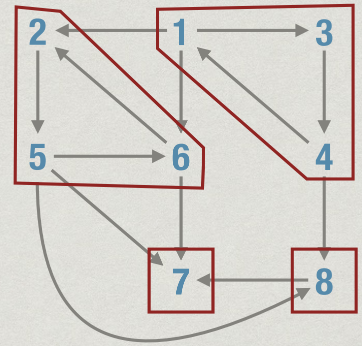
Computing SCCs



- DFS numbering (pre and post) can be used to compute SCCs

[Dasgupta, Papadimitriou,Vazirani]

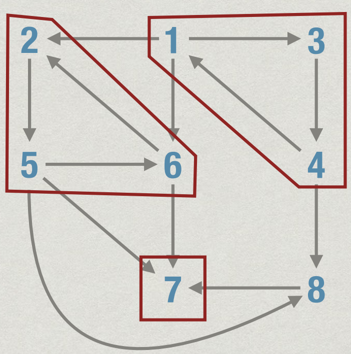
Computing SCCs



- DFS numbering (pre and post) can be used to compute SCCs

[Dasgupta, Papadimitriou,Vazirani]

Computing SCCs



- DFS numbering (pre and post) can be used to compute SCCs

[Dasgupta, Papadimitriou,Vazirani]

Other properties

- A number of other structural properties can be inferred from DFS numbering
- Articulation points (vertices)
  - Removing such a vertex disconnects the graph
- Bridges (edges)
  - Removing such an edge disconnects the graph